1. Basics of Spring Boot

1. What is Spring Boot? Why use it?

Spring Boot is an opinionated framework that simplifies the creation of production-ready, stand-alone Spring applications. It takes a convention-over-configuration approach to reduce boilerplate code and streamline development.

We use it because it enables us to:

- **Quickly bootstrap** a new Spring project with minimal configuration.
- **Create standalone applications** with embedded servers like Tomcat or Jetty, eliminating the need for a separate WAR file deployment.
- **Reduce boilerplate configuration** through its auto-configuration feature.
- Provide a set of **starter dependencies** that manage versioning and provide a single point of dependency management for common functionalities.

2. Difference between Spring and Spring Boot?

- **Spring Framework** provides a broad set of features for building enterprise Java applications. It is a large, flexible, and powerful framework, but it requires significant manual configuration (e.g., XML or Java config) to set up and configure various components.
- **Spring Boot** is an extension of the Spring Framework. It's designed to simplify the development process by providing an opinionated approach, auto-configuration, and standalone capabilities. It essentially builds on top of Spring to make it easier to use.

3. What are the advantages of Spring Boot?

- **Rapid Application Development:** Get a project up and running in minutes.
- **Embedded Servers:** No need to configure a web server separately.
- **Auto-Configuration:** Automatically configures your application based on the dependencies present on the classpath.
- **Starter Dependencies:** Simplify dependency management by providing pre-configured dependencies for common tasks.
- **Stand-alone Applications:** Create executable JAR files that can be run directly from the command line.
- **Production-Ready Features:** Includes features like Actuator for monitoring and health checks.

4. What is the Spring Boot starter dependency?

Spring Boot starter dependencies are a set of convenient dependency descriptors that you can include in your build. They are a one-stop shop for all the Spring and related technologies you need to get a specific feature up and running. For example, the `spring-boot-starter-web` starter includes dependencies for Spring MVC, REST, Tomcat, and more. This saves you from manually adding and managing each dependency's version.

5. What is the role of `spring-boot-starter-parent`?

The spring-boot-starter-parent is a special Maven POM (Project Object Model) that provides a solid foundation for your Spring Boot projects. It defines a set of conventions and defaults, including:

- **Dependency Management:** It manages the versions of all Spring and common third-party libraries, ensuring compatibility.
- **Default Plugin Configuration:** It pre-configures common plugins like maven-surefire-plugin for testing.
- **Sensible Defaults:** It provides default configurations for things like resource filtering and encoding.

By inheriting from it, you get a consistent build configuration without having to manually define everything.

6. What is the difference between Spring Boot and Spring MVC?

- **Spring MVC** is a module within the broader Spring Framework that is used for building web applications (Model-View-Controller architecture). It focuses specifically on handling HTTP requests and responses.
- **Spring Boot** is a framework that simplifies the entire Spring ecosystem, including Spring MVC. When you use spring-boot-starter-web, Spring Boot automatically configures Spring MVC, an embedded server, and other necessary components for you.

7. What is the difference between Spring Boot and Spring Cloud?

- **Spring Boot** is a framework for building individual, stand-alone, production-ready Spring applications. It focuses on simplifying the development of a single application.
- **Spring Cloud** is a collection of tools and frameworks that build on top of Spring Boot to solve common challenges in distributed systems and microservices. It provides features like service discovery, load balancing, circuit breakers, and distributed configuration.

Think of it this way: Spring Boot helps you build the individual "microservices," and Spring Cloud helps you manage and connect them in a distributed environment.

8. How does Spring Boot reduce boilerplate configuration?

Spring Boot achieves this primarily through **Auto-configuration**. When your application starts, Spring Boot inspects the classpath and the beans you have defined. Based on what it finds, it automatically configures the necessary beans and settings. For example, if it detects spring-boot-starter-web and h2 database, it will automatically configure a DataSource and a DispatcherServlet for you.

9. What are properties and YAML configuration files in Spring Boot?

Both **.properties** and **.yml** (YAML Ain't Markup Language) files are used to define configuration for a Spring Boot application. They allow you to externalize settings like database connection URLs, server ports, and custom application properties.

- **application.properties:** A key-value pair file. Example: server.port=8080.
- **application.yml:** A more structured, hierarchical format. It is often preferred for its readability and ability to organize complex configurations. Example:
- YAML

```
server:
  port: 8080
```

-
-

10. How to define custom properties in Spring Boot?

You can define custom properties in application.properties or application.yml. For example, in application.properties:

```
Properties

my.custom.greeting=Hello from Spring Boot!
```

You can then inject this value into your application using the @Value annotation.

```
Java

@Component
public class MyComponent {
    @Value("${my.custom.greeting}")
    private String greeting;
    // ...
}
```

---

2. Spring Boot Annotations

11. What is @SpringBootApplication annotation?

The @SpringBootApplication annotation is a convenience annotation that combines three key annotations:[1]

1. @Configuration: Tags the class as a source of bean definitions for the application context.[2]
2. @EnableAutoConfiguration: Tells Spring Boot to start adding b[3]eans based on classpath settings, other beans, and various property settings.
3. @ComponentScan: Scans the package where the application class is located (and its sub-packages) for components.

It's the main entry point for most Spring Boot applications.

12. Difference between @Component, @Service, @Repository, and @Controller?

These are all stereotypes for Spring components, meaning they are special types of @Component that have a specific role in a typical three-tier architecture.

- **@Component:** The most general-purpose stereotype. It marks a class as a Spring-managed component.
- **@Service:** Marks a class in the **service layer** (business logic). It's a specialization of @Component.
- **@Repository:** Marks a class in the **data access layer** (persistence logic). It also provides a feature called **exception translation**, which converts specific data access exceptions into a Spring-specific DataAccessException.
- **@Controller:** Marks a class in the **web layer** (presentation logic). It is used to handle incoming web requests and return a view.

13. Difference between @RestController and @Controller?

- **@Controller:** Is used in the Spring MVC framework to handle web requests and typically returns a view (e.g., HTML page name).
- **@RestController:** Is a specialized controller used for building RESTful web services. It's a combination of @Controller and @ResponseBody. This means that every method in a @RestController class automatically serializes its return value into a JSON or XML format and writes it directly to the HTTP response body, instead of returning a view name.

14. What is the use of @Autowired?

@Autowired is a mechanism for **dependency injection**. It allows Spring to automatically inject the required beans into your classes. Spring's IoC container will look for a bean that matches the type of the field, constructor, or setter method and inject it automatically.

15. What is the difference between field injection, setter injection, and constructor injection?

These are three ways to use @Autowired for dependency injection.

- **Field Injection:** The dependency is injected directly into a field. It's the simplest but has some drawbacks, like making the class harder to test without a Spring container.
- Java

```
@Autowired
private MyService myService;
```

-
-
- **Setter Injection:** The dependency is injected via a setter method. It's good for optional dependencies.

- Java

```
private MyService myService;
@Autowired
public void setMyService(MyService myService) {
    this.myService = myService;
}
```

-
-
- **Constructor Injection (Recommended):** The dependency is injected through the class constructor. This is the **preferred approach** as it ensures that the object is in a valid state from the moment it is created (dependencies are not null) and makes the class easier to test.
- Java

```
private final MyService myService;
public MyController(MyService myService) {
    this.myService = myService;
}
```

-
-

16. What is the use of `@Value` annotation?

The `@Value` annotation is used to inject values from properties files (`.properties`, `.yml`) or from environment variables into your Spring components. It's often used to inject simple string values, numbers, or boolean flags.

17. What is `@Configuration` and `@Bean`?

- **`@Configuration`:** Annotates a class to signal that it's a source of bean definitions. It's like a Java-based replacement for XML configuration files.
- **`@Bean`:** Annotates a method within a `@Configuration` class. The method's return value is registered as a bean in the Spring ApplicationContext. The name of the bean is the same as the method name.

Example:

Java

```
@Configuration
public class AppConfig {
    @Bean
    public MyService myService() {
        return new MyServiceImpl();
    }
}
```

18. What is @Qualifier?

When Spring's dependency injection finds **multiple beans of the same type**, it doesn't know which one to inject. The @Qualifier annotation is used to resolve this ambiguity by specifying the exact bean to be injected. You provide the name of the desired bean as the value of the qualifier.

Example:

Java

```
@Service("fileStorage")
public class FileStorageService implements StorageService { ... }

@Service("databaseStorage")
public class DatabaseStorageService implements StorageService { ... }

// In another component
@Autowired
@Qualifier("fileStorage")
private StorageService storageService;
```

19. What is @Primary?

The @Primary annotation is an alternative to @Qualifier for resolving ambiguity. When multiple beans of the same type are available, you can mark one of them with @Primary. Spring will then automatically inject the @Primary bean by default, unless a specific @Qualifier is used.

Example:

Java

```
@Service
@Primary
public class FileStorageService implements StorageService { ... }

@Service
public class DatabaseStorageService implements StorageService { ... }

// Spring will automatically inject FileStorageService here
@Autowired
private StorageService storageService;
```

20. What is @Lazy in Spring Boot?

By default, Spring beans are eagerly initialized, meaning they are created at the application startup. The `@Lazy` annotation can be used to indicate that a bean should be **lazily initialized**. This means the bean will not be created until it is actually needed and injected into another component. This can speed up application startup time, but may cause performance delays when the bean is first requested.

---

3. Dependency Injection & Beans

21. What is Inversion of Control (IoC) in Spring Boot?

Inversion of Control (IoC) is a design principle where a framework or container takes control of the flow of a program, rather than the developer. Instead of the developer manually creating and managing objects, the **Spring IoC container** manages the creation, configuration, and lifecycle of these objects. A key aspect of IoC is **Dependency Injection**, where the container "injects" the dependencies into the objects that need them.

22. What are Spring Beans?

Spring Beans are objects that are instantiated, assembled, and managed by the Spring IoC container. They are the fundamental building blocks of a Spring application. Any class you annotate with `@Component`, `@Service`, `@Repository`, or `@Controller` (or define in a `@Configuration` class with `@Bean`) becomes a Spring Bean.

23. What is the Spring ApplicationContext?

The **`ApplicationContext`** is the central interface in a Spring application. It represents the IoC container. It provides a means to access beans, manage their lifecycles, load resources, publish events, and more. It extends `BeanFactory` and provides more enterprise-specific functionality.

24. Difference between `BeanFactory` and `ApplicationContext`?

- **`BeanFactory`**: The basic IoC container. It is a simple, lightweight container that loads bean definitions and provides access to them. It **lazily initializes** beans.
- **`ApplicationContext`**: A more advanced container. It extends `BeanFactory` and adds more features, such as internationalization, event propagation, and resource loading. It **eagerly initializes** singleton beans by default at application startup. For most modern Spring applications, `ApplicationContext` is the preferred choice.

25. What are singleton and prototype beans?

These are the two most common bean scopes in Spring.

- **Singleton (Default):** A single shared instance of the bean is created and managed by the container. All requests for this bean will receive the **same instance**.
- **Prototype:** A new instance of the bean is created for **every request**. This is useful for stateful beans where each user needs their own unique instance.

You can specify the scope using the `@Scope` annotation.

26. How to create custom beans in Spring Boot?

You can create a custom bean using a `@Configuration` class and the `@Bean` annotation.

```
Java

@Configuration
public class MyConfig {
    @Bean
    public CustomBean customBean() {
        return new CustomBean();
    }
}
```

This method will create an instance of `CustomBean` and register it in the Spring context.

27. What is the bean lifecycle in Spring?

The lifecycle of a Spring bean involves several steps:

1. **Instantiation:** The container creates the bean instance.
2. **Population of properties:** The container injects the dependencies into the bean.
3. **Aware interfaces:** The container calls various "aware" interfaces (e.g., `BeanNameAware`).
4. **`BeanPostProcessor` pre-initialization:** The `postProcessBeforeInitialization` method of any `BeanPostProcessor` is called.
5. **Initialization:** The container calls the bean's custom initialization methods (e.g., `@PostConstruct`).
6. **`BeanPostProcessor` post-initialization:** The `postProcessAfterInitialization` method is called.
7. **Ready to use:** The bean is now ready for use.
8. **Destroy:** When the container shuts down, it calls the bean's destruction methods (e.g., `@PreDestroy`).

28. How to use `@PostConstruct` and `@PreDestroy`?

These annotations are part of the JSR-250 specification and are used to mark methods that should run during a bean's lifecycle events.

- **`@PostConstruct`:** The method annotated with `@PostConstruct` is executed **after** the bean has been created and its dependencies have been injected. It's used for initialization tasks.
- **`@PreDestroy`:** The method annotated with `@PreDestroy` is executed **before** the bean is destroyed by the container. It's used for cleanup tasks.

Example:

```
Java

@Component
```

```
public class MyService {
    @PostConstruct
    public void init() {
        System.out.println("Service initialized!");
    }

    @PreDestroy
    public void destroy() {
        System.out.println("Service is shutting down!");
    }
}
```

29. What is the difference between eager and lazy initialization?

- **Eager Initialization (Default):** Beans are created and initialized by the container as soon as the application starts. This can lead to a longer startup time but ensures that all dependencies are met and the application is ready to handle requests immediately.
- **Lazy Initialization:** Beans are not created until they are actually needed and referenced by another bean. This can speed up startup time but may introduce a slight delay the first time a bean is used.

You can enable lazy initialization on a bean using the `@Lazy` annotation.

30. How to override default bean properties in Spring Boot?

You can override default Spring Boot properties in your `application.properties` or `application.yml` file. For example, to change the default server port from 8080 to 9090:

```
Properties

server.port=9090
```

This is a core feature of Spring Boot, allowing for easy externalization and modification of configuration.

---

4. Spring Boot REST APIs

31. How to create REST API in Spring Boot?

You create REST APIs in Spring Boot by using Spring MVC annotations.

1. Create a class and annotate it with **`@RestController`** to handle incoming HTTP requests.
2. Define methods within the class to handle specific HTTP methods and endpoints.
3. Annotate these methods with annotations like **`@GetMapping`**, **`@PostMapping`**, etc., to map them to URL paths.

4. Use annotations like `@PathVariable`, `@RequestParam`, and `@RequestBody` to handle request data.

Example:

```
Java

@RestController
@RequestMapping("/api/products")
public class ProductController {
    @GetMapping("/{id}")
    public Product getProductById(@PathVariable Long id) {
        // logic to retrieve product
        return new Product(id, "Sample Product");
    }
}
```

32. Difference between `@RequestParam` and `@PathVariable`?

- **`@RequestParam`:** Used to extract data from the **query parameters** in the URL. For example, `GET /api/products?id=123`. The ID would be a request parameter.
- Java

```
@GetMapping("/products")
public Product getProduct(@RequestParam Long id) { ... }
```

-
-
- **`@PathVariable`:** Used to extract data directly from the **path of the URL**. For example, `GET /api/products/123`. The `123` would be a path variable.
- Java

```
@GetMapping("/products/{id}")
public Product getProductById(@PathVariable Long id) { ... }
```

-
-

33. What is `@RequestBody` and `@ResponseBody`?

- **`@RequestBody`:** Binds the **HTTP request body** to a method parameter. Spring automatically converts the JSON or XML data from the request body into a Java object. This is typically used with `@PostMapping` or `@PutMapping` for creating or updating resources.
- **`@ResponseBody`:** Binds the **method return value** to the HTTP response body. Spring converts the Java object into JSON or XML and writes it to the response. This is implicitly included in the `@RestController` annotation.

34. Difference between `@GetMapping`, `@PostMapping`, `@PutMapping`, `@DeleteMapping`, and `@PatchMapping`?

These annotations are shortcuts for the more general `@RequestMapping` and map to specific HTTP methods.

- **`@GetMapping`:** Used for **retrieving** a resource. It's a read-only operation. (Idempotent)
- **`@PostMapping`:** Used for **creating** a new resource. (Not Idempotent)
- **`@PutMapping`:** Used for **updating/replacing** a complete resource. (Idempotent)
- **`@DeleteMapping`:** Used for **deleting** a resource. (Idempotent)
- **`@PatchMapping`:** Used for **partially updating** a resource. (Not Idempotent)

Idempotent means that a request can be repeated multiple times without causing different effects on the server.

35. How to handle exceptions in Spring Boot REST API?

You can handle exceptions using a couple of main approaches:

1. **Local `try-catch` blocks:** The most basic way, but can lead to code duplication.
2. **`@ExceptionHandler`:** Define a method in your controller to handle a specific exception for that controller.
3. **`@ControllerAdvice`:** The recommended approach. You create a separate class annotated with `@ControllerAdvice` to handle exceptions **globally** across all controllers in your application.

36. What is `@ControllerAdvice` and `@ExceptionHandler`?

- **`@ExceptionHandler`:** An annotation used to define a method that handles a specific exception type. This method can be placed inside a controller or, more commonly, inside a class with `@ControllerAdvice`.
- **`@ControllerAdvice`:** An annotation that allows you to define a class that provides **global exception handling** and other advice for all controllers. It centralizes exception handling logic, making your controllers cleaner.

37. What is `ResponseEntity` in Spring Boot?

`ResponseEntity` is a class that represents the entire **HTTP response**: the status code, headers, and body. It provides a flexible and explicit way to return a complete HTTP response from a controller method.

Example:

```
Java

@GetMapping("/{id}")
public ResponseEntity<Product> getProductById(@PathVariable Long id) {
    Product product = findProduct(id);
    if (product != null) {
```

```
        return new ResponseEntity<>(product, HttpStatus.OK);
    } else {
        return new ResponseEntity<>(HttpStatus.NOT_FOUND);
    }
}
```

38. What is the use of `HttpStatus` in Spring Boot REST?

`HttpStatus` is an enum that represents HTTP status codes (e.g., `200 OK`, `404 NOT FOUND`, `500 INTERNAL_SERVER_ERROR`). It's used with `ResponseEntity` to explicitly set the HTTP status code of the response, providing a clear contract for REST API clients.

39. How to implement validation in Spring Boot (`@Valid`, `@NotNull`, `@Size`)?

Spring Boot integrates with the Java Bean Validation API (JSR 380).

1. Add the `spring-boot-starter-validation` dependency.
2. Annotate your request DTO (Data Transfer Object) fields with validation annotations like `@NotNull`, `@Size`, `@Pattern`, `@Email`, etc.
3. Annotate the method parameter in your controller with `@Valid`. Spring will then automatically validate the object.

Example:

```
Java

// DTO
public class UserDto {
    @NotNull(message = "Name cannot be null")
    @Size(min = 2, max = 50, message = "Name must be between 2 and 50 characters")
    private String name;
}

// Controller
@PostMapping("/users")
public ResponseEntity<User> createUser(@Valid @RequestBody UserDto userDto) {
    // ...
}
```

40. What is HATEOAS in Spring Boot REST?

HATEOAS stands for **Hypermedia As The Engine Of Application State**. It's a principle of RESTful architecture where a resource representation includes hyperlinks to related resources. This allows a client to navigate through the API by following links, rather than having hardcoded URLs. Spring Boot provides Spring HATEOAS to simplify its implementation.

---

5. Spring Boot Data & JPA

41. What is Spring Data JPA?

Spring Data JPA is a framework that makes it easy to implement JPA-based data access layers. It takes a "convention over configuration" approach and dramatically reduces boilerplate code by automatically generating a repository implementation for you at runtime.

42. Difference between JPA, Hibernate, and Spring Data JPA?

- **JPA (Java Persistence API):** A **specification** for object-relational mapping (ORM) in Java. It defines a set of APIs and a standard for interacting with databases, but it doesn't provide an implementation.
- **Hibernate:** A popular **implementation** of the JPA specification. It provides the actual ORM framework to map Java objects to database tables.
- **Spring Data JPA:** An **abstraction** built on top of JPA. It simplifies the development of JPA-based applications by providing a high-level API for creating data repositories without writing boilerplate `EntityManager` code.

43. What is `CrudRepository` and `JpaRepository`?

These are interfaces provided by Spring Data JPA.

- **`CrudRepository`:** Provides basic CRUD (Create, Read, Update, Delete) operations. It includes methods like `save()`, `findById()`, `findAll()`, and `delete()`.
- **`JpaRepository`:** Extends `PagingAndSortingRepository` which in turn extends `CrudRepository`. It provides additional JPA-specific features, such as `flush()` and `saveAndFlush()`, as well as methods for sorting and pagination.

For most applications, `JpaRepository` is the preferred choice as it provides a richer set of features.

44. Difference between `save()` and `saveAndFlush()`?

- **`save()`:** Persists the entity to the database but does not immediately write the changes to the database. The changes are held in a **transactional cache** and will only be flushed when the transaction commits.
- **`saveAndFlush()`:** Persists the entity to the database **and immediately flushes the changes** to the database, effectively writing them to the disk. This is useful when you need to ensure the data is in the database immediately, for example, before a subsequent query.

45. How to define custom queries in Spring Data JPA?

You can define custom queries in a few ways:

1. **Query Methods:** Spring Data JPA can derive queries from the method name. Example: `findByLastNameAndFirstName(String lastName, String firstName)`.
2. **`@Query` Annotation:** Use the `@Query` annotation to write custom JPQL or native SQL queries directly on the method.

    - **JPQL:** `SELECT p FROM Product p WHERE p.price > :minPrice`
    - **Native SQL:** `SELECT * FROM products WHERE price > :minPrice` (requires `nativeQuery = true`)

46. What is the difference between JPQL and native SQL queries?

- **JPQL (Java Persistence Query Language):** An object-oriented query language used with JPA. It operates on entities and their relationships, not on database tables. This makes your code more portable across different databases.
- **Native SQL:** The actual SQL syntax of your specific database (e.g., MySQL, PostgreSQL). It gives you full control but makes your code less portable.

47. What is the difference between eager and lazy loading?

This refers to how related entities are loaded from the database.

- **Eager Loading:** The related entities are loaded from the database **immediately** along with the main entity. This can lead to performance issues if you load many related entities that aren't needed. (Uses `@ManyToOne` or `@OneToOne`)
- **Lazy Loading (Default for collections):** The related entities are **not loaded until they are actually accessed**. This can be more efficient, as you only fetch what you need. A potential issue is the N+1 problem. (Uses `@ManyToMany` or `@OneToMany`)

48. What is the N+1 problem in Hibernate and how to solve it?

The **N+1 problem** occurs with lazy loading. If you have a list of `N` parent entities and you iterate through them, accessing a lazy-loaded child collection for each one, Hibernate will execute `N` separate queries for the children in addition to the initial query for the parents. This results in `N+1` queries, leading to poor performance.

You can solve it using:

- **Fetch Joins:** Use a `JOIN FETCH` clause in a JPQL query to load the parent and child entities in a single query.
- **`@EntityGraph`:** A more modern approach in JPA to specify which associations should be fetched eagerly.
- **Batch fetching:** Configure Hibernate to fetch a batch of child entities at once.

49. What is optimistic vs pessimistic locking in JPA?

These are strategies for handling concurrent data access.

- **Optimistic Locking:** Assumes that conflicts are rare. It uses a version number or a timestamp on the entity. When an update is attempted, the version number is checked. If it's different, another user has modified the data, and an exception is thrown. **No physical lock is held.**
- **Pessimistic Locking:** Assumes that conflicts are frequent. It places a **physical lock** on the database row to prevent other transactions from reading or updating the data until the current transaction is complete.

50. How to handle transactions in Spring Boot (`@Transactional`)?

The `@Transactional` annotation is used to manage transactions declaratively in Spring.

- When a method annotated with `@Transactional` is called, Spring opens a transaction.
- If the method completes successfully, Spring commits the transaction.
- If an unhandled exception occurs, Spring rolls back the transaction.

This ensures that a group of database operations is treated as a single, atomic unit of work.

---

6. Spring Boot Security

51. What is Spring Security?

Spring Security is a powerful and highly customizable authentication and access-control framework for securing Spring applications. It provides comprehensive security services for web applications, including support for authentication, authorization, CSRF protection, and more.

52. How to secure REST APIs in Spring Boot?

1. Add the `spring-boot-starter-security` dependency.
2. Spring Boot automatically provides basic security features. By default, it will secure all endpoints with basic HTTP authentication and generate a random password.
3. Customize the security configuration by creating a class that extends `WebSecurityConfigurerAdapter` (in older versions) or provides a `SecurityFilterChain` bean.
4. Define security rules using a `http.authorizeRequests()` block to specify which endpoints are public (`permitAll()`) and which require authentication (`authenticated()`) or specific roles (`hasRole()`).

53. Difference between Authentication and Authorization?

- **Authentication:** The process of **verifying the identity** of a user. It answers the question, "Who is this user?" (e.g., username and password, token).
- **Authorization:** The process of **determining if an authenticated user has the right to access a resource**. It answers the question, "What is this user allowed to do?" (e.g., is a user allowed to view a specific page, or delete a record?).

54. What is OAuth2 in Spring Security?

OAuth2 is an authorization framework that enables an application to obtain limited access to a user's account on an HTTP service. Spring Security provides a comprehensive module for implementing both an OAuth2 client (to consume a service) and an OAuth2 server (to provide a service).

55. What is JWT (JSON Web Token) authentication in Spring Boot?

JWT is a compact, URL-safe means of representing claims to be transferred between two parties. In authentication, the server generates a JWT after a user authenticates. The client then stores this token and sends it with every subsequent request. The server validates the token to ensure the user is authenticated without needing to re-authenticate with credentials. Spring Security can be configured to use JWT-based authentication.

56. How to implement role-based access control in Spring Boot?

You can implement role-based access control (RBAC) by using `hasRole()` or `hasAnyRole()` in your security configuration.

Example:

```
Java

@Bean
public SecurityFilterChain securityFilterChain(HttpSecurity http) throws Exception {
    http
        .authorizeHttpRequests(authorize -> authorize
            .requestMatchers("/api/admin/**").hasRole("ADMIN")
            .requestMatchers("/api/users/**").hasAnyRole("USER", "ADMIN")
            .anyRequest().authenticated()
        );
    return http.build();
}
```

57. What is CSRF protection in Spring Security?

CSRF (Cross-Site Request Forgery) is a type of attack where a malicious website tricks a user's browser into making an unwanted request to a trusted site where the user is authenticated. Spring Security provides built-in CSRF protection by default for POST, PUT, and DELETE requests. It works by generating a unique token that must be included in the request, which a malicious site cannot access.

58. How to implement custom user authentication in Spring Boot?

You can implement custom authentication by providing a bean of `UserDetailsService`. This service is responsible for loading user-specific data, such as a username, password, and roles.

Example:

```
Java

@Service
public class CustomUserDetailsService implements UserDetailsService {
    @Override
    public UserDetails loadUserByUsername(String username) {
        // Find user from database
        if ("user".equals(username)) {
```

```
            return new User("user", "{noop}password", List.of(new
SimpleGrantedAuthority("ROLE_USER")));
        }
        throw new UsernameNotFoundException("User not found");
    }
}
```

59. What is the difference between basic auth and bearer token?

- **Basic Authentication:** The client sends the user's username and password, encoded in Base64, in the `Authorization` header of every request. It's simple but insecure if not used with HTTPS.
- **Bearer Token:** The client sends a unique token (like a JWT or OAuth2 token) in the `Authorization` header, prefixed with `Bearer`. The server then validates the token. It's more secure as the user's credentials are not sent with every request.

60. How to disable security for specific endpoints in Spring Boot?

You can disable security for certain endpoints by using the `permitAll()` method in your security configuration.

Example:

Java

```
@Bean
public SecurityFilterChain securityFilterChain(HttpSecurity http) throws Exception {
    http
        .csrf(csrf -> csrf.disable()) // Only for demo
        .authorizeHttpRequests(authorize -> authorize
            .requestMatchers("/public/**").permitAll()
            .anyRequest().authenticated()
        );
    return http.build();
}
```

---

7. Spring Boot Actuator & Monitoring

61. What is Spring Boot Actuator?

Spring Boot Actuator is a module that provides production-ready features to help you monitor and manage your application. It provides built-in endpoints to expose internal information like application health, metrics, environment properties, and more.

62. What are common Actuator endpoints?

Common endpoints include:

- **/health:** Shows application health information (e.g., database connectivity).
- **/info:** Displays arbitrary application information.
- **/metrics:** Provides detailed metrics about the application (e.g., JVM memory usage, HTTP request counts).
- **/env:** Shows the application's environment properties.
- **/beans:** Lists all the Spring beans in the application.

63. How to secure Actuator endpoints?

By default, all actuator endpoints (except /health and /info) are secured. You can configure their security by adding a SecurityFilterChain bean.

Example:

```
Java

@Bean
public SecurityFilterChain actuatorSecurity(HttpSecurity http) throws Exception {
    http.securityMatcher(EndpointRequest.toAnyEndpoint()).authorizeHttpRequests(requests ->
requests.anyRequest().hasRole("ACTUATOR"));
    return http.build();
}
```

64. What is Micrometer in Spring Boot?

Micrometer is a vendor-neutral application metrics facade. It provides a simple API to instrument your application and then ships those metrics to various monitoring systems like Prometheus, Datadog, or Grafana. Spring Boot Actuator uses Micrometer by default to provide its metrics.

65. How to integrate Spring Boot with Prometheus & Grafana?

1. Add the micrometer-registry-prometheus dependency.
2. Spring Boot Actuator automatically exposes a Prometheus endpoint at /actuator/prometheus.
3. Configure Prometheus to scrape this endpoint.
4. Configure Grafana to use Prometheus as a data source to create dashboards and visualize the metrics.

66. What is Health Check API in Spring Boot?

The **Health Check API**, exposed at the /actuator/health endpoint, provides a summary of the application's health status. It can check the status of various components, such as database connectivity, disk space, and custom health indicators.

67. How to customize Actuator metrics?

You can create custom metrics by injecting a MeterRegistry bean and using it to record values.

Example:

```
Java

@Component
public class MyService {
    private final Counter myCounter;
    public MyService(MeterRegistry meterRegistry) {
        this.myCounter = meterRegistry.counter("my.custom.counter", "tag", "value");
    }
    public void doSomething() {
        myCounter.increment();
    }
}
```

68. What is log monitoring in Spring Boot?

Log monitoring is the process of collecting, aggregating, and analyzing log data from a running application. Spring Boot uses SLF4J (Simple Logging Facade for Java) by default, with Logback as the implementation. You can configure log levels and appenders to send logs to files or external systems like Elastic Stack (ELK).

69. What are distributed tracing tools (Zipkin, Sleuth)?

In a microservices architecture, a single user request can span multiple services. **Distributed tracing** is the process of tracking a request as it flows through these services.

- **Spring Cloud Sleuth:** An instrumentation library that adds unique IDs to headers in requests, allowing you to trace them across services.
- **Zipkin:** A distributed tracing system that collects, stores, and visualizes the trace data from Sleuth, helping you analyze latency and debug issues.

70. How to monitor Spring Boot applications in production?

1. **Use Actuator:** For health, metrics, and environment info.
2. **Externalize Logs:** Ship logs to a centralized logging system like ELK or Splunk.
3. **Use Metrics:** Collect metrics with Micrometer and visualize them in a tool like Prometheus/Grafana.
4. **Implement Tracing:** Use Spring Cloud Sleuth and Zipkin for distributed tracing in a microservices environment.
5. **Application Performance Monitoring (APM):** Use APM tools like New Relic, Dynatrace, or AppDynamics for more in-depth monitoring.

---

8. Spring Boot Testing

71. How to test Spring Boot applications?

Spring Boot provides excellent support for testing with the `spring-boot-starter-test` dependency. Key concepts include:

- **Unit Testing:** Testing individual components in isolation. Use mocking frameworks like Mockito.
- **Integration Testing:** Testing how different components work together, with the Spring context. Use annotations like `@SpringBootTest`.
- **Slice Testing:** Testing a specific layer of your application without loading the entire context (e.g., `@DataJpaTest`, `@WebMvcTest`).

72. Difference between unit testing and integration testing in Spring Boot?

- **Unit Testing:** Focuses on testing a **single class or method** in isolation, often by mocking its dependencies. It's fast and doesn't require a running application context.
- **Integration Testing:** Verifies the **interaction between multiple components**. It requires the Spring ApplicationContext to be loaded, making it slower but more comprehensive. It tests how a controller interacts with a service, a service with a repository, etc.

73. What is `@SpringBootTest`?

The `@SpringBootTest` annotation is the primary annotation for integration tests. It loads the entire Spring ApplicationContext, allowing you to test the full application flow. You can also configure it to start a web server for testing REST APIs.

74. What is `@MockBean` and `@AutowiredMockMvc`?

- **`@MockBean`:** A Spring-specific annotation that creates a Mockito mock for a bean and adds it to the application context. This replaces a real bean with a mock, allowing you to test a component in isolation without its dependencies.
- **`@AutowiredMockMvc`:** Used in conjunction with `@WebMvcTest`. It automatically configures `MockMvc`, a powerful object for testing MVC controllers by sending mock HTTP requests. It doesn't start a real server.

75. What is Mockito and how is it used in Spring Boot testing?

Mockito is a popular mocking framework for Java. It allows you to create mock objects for classes and interfaces. You can then define how these mock objects should behave when their methods are called. This is crucial for **unit testing**, as it lets you isolate the class under test from its dependencies.

76. How to test REST APIs in Spring Boot?

You can test REST APIs using `MockMvc` or by starting a real server.

- **`@WebMvcTest` with `MockMvc`:** The most common approach. It only loads the web layer and is faster than a full integration test.
- **`@SpringBootTest` with `TestRestTemplate`:** Starts a real embedded server, which is useful for testing a complete end-to-end flow from the client's perspective.

77. How to test database operations in Spring Boot?

You can test database interactions with **@DataJpaTest**. This annotation configures a slice of the application for testing the data layer. It replaces the real database with an in-memory database like H2 by default, making tests fast and isolated. It also provides a TestEntityManager for direct database access.

78. What is Testcontainers in Spring Boot testing?

Testcontainers is a library that provides lightweight, disposable instances of databases, message brokers, and more in Docker containers. This allows you to test your application against a real database (not an in-memory one) in a controlled and isolated environment, ensuring your tests are more representative of the production environment.

79. What is @DataJpaTest?

@DataJpaTest is a specialized integration test annotation. It auto-configures the Spring context to test the JPA components. It scans for @Entity classes and Spring Data JPA repositories, configures an in-memory database, and provides a TestEntityManager for querying. It's ideal for testing the data layer without loading the entire application.

80. Best practices for testing Spring Boot applications?

- **Use the right tool for the job:** Use unit tests for business logic, @WebMvcTest for the web layer, and @DataJpaTest for the data layer.
- **Favor slices over full integration tests:** @SpringBootTest should be used sparingly as it's the slowest.
- **Mock external services:** Use @MockBean to mock external APIs or services to make tests deterministic and faster.
- **Use Testcontainers for real-world scenarios:** Use Testcontainers to test against a real database to avoid discrepancies between your test and production environments.
- **Name tests clearly:** Use a descriptive naming convention for your test classes and methods.

---

9. Spring Boot Advanced Concepts

81. What is Spring Boot Auto-Configuration?

Auto-configuration is a core feature of Spring Boot. It's a mechanism that automatically configures your application based on the dependencies present in your classpath. For example, if you add the spring-boot-starter-web dependency, Spring Boot will automatically configure a DispatcherServlet, an embedded Tomcat server, and other web-related beans.

It works through a series of classes annotated with @ConditionalOn... (e.g., @ConditionalOnClass, @ConditionalOnMissingBean), which determine whether a certain configuration should be applied.

82. How does Spring Boot run an embedded server (Tomcat, Jetty, Undertow)?

Spring Boot's auto-configuration detects if a web server starter dependency (like `spring-boot-starter-tomcat`) is on the classpath. It then automatically configures and starts the corresponding embedded server instance when the application starts. The main `main()` method of the application effectively acts as the server process itself, eliminating the need for a separate WAR file and server installation.

83. What is the difference between WAR and JAR deployment in Spring Boot?

- **JAR (Java Archive):** A standard format for packaging Java applications. Spring Boot packages a stand-alone application into an executable JAR, including all dependencies and an embedded server. This is the **default and recommended way** to deploy Spring Boot applications.
- **WAR (Web Application Archive):** A standard format for packaging web applications to be deployed on an external web server like Tomcat or Jetty. You can configure a Spring Boot app to produce a WAR file, but this defeats the purpose of the embedded server and the "run-anywhere" nature of Spring Boot.

84. How does Spring Boot's `application.properties` vs `application.yml` work?

- **`application.properties`:** A flat, key-value based format. Simple and widely used.
- **`application.yml`:** A hierarchical, more readable format that supports complex nesting. It's often preferred for its clear structure.

Spring Boot can read from both formats. If both are present, properties from `application.properties` will override those from `application.yml`. You can also have multiple profile-specific files like `application-dev.properties` and `application-prod.properties`.

85. How to create profiles in Spring Boot (`@Profile`)?

Spring Profiles allow you to map configuration and components to specific environments (e.g., `dev`, `prod`, `test`).

- **`@Profile`:** You can annotate beans or `@Configuration` classes with `@Profile` to indicate that they should only be registered when a specific profile is active.
- **Profile-specific properties:** You can create separate configuration files for each profile (e.g., `application-dev.properties`).

You activate a profile using the command line (`--spring.profiles.active=dev`) or in your `application.properties` file.

86. What is configuration properties binding (`@ConfigurationProperties`)?

`@ConfigurationProperties` is a powerful way to externalize configuration. It allows you to bind a group of properties from your configuration file to a Java object. This provides type-safe and structured access to your configuration.

Example:

```
Java

@Configuration
@ConfigurationProperties(prefix = "mail")
public class MailProperties {
    private String host;
    private int port;
    // getters and setters
}

// In application.properties
mail.host=smtp.gmail.com
mail.port=587
```

87. How to externalize configuration in Spring Boot?

Spring Boot provides a very flexible way to externalize configuration. Properties can be defined in a number of locations, in this order of precedence (highest first):

- Command-line arguments.
- Environment variables.
- Profile-specific application properties (`application-{profile}.properties`).
- Default application properties (`application.properties`).

This allows you to deploy the same application artifact in different environments with different configurations.

88. What is the difference between synchronous and asynchronous calls in Spring Boot?

- **Synchronous:** A method call that **waits for the result** before continuing. The thread that made the call is blocked until the called method returns. This is the default behavior.
- **Asynchronous:** A method call that **does not block** and returns immediately. The task runs in a separate thread. This is useful for long-running operations. You can enable this behavior in Spring Boot with `@EnableAsync` and `@Async`.

89. What is Spring Boot Scheduler (`@Scheduled`)?

The `@Scheduled` annotation is used to create **scheduled tasks**. It allows you to run a method at a fixed interval, at a fixed delay, or using a cron expression. You must also enable scheduling with `@EnableScheduling` on your main application class or a configuration class.

Example:

```
Java


@Component
public class MyScheduler {
    @Scheduled(fixedRate = 5000) // Run every 5 seconds
    public void doSomething() {
        System.out.println("Scheduled task running!");
    }
}
```

90. What is Spring Boot Async (@Async)?

The @Async annotation marks a method to be run **asynchronously** in a separate thread. This is useful for non-blocking operations. When a method is called, Spring will execute it in a thread from a thread pool and immediately return control to the caller. You must enable asynchronous processing with @EnableAsync.

---

10. Microservices with Spring Boot

91. What are microservices? How does Spring Boot support them?

Microservices is an architectural style that structures an application as a collection of small, independent, and loosely coupled services. Each service is built around a specific business capability, can be developed and deployed independently, and communicates with other services, often via a lightweight protocol like REST.

Spring Boot is an excellent choice for building microservices because its features align perfectly with the microservices philosophy:

- **Embedded servers:** Each service can be a standalone, executable JAR.
- **Auto-configuration:** Rapid development of each individual service.
- **Starters:** Easy to pull in necessary dependencies for each service.
- **Actuator:** Provides built-in monitoring for each service.

92. What is Spring Cloud?

Spring Cloud is a collection of libraries and tools that build on top of Spring Boot to make it easier to develop and deploy cloud-native applications, particularly microservices. It provides a set of patterns and solutions for common challenges in distributed systems, such as service discovery, distributed configuration, and load balancing.

93. How to implement service discovery in Spring Boot (Eureka)?

Service Discovery is a mechanism for services to find and communicate with each other without hardcoding their locations.

- **Eureka Server:** A registry that services can register themselves with.

- **Eureka Client:** A library that allows a microservice to register itself with the Eureka server and also discover other services.

You would add the `spring-cloud-starter-netflix-eureka-server` dependency to your discovery server and `spring-cloud-starter-netflix-eureka-client` to each microservice.

94. What is API Gateway (Zuul, Spring Cloud Gateway)?

An **API Gateway** is a single entry point for all client requests. It provides a single, unified API for a set of microservices. It can handle routing, load balancing, security, and monitoring.

- **Zuul:** An older, Netflix-based API Gateway.
- **Spring Cloud Gateway:** A more modern, reactive API Gateway that is the preferred choice for new projects.

95. How to implement load balancing with Ribbon?

Ribbon is a client-side load balancer that works with a service discovery tool like Eureka. When a service needs to call another service, it gets a list of available instances from the discovery server. Ribbon then applies a load-balancing algorithm (e.g., round-robin) to choose which instance to call.

96. What is Hystrix and Circuit Breaker pattern in Spring Boot?

- **Circuit Breaker Pattern:** A design pattern used in microservices to prevent a failure in one service from cascading to others. When a service repeatedly fails, the circuit breaker "opens," and subsequent calls fail immediately without attempting to call the faulty service.
- **Hystrix:** A Netflix library for implementing the Circuit Breaker pattern. It provides a way to wrap calls to other services and define a fallback method to execute if the primary call fails or times out. (Note: Hystrix is no longer under active development, and Resilience4j is the recommended alternative.)

97. What is Config Server in Spring Cloud?

A **Spring Cloud Config Server** provides centralized configuration management for all microservices. It can pull configuration from a Git repository or other sources, and services can pull their configuration from the server at startup. This allows you to manage configuration for all services in one place and change it without rebuilding and redeploying them.

98. How to secure microservices in Spring Boot?

In a microservices architecture, you can secure communication between services in several ways:

- **API Gateway Security:** The gateway handles authentication and authorization for external clients.
- **Service-to-Service Security:** Use technologies like OAuth2 or JWTs for internal service communication.
- **Shared Secrets/Token-based:** A simpler approach for internal communication.

99. What is distributed tracing in Spring Boot microservices?

In a microservices environment, a single request can fan out to many services. **Distributed tracing** provides a way to track the entire request flow from start to finish across all services. Spring Cloud Sleuth is an excellent tool for this, as it adds trace and span IDs to log and HTTP headers, which can then be visualized in a tool like Zipkin.

100. What are best practices for building microservices with Spring Boot?

- **Keep services small and focused:** Each service should have a single responsibility.
- **Decentralize data management:** Each service should own its data store.
- **Design for failure:** Use circuit breakers, bulkheads, and retries.
- **Centralize configuration:** Use Spring Cloud Config to manage properties.
- **Implement service discovery:** Use Eureka or a similar tool.
- **Use an API Gateway:** Provide a single entry point for clients.
- **Monitor everything:** Use Actuator, Micrometer, and distributed tracing.
- **Automate everything:** Use CI/CD pipelines for building and deploying.