Section-A

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

Answer:

Spring Boot simplifies the development of Spring-based applications by providing a range of features that make it easier to get started with minimal configuration. Here are some reasons to choose Spring Boot over the traditional Spring Framework:

Auto Configuration: Spring Boot automatically configures your application based on the dependencies present in the classpath. This reduces the need for extensive configuration.

Embedded Servers: It provides embedded servers like Tomcat, Jetty, and Undertow, allowing you to run applications standalone without needing an external server.

Production Ready: Features like health checks, metrics, and application monitoring are built-in, making it easier to deploy production-ready applications.

Convention over Configuration: Spring Boot uses sensible defaults and configurations, which speeds up development.

Example:

java

Copy code

```
@SpringBootApplication
public class MyApplication {
   public static void main(String[] args) {
      SpringApplication.run(MyApplication.class, args);
   }
}
```

2. What all Spring Boot starters you have used or what all modules have you worked on?

Answer:

Spring Boot starters are dependency descriptors that simplify adding libraries to your project. Common starters include:

spring-boot-starter-web: For building web applications, including RESTful applications using Spring MVC.

spring-boot-starter-data-jpa: For working with JPA and Hibernate.

spring-boot-starter-security: For adding Spring Security to your application.

spring-boot-starter-test: For testing Spring Boot applications with JUnit and other testing libraries.

Example:

xml

Copy code

```
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-web</artifactId>
</dependency>
```

3. How will you run a Spring Boot application?

Answer:

You can run a Spring Boot application in several ways:

Command Line: Use mvn spring-boot:run if you're using Maven, or ./gradlew bootRun if you're using Gradle.

IDE: Run the main class containing the @SpringBootApplication annotation directly from an IDE like IntelliJ IDEA or Eclipse.

Executable JAR: Build an executable JAR with mvn clean package or ./gradlew build and run it using java -jar target/myapp.jar.

Example:

bash

Copy code

```
java -jar myapp.jar
```

4. What is the purpose of the @SpringBootApplication annotation in a Spring Boot application?

Answer:

@SpringBootApplication is a convenience annotation that combines three annotations:

@Configuration: Marks the class as a source of bean definitions.

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

@ComponentScan: Enables component scanning for the application context.

Example:

java

Copy code

```
@SpringBootApplication
public class MyApplication {
  public static void main(String[] args) {
    SpringApplication.run(MyApplication.class, args);
  }
}
```

5. Can I directly use the above three annotations in my main class instead of using @SpringBootApplication annotation? If yes, will my application work as expected?

Answer:

Yes, you can use @Configuration, @EnableAutoConfiguration, and @ComponentScan annotations individually in place of @SpringBootApplication. Your application will work as expected with these annotations.

Example:

java

Copy code

```
@Configuration
@EnableAutoConfiguration
@ComponentScan
public class MyApplication {
  public static void main(String[] args) {
    SpringApplication.run(MyApplication.class, args);
  }
}
```

6. What is AutoConfiguration in Spring Boot?

Answer:

AutoConfiguration is a feature in Spring Boot that automatically configures your application based on the dependencies present in the classpath. It helps reduce the need for manual configuration and sets up beans and configurations that are commonly used.

Example:

If you add spring-boot-starter-data-jpa to your project, Spring Boot will automatically configure a DataSource, EntityManagerFactory, and TransactionManager based on your properties.

7. How can you disable a specific auto-configuration class in Spring Boot?

Answer:

You can disable a specific auto-configuration class by using the exclude attribute of the @SpringBootApplication annotation or by setting it in application.properties or application.yml.

Example:

java

Copy code

```
@SpringBootApplication(exclude = {DataSourceAutoConfiguration.class})
public class MyApplication {
   public static void main(String[] args) {
      SpringApplication.run(MyApplication.class, args);
   }
}
```

In application.properties:

properties

Copy code

```
spring.autoconfigure.exclude=org.springframework.boot.autoconfigure.jdbc.DataSourceAutoConfiguration
```

8. How can you customize the default configuration in Spring Boot?

Answer:

You can customize default configurations by:

Using application.properties or application.yml: Override default values by providing your custom configurations.

Creating configuration classes: Define beans and configurations in Java classes annotated with @Configuration.

Example (application.properties):

properties

Copy code

server.port=8081

Example (Configuration Class):

java

Copy code

```
@Configuration
public class MyConfig {
   @Bean
   public MyBean myBean() {
      return new MyBean();
   }
}
```

9. How does the run() method in Spring Boot work internally?

Answer:

The run() method of SpringApplication performs several tasks:

Create an ApplicationContext: It initializes the Spring application context.

Register Context Initializers: It applies context initializers to prepare the context.

Configure the Application: It configures the application based on the environment and properties.

Run the Application: It starts the application by running the main application class with the provided arguments.

Example:

java

Copy code

```
SpringApplication.run(MyApplication.class, args);
```

10. What is CommandLineRunner in Spring Boot?

Answer:

CommandLineRunner is a functional interface in Spring Boot that allows you to run specific code after the application context has been initialized. It's used to execute code at startup.

Example:

java

Copy code

```
@Component
public class MyStartupRunner implements CommandLineRunner {
   @Override
   public void run(String... args) throws Exception {
      System.out.println("Application started with command-line arguments: " +
Arrays.toString(args));
   }
}
```

Section-B

1. Can you explain the purpose of stereotype annotations in the Spring framework?

Answer:

Stereotype annotations are used to define the role or purpose of a component in the Spring application. They help in automatic component scanning and bean creation.

@Component: Generic stereotype for any Spring-managed component.

@Service: Indicates a service layer component.

@Repository: Indicates a data access layer component.

@Controller: Indicates a web controller.

Example:

java

Copy code

```
@Service
public class MyService {
   // Business logic
}
```

2. How can you define a bean in the Spring Framework?

Answer:

You can define a bean in Spring using several methods:

Using @Bean annotation in a @Configuration class.

Using @Component, @Service, @Repository, @Controller annotations.

Example (@Bean annotation):

java

Copy code

```
@Configuration
public class AppConfig {
   @Bean
   public MyBean myBean() {
      return new MyBean();
```

```
    }
}
```

Example (using @Component):

```
java
Copy code
@Component
public class MyComponent {
    // Component logic
}
```

3. What is Dependency Injection?

Answer:

Dependency Injection (DI) is a design pattern used to manage dependencies between objects. It allows you to inject dependencies into a class, rather than the class creating its own dependencies. This promotes loose coupling and easier testing.

Example:

```
java
Copy code
@Service
public class MyService {
    private final MyRepository myRepository;

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

}

4. How many ways can we perform dependency injection in Spring or Spring Boot?

Answer:

Dependency Injection can be performed in several ways:

Constructor Injection: Dependencies are provided through the class constructor.

Setter Injection: Dependencies are provided through setter methods.

Field Injection: Dependencies are injected directly into fields using @Autowired (not recommended due to lack of immutability).

Example (Constructor Injection):

java

Copy code

@Autowired

public MyService(MyRepository myRepository) {

  this.myRepository = myRepository;

}

Example (Setter Injection):

java

Copy code

@Autowired

public void setMyRepository(MyRepository myRepository) {

  this.myRepository = myRepository;

}

5. Where would you choose to use setter injection over constructor injection, and vice versa?

Answer:

Constructor Injection: Preferred for mandatory dependencies and ensures the object is fully initialized. It's useful for immutable objects and in cases where dependencies are required for the object's correct operation.

Setter Injection: Useful for optional dependencies or when you need to change dependencies after object creation. It’s less suitable for mandatory dependencies since it allows for partial initialization.

Example (Constructor Injection):

java

Copy code

```
public class MyService {
    private final MyRepository myRepository;

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

Example (Setter Injection):

java

Copy code

```
public class MyService {
    private MyRepository myRepository;

    @Autowired
    public void setMyRepository(MyRepository myRepository) {
```

```
    this.myRepository = myRepository;
  }
}
```

6. Can you provide an example of a real-world use case where @PostConstruct is particularly useful?

Answer:

@PostConstruct is used to execute initialization logic after the bean's properties have been set. It’s useful for actions that need to be performed after the bean is fully initialized.

Example:

java

Copy code

```
@Component
public class MyService {
  private final MyRepository myRepository;

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

  @PostConstruct
  public void init() {
    // Initialization logic, e.g., loading data into cache
    System.out.println("Service initialized.");
  }
```

}

7. How can we dynamically load values in a Spring Boot application?

Answer:

Dynamically loading values can be achieved using:

Profiles: Load different configurations based on the active profile.

Configuration Properties: Use @ConfigurationProperties to bind external configurations to Java objects.

Environment Variables: Inject environment variables using @Value.

Example (Profiles):

properties

Copy code

application-dev.properties

server.port=8081

application-prod.properties

server.port=8080

Example (Configuration Properties):

java

Copy code

@ConfigurationProperties(prefix = "my")

public class MyConfigProperties {

  private String name;

  private int age;

```
    // getters and setters
}
```

Example (Environment Variables):

java

Copy code

```
@Value("${MY_VARIABLE}")
private String myVariable;
```

8. Can you explain the key differences between YAML and properties files, and in what scenario you might prefer one format over the other?

Answer:

YAML: Offers a hierarchical, human-readable format with support for complex structures. Preferred for larger configurations due to its readability.

Properties: Simple key-value pairs with no hierarchical structure. Preferred for smaller, simpler configurations.

Example (YAML):

yaml

Copy code

```
server:
  port: 8081
```

Example (Properties):

properties

Copy code

```
server.port=8081
```

9. What is the difference between yml and yaml files?

Answer:

There is no functional difference between yml and yaml files; they are simply different file extensions for the same format. .yml is a shorthand form of .yaml.

Example:

application.yml

application.yaml

Both files are parsed the same way by Spring Boot.

10. If I configure the same values in both properties and YAML files, which value will be loaded in Spring Boot, or who will load first, properties or YAML file?

Answer:

Spring Boot prioritizes the order in which configuration files are loaded. By default, properties files have higher precedence over YAML files.

Order of Precedence:

Command-line arguments

Java system properties

application.properties (in src/main/resources)

application.yml (in src/main/resources)

Profile-specific properties and YAML files

Example:

If the same property is defined in both application.properties and application.yml, the value in application.properties will take precedence.

11. How to load external properties in Spring Boot?

Answer:

You can load external properties by specifying the location in the spring.config.location environment variable or as a command-line argument.

Example (Command-Line Argument):

bash

Copy code

java -jar myapp.jar --spring.config.location=file:/path/to/config/

Example (Environment Variable):

bash

Copy code

export SPRING_CONFIG_LOCATION=file:/path/to/config/

12. How to map or bind config properties to a Java object?

Answer:

You can map or bind configuration properties to a Java object using the @ConfigurationProperties annotation.

Example:

java

Copy code

```
@ConfigurationProperties(prefix = "my")
public class MyProperties {
   private String name;
```

```
    private int age;

    // getters and setters
}
```

In application.yml:

```
yaml
Copy code
my:
  name: John Doe
  age: 30
```

Registering the Configuration Properties:

```
java
Copy code
@Configuration
@EnableConfigurationProperties(MyProperties.class)
public class AppConfig {
}
```

Section-C

How will you resolve bean dependency ambiguity?

Answer:

Bean dependency ambiguity occurs when there are multiple beans of the same type and Spring doesn't know which one to inject. This can be resolved using:

@Qualifier: Specify the exact bean to inject.

Primary Bean: Use @Primary to mark a bean as the default.

Profile-specific Beans: Use Spring profiles to define beans for specific environments.

Example (Using @Qualifier):

java

Copy code

```
@Autowired
@Qualifier("specificBean")
private MyBean myBean;
```

Example (Using @Primary):

java

Copy code

```
@Bean
@Primary
public MyBean defaultBean() {
   return new MyBean();
}
```

Can we avoid this dependency ambiguity without using @Qualifier?

Answer:

Yes, dependency ambiguity can be avoided by:

Using @Primary to designate a default bean.

Defining bean names explicitly and using those names to resolve ambiguity.

Example (Using Bean Names):

java

Copy code

```
@Bean(name = "beanA")
public MyBean beanA() {
   return new MyBean();
}

@Bean(name = "beanB")
public MyBean beanB() {
   return new MyBean();
}
```

What is bean scope and can you explain different types of bean scope?

Answer:

Bean scope defines the lifecycle and visibility of a bean in the Spring container. Types of bean scopes include:

Singleton: One instance per Spring container. Default scope.

Prototype: A new instance for each request.

Request: One instance per HTTP request (Web context only).

Session: One instance per HTTP session (Web context only).

GlobalSession: One instance per global HTTP session (Web context only, less common).

Example (Defining Scope):

java

Copy code

```
@Component
@Scope("prototype")
public class MyBean {
```

}

How to define a custom bean scope?

Answer:

To define a custom bean scope, implement the Scope interface and register it with the Spring container.

Example (Custom Scope Implementation):

java

Copy code

```
public class CustomScope implements Scope {
   // Implementation details
}

@Configuration
public class AppConfig {
   @Bean
   public CustomScope customScope() {
      return new CustomScope();
   }
}
```

Can you provide a few real-time use cases for when to choose Singleton scope and Prototype scope?

Answer:

Singleton Scope:

Shared Resources: Use for beans representing shared resources, such as configuration settings or data sources.

Performance: Singleton beans are reused, which can improve performance by avoiding repeated creation.

Prototype Scope:

Stateful Beans: Use for beans that hold state or require individual configuration, like user sessions or temporary data processing.

Complex Objects: When each instance needs unique setup or initialization.

Can we inject a prototype bean into a singleton bean? If yes, what will happen if we inject a prototype bean into a singleton bean?

Answer:

Yes, you can inject a prototype bean into a singleton bean. However, the prototype bean will be instantiated only once, and its instance will be reused throughout the singleton bean's lifetime. To ensure a new prototype instance each time, use @Lookup or manually retrieve the prototype bean from the ApplicationContext.

Example (Using @Lookup):

java

Copy code

```
@Component
public class SingletonBean {
   @Lookup
   public PrototypeBean getPrototypeBean() {
      return null; // Spring will override this method
   }
}
```

What is the difference between Spring Singleton and plain Singleton?

Answer:

Spring Singleton: Ensures that only one instance of a bean is created within the Spring container. Managed by Spring and includes Spring-specific features such as dependency injection and lifecycle management.

Plain Singleton: A standard design pattern where only one instance of a class is created manually. It does not have dependency injection or lifecycle management.

Example (Spring Singleton):

java

Copy code

@Component

public class MyBean {

}

What is the purpose of the BeanPostProcessor interface in Spring, and how can you use it to customize bean initialization and destruction?

Answer:

BeanPostProcessor is used to perform custom processing on beans before and after their initialization. It allows you to modify bean properties or wrap beans with additional functionality.

Example (Using BeanPostProcessor):

java

Copy code

@Component

public class MyBeanPostProcessor implements BeanPostProcessor {

  @Override

```
    public Object postProcessBeforeInitialization(Object bean, String beanName) throws
BeansException {
        // Custom logic before initialization
        return bean;
    }

    @Override
    public Object postProcessAfterInitialization(Object bean, String beanName) throws
BeansException {
        // Custom logic after initialization
        return bean;
    }
}
```

Section-D

Have you worked on Restful web services? If yes, what all HTTP methods have you used in your project?

Answer:

Yes, I have worked on RESTful web services. The HTTP methods commonly used include:

GET: Retrieve data from the server.

POST: Create new resources.

PUT: Update existing resources.

DELETE: Remove resources.

PATCH: Partially update resources.

How can you specify the HTTP method type for your REST endpoint?

Answer:

You can specify the HTTP method type using annotations provided by Spring MVC, such as @GetMapping, @PostMapping, @PutMapping, @DeleteMapping, and @PatchMapping.

Example:

java

Copy code

```
@RestController
public class MyController {
  @GetMapping("/items")
  public List<Item> getItems() {
    // Handle GET request
  }

  @PostMapping("/items")
  public Item createItem(@RequestBody Item item) {
    // Handle POST request
  }
}
```

Can you design a REST endpoint, assuming you have a product database, and your task is to create an API to filter a list of products by productType? Design endpoints in a way that takes "productType" as input. If the user provides this input, the endpoint should filter products based on the specified condition. If productType is not provided, the endpoint should return all the products.

Answer:

Example:

java

Copy code

```
@RestController
@RequestMapping("/products")
public class ProductController {
  @Autowired
  private ProductService productService;

  @GetMapping
  public List<Product> getProducts(@RequestParam(value = "productType", required = false) String productType) {
    if (productType != null) {
      return productService.findByProductType(productType);
    } else {
      return productService.findAll();
    }
  }
}
```

What is the difference between @PathVariable and @RequestParam?

Answer:

@PathVariable: Used to extract values from URI templates. It is part of the URI path.

@RequestParam: Used to extract query parameters from the request URL. It is part of the query string.

Example:

java

Copy code

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

```
public Item getItemById(@PathVariable("id") Long id) {
   // Handle request
}

@GetMapping("/items")
public List<Item> getItems(@RequestParam("category") String category) {
   // Handle request
}
```

Why did you use @RestController and why not @Controller?

Answer:

@RestController: Used for RESTful web services. It combines @Controller and @ResponseBody, meaning that the return value of methods is written directly to the HTTP response body.

@Controller: Typically used for web applications where you return views (e.g., JSP, Thymeleaf) instead of data.

Example:

java

Copy code

```
@RestController
public class MyRestController {
   @GetMapping("/data")
   public Data getData() {
      // Return data as JSON or XML
   }
}
```

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

Answer:

You can use @RequestBody to automatically bind JSON payloads to Java objects.

Example:

java

Copy code

```
@PostMapping("/items")
public ResponseEntity<Item> createItem(@RequestBody Item item) {
   // Process the item
   return ResponseEntity.ok(item);
}
```

Can we perform update operation in POST HTTP method? If yes, then why do we need PUT mapping or PUT HTTP method?

Answer:

While you can use POST for updates, the POST method is generally intended for creating resources. PUT is used to update or replace a resource at a specific URI. It semantically differentiates between creation (POST) and update (PUT).

Example (Using PUT for Updates):

java

Copy code

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

```
public ResponseEntity<Item> updateItem(@PathVariable("id") Long id, @RequestBody
Item item) {
   // Update the item
   return ResponseEntity.ok(item);
}
```

Can we pass Request Body in GET HTTP method?

Answer:

Technically, the HTTP specification allows a body in GET requests, but it is not recommended and is not supported by all servers and clients. GET requests are intended to retrieve data without altering the state of the server.

How can we perform content negotiation (XML/JSON) in REST endpoint?

Answer:

Content negotiation allows clients to specify the format of the response (e.g., JSON or XML) using the Accept header. Spring automatically handles this based on the media type.

Example:

java

Copy code

```
@GetMapping("/items")
public ResponseEntity<List<Item>> getItems(@RequestHeader HttpHeaders headers) {
   List<Item> items = fetchItems();
   return ResponseEntity.ok().contentType(MediaType.APPLICATION_JSON).body(items);
}
```

What all status codes have you observed in your application?

Answer:

Common HTTP status codes include:

200 OK: Successful request.

201 Created: Resource created.

204 No Content: Successful request with no content.

400 Bad Request: Client error in request.

401 Unauthorized: Authentication required.

403 Forbidden: Access denied.

404 Not Found: Resource not found.

500 Internal Server Error: Server error.

How can you customize the status code for your endpoint?

Answer:

Customize status codes using ResponseEntity to set the desired status code.

Example:

java

Copy code

```
@PostMapping("/items")
public ResponseEntity<Item> createItem(@RequestBody Item item) {
   // Save item
   return ResponseEntity.status(HttpStatus.CREATED).body(item);
}
```

How can you enable Cross-Origin Resource Sharing (CORS)?

Answer:

You can enable CORS globally or at the controller level.

Example (Global CORS Configuration):

java

Copy code

```
@Configuration
public class WebConfig implements WebMvcConfigurer {
   @Override
   public void addCorsMappings(CorsRegistry registry) {
      registry.addMapping("/**").allowedOrigins("http://example.com");
   }
}
```

Example (Controller-Level CORS Configuration):

java

Copy code

```
@RestController
@RequestMapping("/items")
@CrossOrigin(origins = "http://example.com")
public class ItemController {
   // Controller methods
}
```

How can you upload a file in Spring?

Answer:

Use @RequestParam to handle file uploads.

Example:

java

Copy code

```
@PostMapping("/upload")
public ResponseEntity<String> uploadFile(@RequestParam("file") MultipartFile file) {
   // Process the file
   return ResponseEntity.ok("File uploaded successfully");
}
```

How do you maintain versioning for your REST API?

Answer:

API versioning can be maintained using URI versioning, request parameter versioning, or header versioning.

Example (URI Versioning):

java

Copy code

```
@GetMapping("/v1/items")
public List<Item> getItemsV1() {
   // Version 1 logic
}

@GetMapping("/v2/items")
public List<Item> getItemsV2() {
   // Version 2 logic
```

}

How will you document your REST API?

Answer:

Document REST APIs using tools like Swagger (OpenAPI) for generating interactive API documentation.

Example (Using SpringFox Swagger):

java

Copy code

```
@Configuration
@EnableSwagger2
public class SwaggerConfig {
  @Bean
  public Docket api() {
    return new Docket(DocumentationType.SWAGGER_2)
        .select()
        .apis(RequestHandlerSelectors.any())
        .paths(PathSelectors.any())
        .build();
  }
}
```

How can you hide certain REST endpoints to prevent them from being exposed externally?

Answer:

You can hide endpoints by using security configurations or by not exposing them via controllers.

Example (Using Security Configuration):

java

Copy code

```
@Configuration
@EnableWebSecurity
public class SecurityConfig extends WebSecurityConfigurerAdapter {
  @Override
  protected void configure(HttpSecurity http) throws Exception {
    http
      .authorizeRequests()
      .antMatchers("/hidden/**").denyAll()
      .anyRequest().permitAll();
  }
}
```

How will you consume a RESTful API?

Answer:

You can consume RESTful APIs using HTTP clients such as RestTemplate, WebClient, or external tools like Postman or curl.

Example (Using RestTemplate):

java

Copy code

```
@Service
public class ApiService {
```

```
    @Autowired
    private RestTemplate restTemplate;

    public Item getItem(Long id) {
        return restTemplate.getForObject("http://api.example.com/items/" + id, Item.class);
    }
}
```

Section-E

How will you handle exceptions in your project?

Answer:

Exceptions can be handled using:

Global Exception Handling: Use @ControllerAdvice and @ExceptionHandler to handle exceptions globally.

Custom Exception Classes: Define custom exceptions and handle them in a centralized manner.

HTTP Status Codes: Map exceptions to appropriate HTTP status codes.

Example (Using @ControllerAdvice):

java

Copy code

```
@ControllerAdvice
public class GlobalExceptionHandler {
    @ExceptionHandler(ResourceNotFoundException.class)
    public ResponseEntity<String>
handleResourceNotFound(ResourceNotFoundException ex) {
        return ResponseEntity.status(HttpStatus.NOT_FOUND).body(ex.getMessage());
```

```
    }

    @ExceptionHandler(Exception.class)
    public ResponseEntity<String> handleGenericException(Exception ex) {
        return ResponseEntity.status(HttpStatus.INTERNAL_SERVER_ERROR).body("An error occurred");
    }
}
```

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

Answer:

Exception Hierarchy: Define a base exception class and handle all derived exceptions in one handler.

Custom Exception Class: Use custom exceptions to centralize handling logic.

Exception Handler Methods: Use a single method to handle multiple exceptions if they require similar handling.

Example:

java

Copy code

```
@ControllerAdvice
public class GlobalExceptionHandler {
    @ExceptionHandler({ResourceNotFoundException.class, InvalidInputException.class})
    public ResponseEntity<String> handleMultipleExceptions(Exception ex) {
        return ResponseEntity.status(HttpStatus.BAD_REQUEST).body(ex.getMessage());
    }
```

}

How will you validate or sanitize your input payload?

Answer:

Validation Annotations: Use annotations like @NotNull, @Size, @Pattern on fields of DTOs.

Custom Validators: Implement custom validation logic for complex scenarios.

Sanitization: Clean or format input data to prevent security issues.

Example (Using Validation Annotations):

java

Copy code

```
public class UserDTO {
  @NotNull
  @Size(min = 2, max = 30)
  private String name;

  @Email
  private String email;
}
```

How can you populate validation error message to the end user?

Answer:

BindingResult: Use BindingResult to capture validation errors and return them to the user.

Exception Handling: Customize error messages in global exception handlers.

Example:

java

Copy code

```
@PostMapping("/users")
public ResponseEntity<?> createUser(@Valid @RequestBody UserDTO userDTO,
BindingResult bindingResult) {
  if (bindingResult.hasErrors()) {
    return ResponseEntity.badRequest().body(bindingResult.getAllErrors());
  }
  // Process userDTO
  return ResponseEntity.ok("User created successfully");
}
```

How can you define custom bean validation?

Answer:

Custom Validator: Implement ConstraintValidator interface and annotate your field with a custom constraint annotation.

Example (Custom Validator):

java

Copy code

```
@Target({ElementType.FIELD})
@Retention(RetentionPolicy.RUNTIME)
@Constraint(validatedBy = CustomValidator.class)
public @interface ValidCustom {
  String message() default "Invalid value";
  Class<?>[] groups() default {};
  Class<? extends Payload>[] payload() default {};
```

```
}

public class CustomValidator implements ConstraintValidator<ValidCustom, String> {
  @Override
  public boolean isValid(String value, ConstraintValidatorContext context) {
    // Custom validation logic
    return value != null && value.matches("^[a-zA-Z0-9]*$");
  }
}
```

Use case: Let's say you find a bug in the production environment, and now you want to debug the scenario. How can you do that from your local?

Answer:

Remote Debugging: Attach a debugger to the production environment if allowed.

Logs: Use extensive logging to track issues.

Replicate Environment: Reproduce the issue in a local or staging environment with similar configurations.

How can you enable a specific environment without using profiles? OR what is the alternative to profiles to achieve the same use case?

Answer:

Configuration Properties: Use environment-specific properties files (e.g., application-dev.properties).

Programmatic Configuration: Load configurations programmatically based on conditions.

Example (Using Environment-Specific Properties):

properties

Copy code

```
# application-dev.properties
server.port=8081
```

Example (Programmatic Configuration):

java

Copy code

```
@Configuration
public class AppConfig {
   @Bean
   public DataSource dataSource() {
      if (/* condition for dev environment */) {
         return new DriverManagerDataSource("jdbc:h2:mem:dev");
      }
      return new DriverManagerDataSource("jdbc:mysql://prod-db");
   }
}
```

What is the difference between @Profile and @Conditional?

Answer:

@Profile: Used to conditionally load beans based on the active profile.

@Conditional: Provides more granular control by allowing you to specify complex conditions for bean creation.

Example (Using @Profile):

java

Copy code

```
@Profile("dev")
@Bean
public DataSource devDataSource() {
   return new DriverManagerDataSource("jdbc:h2:mem:dev");
}
```

Example (Using @Conditional):

java

Copy code

```
@Conditional(OnPropertyCondition.class)
@Bean
public DataSource conditionalDataSource() {
   return new DriverManagerDataSource("jdbc:mysql://conditional-db");
}
```

What is AOP?

Answer:

Aspect-Oriented Programming (AOP) is a programming paradigm that provides a way to modularize cross-cutting concerns such as logging, security, and transactions, which are not easily handled by traditional OOP.

What is pointcut & join points in AOP?

Answer:

Pointcut: An expression that defines a set of join points where advice should be applied.

Join Point: A point during the execution of a program, such as method execution or object instantiation.

Example (Pointcut Expression):

java

Copy code

```
@Pointcut("execution(* com.example.service.*.*(..))")
public void serviceMethods() {}
```

What are different types of advice?

Answer:

Before Advice: Executed before a join point.

After Advice: Executed after a join point, regardless of its outcome.

After Returning Advice: Executed after a join point successfully completes.

After Throwing Advice: Executed if a join point throws an exception.

Around Advice: Wraps a join point and can modify its execution.

Example (Before Advice):

java

Copy code

```
@Before("serviceMethods()")
public void beforeAdvice(JoinPoint joinPoint) {
   // Logic to execute before method
}
```

Use case: Can I use AOP to evaluate the performance of a method, or is it possible to design a logging framework to capture request and response body of a method?

Answer:

Performance Evaluation: Yes, you can use AOP to measure the execution time of methods.

Logging Framework: Yes, AOP can be used to log request and response bodies.

Example (Performance Evaluation):

java

Copy code

```
@Around("serviceMethods()")
public Object aroundAdvice(ProceedingJoinPoint joinPoint) throws Throwable {
   long start = System.currentTimeMillis();
   Object result = joinPoint.proceed();
   long elapsedTime = System.currentTimeMillis() - start;
   System.out.println("Method executed in " + elapsedTime + " ms");
   return result;
}
```

Example (Logging Request and Response):

java

Copy code

```
@Around("serviceMethods()")
public Object logAround(ProceedingJoinPoint joinPoint) throws Throwable {
   System.out.println("Request: " + Arrays.toString(joinPoint.getArgs()));
   Object result = joinPoint.proceed();
   System.out.println("Response: " + result);
   return result;
}
```

Section-F

How does your application interact with the database and which framework are you using?

Answer:

The application interacts with the database using frameworks like Spring Data JPA, Hibernate, or MyBatis. These frameworks abstract the database interactions and provide a way to perform CRUD operations.

Example (Spring Data JPA):

java

Copy code

@Repository

public interface UserRepository extends JpaRepository<User, Long> {

}

Why is it important to configure a physical naming strategy?

Answer:

Configuring a physical naming strategy ensures that table and column names in the database are consistent and follow a naming convention. This is crucial for maintaining database schemas across different environments.

Example (Configuring Naming Strategy):

properties

Copy code

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

What are the key benefits of using Spring Data JPA?

Answer:

Simplified Data Access: Reduces boilerplate code for data access layers.

Repository Abstraction: Provides JpaRepository with common data access methods.

Query Methods: Supports creating queries from method names.

Pagination and Sorting: Built-in support for pagination and sorting.

What are the differences between Hibernate JPA and Spring Data JPA?

Answer:

Hibernate JPA: An implementation of the JPA specification that provides data access functionality and object-relational mapping.

Spring Data JPA: An abstraction layer over JPA and Hibernate that simplifies data access and repository management.

How can you connect multiple databases or data sources in a single application?

Answer:

Multiple DataSource Beans: Define multiple DataSource beans and configure them using @Primary or custom qualifiers.

Multiple EntityManagerFactory Beans: Define multiple EntityManagerFactory beans for each data source.

Example (Multiple DataSources):

java

Copy code

```
@Configuration
public class DataSourceConfig {
    @Bean
    @Primary
```

```
    public DataSource dataSourceOne() {
        return new DriverManagerDataSource("jdbc:mysql://db1");
    }

    @Bean
    public DataSource dataSourceTwo() {
        return new DriverManagerDataSource("jdbc:mysql://db2");
    }
}
```

What are the different ways to define custom queries in Spring Data JPA?

Answer:

Query Methods: Define queries using method names.

JPQL Queries: Use the @Query annotation to write JPQL queries.

Native Queries: Use the @Query annotation with nativeQuery = true for SQL queries.

Example (Using @Query):

java

Copy code

```
@Query("SELECT u FROM User u WHERE u.name = :name")
User findByName(@Param("name") String name);
```

How will you define entity relationships or association mapping in Spring Data JPA?

Answer:

One-to-One: @OneToOne

One-to-Many: @OneToMany

Many-to-One: @ManyToOne

Many-to-Many: @ManyToMany

Example (One-to-Many):

java

Copy code

```
@Entity
public class User {
   @OneToMany(mappedBy = "user")
   private List<Order> orders;
}
```

Is it possible to execute join query in Spring Data JPA? If yes, how can you add some insights?

Answer:

Yes, you can execute join queries using JPQL or native queries.

Example (JPQL Join Query):

java

Copy code

```
@Query("SELECT u FROM User u JOIN u.orders o WHERE o.status = :status")
List<User> findUsersByOrderStatus(@Param("status") String status);
```

How will you implement pagination and sorting in Spring Data JPA?

Answer:

Use Pageable and Sort to handle pagination and sorting in repository methods.

Example:

java

Copy code

```
@Query("SELECT u FROM User u")
Page<User> findAllUsers(Pageable pageable);

@Query("SELECT u FROM User u ORDER BY u.name")
List<User> findAllUsersSorted(Sort sort);
```