Spring Boot Interview Questions

Comprehensive Guide with Detailed Answers

Core Java / Spring Boot Basics

1. What is Spring Boot and what are its main features?

Spring Boot is an extension of the Spring Framework that simplifies the development of production-ready applications. Its main features include:

- **Auto-configuration:** Automatically configures Spring application based on dependencies present in the classpath
- **Standalone:** Creates standalone applications with embedded servers (Tomcat, Jetty, Undertow)
- **Opinionated defaults:** Provides sensible defaults to reduce configuration
- **Production-ready features:** Includes metrics, health checks, and externalized configuration via Actuator
- **No code generation:** Eliminates the need for XML configuration
- **Starter dependencies:** Simplified dependency management through starter POMs

2. What are the advantages of Spring Boot over the Spring Framework?

Spring Boot provides several advantages over traditional Spring Framework:

- **Reduced boilerplate code:** Minimal configuration required to start an application
- **Embedded server:** No need to deploy WAR files to external servers
- **Simplified dependency management:** Starter dependencies eliminate version conflicts
- **Auto-configuration:** Intelligent defaults based on classpath scanning
- **Production-ready:** Built-in Actuator for monitoring and management
- **Faster development:** Quick project setup with Spring Initializr
- **Microservices-friendly:** Ideal for building cloud-native applications

3. What is the use and effect of the @SpringBootApplication annotation?

`@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` : Scans for components in the current package and sub-packages

It's typically placed on the main class and serves as the entry point for the Spring Boot application. When the application starts, it triggers component scanning, auto-configuration, and bean registration.

4. What is Spring Initializr and how is it used to bootstrap a Spring Boot project?

Spring Initializr is a web-based tool (start.spring.io) that generates Spring Boot project structures. It allows developers to:

- Select project metadata (group, artifact, name)
- Choose build tool (Maven or Gradle)
- Select Java version
- Add dependencies (Spring Web, JPA, Security, etc.)
- Choose packaging type (JAR or WAR)

It generates a pre-configured project with proper directory structure, build files, and main application class, allowing developers to start coding immediately without manual setup.

5. Explain the concept of Spring Boot Starters.

Spring Boot Starters are pre-configured dependency descriptors that bundle commonly used libraries together. Key benefits include:

- **Simplified dependency management:** Single dependency includes all related libraries
- **Version compatibility:** Ensures compatible versions of dependencies work together
- **Reduced configuration:** No need to specify individual library versions

Common starters include:

- `spring-boot-starter-web` : For building web applications (includes Spring MVC, Tomcat, Jackson)
- `spring-boot-starter-data-jpa` : For JPA with Hibernate
- `spring-boot-starter-security` : For Spring Security
- `spring-boot-starter-test` : For testing frameworks

6. How does auto-configuration work in Spring Boot?

Auto-configuration in Spring Boot works through conditional bean registration:

- **Classpath scanning:** Spring Boot scans the classpath for specific libraries
- **Conditional annotations:** Uses `@ConditionalOnClass`, `@ConditionalOnMissingBean`, `@ConditionalOnProperty` to determine if configuration should be applied
- **META-INF/spring.factories:** Auto-configuration classes are listed in this file within starter JARs
- **Intelligent defaults:** Configures beans with sensible defaults that can be overridden

For example, if H2 database is on the classpath, Spring Boot automatically configures an in-memory database. Developers can override these defaults by providing their own bean definitions.

7. What is the purpose of embedded servers (like Tomcat, Jetty) in Spring Boot?

Embedded servers in Spring Boot serve several purposes:

- **Standalone deployment:** Applications run as executable JAR files without external server installation
- **Simplified deployment:** No need to package as WAR and deploy to application servers
- **Consistent environment:** Same server configuration across development and production
- **Microservices-friendly:** Each service can run independently with its own server
- **Quick startup:** Faster development cycle with embedded server restarts

Default embedded server is Tomcat, but can be switched to Jetty or Undertow by excluding Tomcat dependency and adding the alternative.

8. How can you change the default port of the embedded server in a Spring Boot application?

The default port (8080) can be changed in multiple ways:

```
# application.properties server.port=9090 # application.yml server: port: 9090 #
Command line java -jar application.jar --server.port=9090 # Environment variable
SERVER_PORT=9090 # Programmatically @Configuration public class ServerConfig
implements WebServerFactoryCustomizer<ConfigurableWebServerFactory> { @Override
public void customize(ConfigurableWebServerFactory factory) { factory.setPort(9090);
} }
```

Setting `server.port=0` assigns a random available port, useful for testing or running multiple instances.

9. What is the difference between @Controller and @RestController in Spring Boot?

- **@Controller:** Traditional Spring MVC controller that returns view names (HTML pages). Typically used with `@ResponseBody` on methods that return data instead of views.
- **@RestController:** Convenience annotation combining `@Controller` and `@ResponseBody`. Every method automatically serializes return values to JSON/XML instead of resolving to views. Used for building RESTful web services.

```
// Traditional Controller @Controller public class WebController { @GetMapping("/page") public String getPage() { return "homepage"; // returns view name } @GetMapping("/data") @ResponseBody public User getData() { return new User(); // returns JSON } } // REST Controller @RestController public class ApiController { @GetMapping("/users") public List<User> getUsers() { return userList; // automatically returns JSON } }
```

10. What are Spring Profiles and how are they used?

Spring Profiles provide a way to segregate application configuration and make it available only in certain environments. Uses include:

- **Environment-specific configuration:** Different settings for dev, test, and production
- **Bean activation:** Load specific beans based on active profile
- **Property files:** Use profile-specific property files like `application-dev.properties`

```
# Activating profiles spring.profiles.active=dev,mysql # Profile-specific beans @Configuration @Profile("dev") public class DevConfig { @Bean public DataSource dataSource() { return new H2DataSource(); } } @Configuration @Profile("prod") public class ProdConfig { @Bean public DataSource dataSource() { return new MySQLDataSource(); } } # Profile-specific properties application-dev.properties application-prod.properties
```

Dependency Management, Configuration & Annotations

11. What are the basic annotations used in Spring Boot?

Essential Spring Boot annotations include:

- `@Component`: Generic stereotype for any Spring-managed component
- `@Service`: Specialization of @Component for service layer
- `@Repository`: Specialization for data access layer, enables exception translation
- `@Controller`: Marks a class as Spring MVC controller
- `@RestController`: Combines @Controller and @ResponseBody
- `@Autowired`: Enables dependency injection
- `@Configuration`: Indicates class contains bean definitions
- `@Bean`: Declares a method that returns a Spring bean
- `@Value`: Injects values from property files
- `@GetMapping/@PostMapping/@PutMapping/@DeleteMapping`: HTTP method-specific request mappings

12. What is the difference between @Component, @Service, and @Repository annotations?

All three are specializations of `@Component` with semantic differences:

- **@Component:** Generic stereotype for any Spring-managed component. Use when a class doesn't fit other stereotypes.
- **@Service:** Indicates the class holds business logic. Used in the service layer. No additional functionality beyond @Component, but provides semantic clarity.
- **@Repository:** Marks the class as a Data Access Object (DAO). Provides additional benefit of automatic exception translation from database-specific exceptions to Spring's DataAccessException hierarchy.

Using specific annotations improves code readability and allows for future enhancements. Spring can apply specific behaviors to classes based on their stereotype.

13. What is YAML configuration in Spring Boot and how is it different from properties files?

YAML (YAML Ain't Markup Language) is a human-readable data serialization format that Spring Boot supports as an alternative to properties files.

```
# application.properties spring.datasource.url=jdbc:mysql://localhost:3306/db spring.datasource.username=root spring.datasource.password=pass server.port=8080 # application.yml spring: datasource: url: jdbc:mysql://localhost:3306/db username: root password: pass server: port: 8080
```

Differences:

- **Structure:** YAML uses indentation-based hierarchy, properties use dot notation
- **Readability:** YAML is more readable for complex configurations
- **Lists/Arrays:** YAML has native support for lists, properties require indexed notation
- **Comments:** YAML uses #, properties also use #
- **Profile separation:** YAML can have multiple profiles in one file using ---

14. How do you exclude a specific auto-configuration class in Spring Boot?

Auto-configuration classes can be excluded in several ways:

```
# Using @SpringBootApplication @SpringBootApplication(exclude = {DataSourceAutoConfiguration.class}) public class Application { public static void main(String[] args) { SpringApplication.run(Application.class, args); } } # Using @EnableAutoConfiguration @Configuration @EnableAutoConfiguration(exclude = {DataSourceAutoConfiguration.class}) public class AppConfig {} # Using properties file
spring.autoconfigure.exclude=org.springframework.boot.autoconfigure.jdbc.DataSourceAuto(
# Multiple exclusions @SpringBootApplication(exclude = { DataSourceAutoConfiguration.class, HibernateJpaAutoConfiguration.class })
```

Exclusion is useful when you want to provide custom configuration or when certain auto-configurations conflict with your requirements.

15. Can you create a non-web (console/batch) Spring Boot application? How?

Yes, Spring Boot can create console or batch applications by implementing `CommandLineRunner` or `ApplicationRunner` interfaces:

```
@SpringBootApplication public class ConsoleApplication implements CommandLineRunner { public static void main(String[] args) { SpringApplication.run(ConsoleApplication.class, args); } @Override public void run(String... args) throws Exception { System.out.println("Console application started"); // Business logic here } } // Or using ApplicationRunner @Component public class MyRunner implements ApplicationRunner { @Override public void run(ApplicationArguments args) throws Exception { // Access parsed arguments List<String> nonOptionArgs = args.getNonOptionArgs(); } }
```

To create a non-web application, exclude web starter dependencies or set `spring.main.web-application-type=none` in properties.

16. What is dependency management in Spring Boot and how is it simplified?

Spring Boot simplifies dependency management through:

- **Parent POM:** `spring-boot-starter-parent` manages versions of common dependencies
- **Dependency BOM:** Bill of Materials defines compatible versions
- **Starter dependencies:** Bundle related dependencies together
- **Version omission:** No need to specify versions for managed dependencies

```
<parent> <groupId>org.springframework.boot</groupId> <artifactId>spring-boot-starter-parent</artifactId> <version>3.2.0</version> </parent> <dependencies> <!-- No version needed --> <dependency> <groupId>org.springframework.boot</groupId> <artifactId>spring-boot-starter-web</artifactId> </dependency> </dependencies>
```

This eliminates version conflicts and ensures compatible dependency combinations.

17. What is the role of @ComponentScan in Spring Boot?

`@ComponentScan` tells Spring where to look for components, configurations, and services:

- **Default behavior:** Scans the package of the annotated class and all sub-packages
- **Custom packages:** Can specify base packages to scan
- **Included in @SpringBootApplication:** Automatically scans from the main application package

```
// Default - scans current package and sub-packages @SpringBootApplication public
class Application {} // Custom packages @Configuration @ComponentScan(basePackages =
{"com.example.service", "com.example.repository"}) public class AppConfig {} // Using
base package classes (type-safe) @ComponentScan(basePackageClasses =
{ServiceMarker.class, RepoMarker.class}) public class AppConfig {} // Exclude
specific components @ComponentScan( basePackages = "com.example", excludeFilters =
@Filter(type = FilterType.REGEX, pattern = "com.example.exclude.*") )
```

18. What is the flow of HTTP requests through a Spring Boot application?

HTTP request flow in Spring Boot follows this sequence:

1. **DispatcherServlet:** Front controller receives all HTTP requests
2. **Handler Mapping:** Determines which controller method should handle the request based on URL and HTTP method
3. **Interceptors (Pre):** Execute before the controller method (authentication, logging)
4. **Controller:** Handler method processes the request
5. **Service Layer:** Business logic execution
6. **Repository Layer:** Data access and persistence
7. **Controller Return:** Returns data or view name
8. **View Resolver:** Resolves view name to actual view (if not REST)
9. **Interceptors (Post):** Execute after controller method
10. **Response:** Serialized to JSON/XML (REST) or rendered view (MVC)
11. **Client:** Receives HTTP response

19. How do you enable debug logs in a Spring Boot application?

Debug logging can be enabled through several methods:

```
# application.properties logging.level.root=DEBUG
logging.level.org.springframework.web=DEBUG logging.level.com.example.myapp=DEBUG #
application.yml logging: level: root: DEBUG org.springframework.web: DEBUG
com.example.myapp: DEBUG # Command line java -jar app.jar --debug java -jar app.jar -
-logging.level.root=DEBUG # Enable Spring Boot debug mode (shows auto-configuration
report) java -jar app.jar --debug # or in properties debug=true # Logback
configuration (logback-spring.xml) <logger name="com.example" level="DEBUG"/>
```

Spring Boot debug mode provides detailed auto-configuration report showing what was configured and why.

20. Discuss how Spring Boot handles externalized configurations.

Spring Boot supports multiple externalized configuration sources with a specific precedence order (higher overrides lower):

Command line arguments

Java System properties

OS environment variables

Profile-specific properties (application-{profile}.properties)

Application properties (application.properties/yml)

@PropertySource annotations

Default properties

```
# Environment variables export DATABASE_URL=jdbc:mysql://prod-db:3306/mydb # Command
line java -jar app.jar --server.port=9090 --spring.profiles.active=prod #
application.properties server.port=8080 app.name=MyApp # Profile-specific
application-dev.properties application-prod.properties # Accessing in code
@Value("${app.name}") private String appName; @ConfigurationProperties(prefix =
"app") public class AppProperties { private String name; // getters/setters }
```

This allows flexible configuration management across different environments without code changes.

Data Access, Persistence & Databases

21. How do you integrate and configure a relational database (e.g., MySQL) with Spring Boot?

Integrating MySQL with Spring Boot involves:

```
<!-- pom.xml dependencies --> <dependency> <groupId>org.springframework.boot</groupId> <artifactId>spring-boot-starter-data-jpa</artifactId> </dependency> <dependency> <groupId>mysql</groupId> <artifactId>mysql-connector-java</artifactId> </dependency> # application.properties spring.datasource.url=jdbc:mysql://localhost:3306/mydb spring.datasource.username=root spring.datasource.password=password spring.datasource.driver-class-name=com.mysql.cj.jdbc.Driver spring.jpa.hibernate.ddl-auto=update spring.jpa.show-sql=true spring.jpa.properties.hibernate.dialect=org.hibernate.dialect.MySQL8Dialect # Connection pool configuration spring.datasource.hikari.maximum-pool-size=10 spring.datasource.hikari.minimum-idle=5
```

Spring Boot auto-configures DataSource, JPA, and transaction management based on these properties.

22. What is Spring Data JPA and how does it relate to Hibernate?

Spring Data JPA is a Spring module that simplifies data access layer implementation:

- **Repository abstraction:** Eliminates boilerplate DAO code
- **JPA provider:** Uses Hibernate as default JPA implementation
- **Query methods:** Derives queries from method names
- **Custom queries:** Supports JPQL and native SQL via @Query

```
// Entity @Entity public class User { @Id @GeneratedValue(strategy = GenerationType.IDENTITY) private Long id; private String email; private String name; } // Repository interface - no implementation needed public interface UserRepository extends JpaRepository<User, Long> { // Spring Data generates implementation List<User> findByEmail(String email); List<User> findByNameContaining(String name); @Query("SELECT u FROM User u WHERE u.age > :age") List<User> findUsersOlderThan(@Param("age") int age); } // Usage in service @Service public class UserService { @Autowired private UserRepository userRepository; public User saveUser(User user) { return userRepository.save(user); } }
```

Hibernate handles ORM (Object-Relational Mapping) while Spring Data JPA provides repository pattern on top.

23. How do you configure multiple data sources in Spring Boot?

Multiple data sources require manual configuration:

```
# application.properties # Primary datasource
spring.datasource.primary.url=jdbc:mysql://localhost:3306/db1
spring.datasource.primary.username=user1 spring.datasource.primary.password=pass1 #
Secondary datasource
spring.datasource.secondary.url=jdbc:postgresql://localhost:5432/db2
spring.datasource.secondary.username=user2 spring.datasource.secondary.password=pass2
// Configuration class @Configuration public class DataSourceConfig { @Primary @Bean
@ConfigurationProperties("spring.datasource.primary") public DataSource
primaryDataSource() { return DataSourceBuilder.create().build(); } @Bean
@ConfigurationProperties("spring.datasource.secondary") public DataSource
secondaryDataSource() { return DataSourceBuilder.create().build(); } }
```

Each data source requires separate EntityManager and TransactionManager configurations.

24. What is the difference between JDBC and JPA in Spring Boot?

JDBC (Java Database Connectivity):

- Low-level database access
- Requires manual SQL queries
- Manual result set mapping to objects
- More control over SQL execution
- Better performance for simple queries
- Uses JdbcTemplate in Spring

JPA (Java Persistence API):

- High-level ORM (Object-Relational Mapping) specification
- Automatic SQL generation from entity mappings
- Automatic object-relational mapping
- Database-independent (dialect-based)
- Built-in caching and lazy loading
- More abstraction, less boilerplate code
- Uses Hibernate as implementation in Spring Boot

Use JDBC for simple queries or performance-critical operations; use JPA for complex domain models and rapid development.

25. How do you integrate a NoSQL database (e.g., MongoDB) in Spring Boot?

MongoDB integration with Spring Boot:

```
<!-- pom.xml --> <dependency> <groupId>org.springframework.boot</groupId> <artifactId>spring-boot-starter-data-mongodb</artifactId> </dependency> # application.properties spring.data.mongodb.uri=mongodb://localhost:27017/mydb spring.data.mongodb.database=mydb // Document entity @Document(collection = "users") public class User { @Id private String id; private String name; private String email; } // Repository public interface UserRepository extends MongoRepository<User, String> { List<User> findByName(String name); List<User> findByEmailContaining(String email); } // Service usage @Service public class UserService { @Autowired private UserRepository userRepository; public User createUser(User user) { return userRepository.save(user); } }
```

Spring Data MongoDB provides similar repository abstraction as Spring Data JPA.

REST, Web Services & API Design

26. How do you build RESTful APIs using Spring Boot?

Building REST APIs in Spring Boot:

```
@RestController @RequestMapping("/api/users") public class UserController { @Autowired private UserService userService; // GET all users @GetMapping public List<User> getAllUsers() { return userService.findAll(); } // GET user by ID @GetMapping("/{id}") public ResponseEntity<User> getUserById(@PathVariable Long id) { return userService.findById(id) .map(ResponseEntity::ok) .orElse(ResponseEntity.notFound().build()); } // POST create user @PostMapping public ResponseEntity<User> createUser(@RequestBody @Valid User user) { User created = userService.save(user); return ResponseEntity.status(HttpStatus.CREATED).body(created); } // PUT update user @PutMapping("/{id}") public ResponseEntity<User> updateUser(@PathVariable Long id, @RequestBody User user) { return userService.update(id, user) .map(ResponseEntity::ok) .orElse(ResponseEntity.notFound().build()); } // DELETE user @DeleteMapping("/{id}") public ResponseEntity<Void> deleteUser(@PathVariable Long id) { userService.delete(id); return ResponseEntity.noContent().build(); } }
```

27. Explain how exception handling is done in Spring Boot REST APIs.

Spring Boot provides centralized exception handling using `@ControllerAdvice` and `@ExceptionHandler`:

```
// Custom exception public class ResourceNotFoundException extends RuntimeException { public ResourceNotFoundException(String message) { super(message); } } // Global exception handler @ControllerAdvice public class GlobalExceptionHandler { @ExceptionHandler(ResourceNotFoundException.class) public ResponseEntity<ErrorResponse> handleResourceNotFound(ResourceNotFoundException ex) { ErrorResponse error = new ErrorResponse( HttpStatus.NOT_FOUND.value(), ex.getMessage(), System.currentTimeMillis() ); return new ResponseEntity<>(error, HttpStatus.NOT_FOUND); } @ExceptionHandler(MethodArgumentNotValidException.class) public ResponseEntity<ErrorResponse> handleValidationException(MethodArgumentNotValidException ex) { Map<String, String> errors = new HashMap<>(); ex.getBindingResult().getFieldErrors().forEach(error -> errors.put(error.getField(), error.getDefaultMessage()) ); ErrorResponse errorResponse = new ErrorResponse( HttpStatus.BAD_REQUEST.value(), "Validation failed", errors ); return new ResponseEntity<>(errorResponse, HttpStatus.BAD_REQUEST); } @ExceptionHandler(Exception.class) public ResponseEntity<ErrorResponse> handleGlobalException(Exception ex) { ErrorResponse error = new ErrorResponse( HttpStatus.INTERNAL_SERVER_ERROR.value(), "Internal server error", System.currentTimeMillis() ); return new ResponseEntity<>(error, HttpStatus.INTERNAL_SERVER_ERROR); } }
```

28. What is HATEOAS and how might you implement it in Spring Boot?

HATEOAS (Hypermedia As The Engine Of Application State) adds hypermedia links to REST responses, making APIs self-documenting and discoverable:

```
<dependency> <groupId>org.springframework.boot</groupId> <artifactId>spring-boot-starter-hateoas</artifactId> </dependency> @RestController @RequestMapping("/api/users") public class UserController { @GetMapping("/{id}") public EntityModel<User> getUserById(@PathVariable Long id) { User user = userService.findById(id); EntityModel<User> model = EntityModel.of(user); model.add(linkTo(methodOn(UserController.class).getUserById(id)).withSelfRel()); model.add(linkTo(methodOn(UserController.class).getAllUsers()).withRel("all-users")); model.add(linkTo(methodOn(OrderController.class).getUserOrders(id)).withRel("orders")); return model; } } // Response example: { "id": 1, "name": "John", "_links": { "self": {"href": "http://localhost:8080/api/users/1"}, "all-users": {"href": "http://localhost:8080/api/users"}, "orders": {"href": "http://localhost:8080/api/users/1/orders"} } }
```

HATEOAS makes APIs more RESTful by providing clients with navigation options.

29. How do you version REST APIs in a Spring Boot application?

REST API versioning can be implemented using several strategies:

- **URI versioning:** Version in URL path
- **Request parameter versioning:** Version as query parameter
- **Header versioning:** Version in custom header
- **Content negotiation:** Version in Accept header

```
// URI versioning @RestController @RequestMapping("/api/v1/users") public class UserControllerV1 { } @RestController @RequestMapping("/api/v2/users") public class UserControllerV2 { } // Request parameter versioning @GetMapping(value = "/users", params = "version=1") public List<UserV1> getUsersV1() { } @GetMapping(value = "/users", params = "version=2") public List<UserV2> getUsersV2() { } // Header versioning @GetMapping(value = "/users", headers = "X-API-VERSION=1") public List<UserV1> getUsersV1() { } // Content negotiation @GetMapping(value = "/users", produces = "application/vnd.company.app-v1+json") public List<UserV1> getUsersV1() { } @GetMapping(value = "/users", produces = "application/vnd.company.app-v2+json") public List<UserV2> getUsersV2() { }
```

URI versioning is most common and explicit, while content negotiation is more RESTful but complex.

30. What are some best practices for designing microservices with Spring Boot?

Key microservices design best practices:

- **Single Responsibility:** Each service should have one business capability
- **Loose Coupling:** Services should be independently deployable
- **API Gateway:** Use Spring Cloud Gateway for routing and cross-cutting concerns
- **Service Discovery:** Implement with Eureka or Consul
- **Configuration Management:** Centralize with Spring Cloud Config
- **Circuit Breaker:** Use Resilience4j for fault tolerance
- **Distributed Tracing:** Implement with Sleuth and Zipkin
- **Database per Service:** Each service owns its data
- **Asynchronous Communication:** Use message queues for loose coupling
- **Health Checks:** Expose health endpoints via Actuator
- **Containerization:** Package as Docker containers
- **Security:** Implement OAuth2/JWT for authentication
- **Monitoring:** Use Prometheus and Grafana

Microservices, Messaging & Infrastructure

31. What is microservices architecture and how does Spring Boot support it?

Microservices architecture breaks applications into small, independent services that communicate over network. Spring Boot supports microservices through:

- **Standalone JARs:** Each service runs independently with embedded server
- **Spring Cloud:** Provides tools for distributed systems (Config, Gateway, Eureka)
- **RESTful APIs:** Easy service-to-service communication
- **Actuator:** Health checks and metrics for monitoring
- **Lightweight:** Fast startup and low resource consumption

Key characteristics of microservices:

- Independently deployable and scalable
- Technology diversity (different services can use different tech stacks)
- Fault isolation (failure in one service doesn't crash entire system)
- Organized around business capabilities
- Decentralized data management

32. How do you implement inter-service communication in microservices using Spring Boot?

Inter-service communication can be synchronous or asynchronous:

```
// Synchronous - RestTemplate (legacy) @Service public class OrderService {
@Autowired private RestTemplate restTemplate; public User getUser(Long userId) {
return restTemplate.getForObject( "http://user-service/api/users/" + userId,
User.class ); } } // Synchronous - WebClient (recommended) @Service public class
OrderService { @Autowired private WebClient.Builder webClientBuilder; public
Mono<User> getUser(Long userId) { return webClientBuilder.build() .get()
.uri("http://user-service/api/users/" + userId) .retrieve() .bodyToMono(User.class);
} } // Synchronous - Feign Client (declarative) @FeignClient(name = "user-service")
public interface UserClient { @GetMapping("/api/users/{id}") User
getUserById(@PathVariable Long id); } // Asynchronous - Message Queue
(RabbitMQ/Kafka) @Service public class OrderService { @Autowired private
RabbitTemplate rabbitTemplate; public void createOrder(Order order) {
rabbitTemplate.convertAndSend("order-exchange", "order.created", order); } }
```

Use synchronous for immediate response needs, asynchronous for eventual consistency and better resilience.

33. How do you integrate message brokers (e.g., Kafka, RabbitMQ) with Spring Boot?

RabbitMQ Integration:

```
<dependency> <groupId>org.springframework.boot</groupId> <artifactId>spring-boot-
starter-amqp</artifactId> </dependency> // Configuration
spring.rabbitmq.host=localhost spring.rabbitmq.port=5672
spring.rabbitmq.username=guest spring.rabbitmq.password=guest // Producer @Service
public class MessageProducer { @Autowired private RabbitTemplate rabbitTemplate;
public void sendMessage(String message) { rabbitTemplate.convertAndSend("my-
exchange", "routing-key", message); } } // Consumer @Component public class
MessageConsumer { @RabbitListener(queues = "my-queue") public void
receiveMessage(String message) { System.out.println("Received: " + message); } }
```

Kafka Integration:

```
<dependency> <groupId>org.springframework.kafka</groupId> <artifactId>spring-
kafka</artifactId> </dependency> // Configuration spring.kafka.bootstrap-
servers=localhost:9092 spring.kafka.consumer.group-id=my-group // Producer @Service
public class KafkaProducer { @Autowired private KafkaTemplate<String, String>
kafkaTemplate; public void sendMessage(String message) { kafkaTemplate.send("my-
topic", message); } } // Consumer @Component public class KafkaConsumer {
@KafkaListener(topics = "my-topic", groupId = "my-group") public void listen(String
message) { System.out.println("Received: " + message); } }
```

34. What is service discovery and how is it implemented in Spring Cloud with Spring Boot?

Service discovery allows services to find and communicate with each other dynamically without hardcoding locations. Spring Cloud uses Netflix Eureka:

```
// Eureka Server <dependency> <groupId>org.springframework.cloud</groupId> <artifactId>spring-cloud-starter-netflix-eureka-server</artifactId> </dependency> @SpringBootApplication @EnableEurekaServer public class EurekaServerApplication { } # application.properties server.port=8761 eureka.client.register-with-eureka=false eureka.client.fetch-registry=false // Eureka Client (Service Registration) <dependency> <groupId>org.springframework.cloud</groupId> <artifactId>spring-cloud-starter-netflix-eureka-client</artifactId> </dependency> @SpringBootApplication @EnableDiscoveryClient public class UserServiceApplication { } # application.properties spring.application.name=user-service eureka.client.service-url.defaultZone=http://localhost:8761/eureka // Service consumption with load balancing @Configuration public class RestTemplateConfig { @Bean @LoadBalanced public RestTemplate restTemplate() { return new RestTemplate(); } } @Service public class OrderService { @Autowired private RestTemplate restTemplate; public User getUser(Long id) { // Service name instead of URL return restTemplate.getForObject( "http://user-service/api/users/" + id, User.class ); } }
```

Eureka enables automatic service registration, discovery, and client-side load balancing.

35. Explain circuit breaker patterns and how they are used in Spring Boot microservices.

Circuit breaker prevents cascading failures by stopping requests to failing services. Resilience4j is the recommended implementation:

```
<dependency> <groupId>io.github.resilience4j</groupId> <artifactId>resilience4j-
spring-boot2</artifactId> </dependency> # application.yml
resilience4j.circuitbreaker: instances: userService: slidingWindowSize: 10
failureRateThreshold: 50 waitDurationInOpenState: 10000
permittedNumberOfCallsInHalfOpenState: 3 @Service public class OrderService {
@CircuitBreaker(name = "userService", fallbackMethod = "getUserFallback") public User
getUser(Long id) { return restTemplate.getForObject( "http://user-service/api/users/"
+ id, User.class ); } // Fallback method public User getUserFallback(Long id,
Exception ex) { return new User(id, "Default User", "default@email.com"); } }
```

Circuit breaker states:

- **Closed:** Normal operation, requests pass through
- **Open:** Too many failures, requests immediately fail with fallback
- **Half-Open:** Limited requests allowed to test if service recovered

Additional Resilience4j patterns: Retry, RateLimiter, Bulkhead, TimeLimiter.

Monitoring, Production-Readiness & Performance

36. What is the purpose of the Spring Boot Actuator and how do you enable it?

Spring Boot Actuator provides production-ready features for monitoring and managing applications:

```
<dependency> <groupId>org.springframework.boot</groupId> <artifactId>spring-boot-starter-actuator</artifactId> </dependency> # application.properties # Expose all endpoints management.endpoints.web.exposure.include=* # Expose specific endpoints management.endpoints.web.exposure.include=health,info,metrics # Base path for actuator endpoints management.endpoints.web.base-path=/actuator # Show detailed health information management.endpoint.health.show-details=always
```

Key features:

- Application health monitoring
- Metrics collection and exposure
- Application configuration visibility
- Thread dumps and heap dumps
- HTTP request tracing
- Environment property inspection

37. What are some key endpoints provided by Actuator?

Important Actuator endpoints:

- **/actuator/health:** Application health status (UP, DOWN, OUT_OF_SERVICE)
- **/actuator/info:** Custom application information
- **/actuator/metrics:** Application metrics (JVM, HTTP requests, custom metrics)
- **/actuator/beans:** All Spring beans in the application context
- **/actuator/env:** Environment properties and configuration
- **/actuator/mappings:** All @RequestMapping paths
- **/actuator/loggers:** View and modify logging levels at runtime
- **/actuator/threaddump:** Thread dump for debugging
- **/actuator/heapdump:** Heap dump for memory analysis
- **/actuator/httptrace:** Recent HTTP request-response exchanges
- **/actuator/prometheus:** Metrics in Prometheus format

Custom health indicators can be created by implementing HealthIndicator interface. Actuator endpoints should be secured in production.

38. How can you optimize startup time for a Spring Boot application?

Strategies to optimize Spring Boot startup time:

- **Lazy initialization:** `spring.main.lazy-initialization=true`
- **Exclude unused auto-configurations:** Remove unnecessary starters and exclude auto-config classes
- **Use Spring Native:** Compile to native image with GraalVM
- **Reduce component scanning:** Specify exact packages in @ComponentScan
- **Optimize dependencies:** Remove unused dependencies
- **Use profiles:** Load only necessary beans per environment
- **Disable Actuator in dev:** Enable only in production
- **JVM tuning:** Optimize heap size and GC settings
- **Use Class Data Sharing (CDS):** Share class metadata between JVM instances

```
# Lazy initialization spring.main.lazy-initialization=true # Exclude auto-
configurations
spring.autoconfigure.exclude=org.springframework.boot.autoconfigure.jdbc.DataSourceAutoC
# JVM options java -XX:+UseSerialGC -Xss256k -Xms64m -Xmx512m -jar app.jar
```

39. How do you implement caching in Spring Boot?

Spring Boot provides caching abstraction with multiple cache providers:

```
<dependency> <groupId>org.springframework.boot</groupId> <artifactId>spring-boot-
starter-cache</artifactId> </dependency> // Enable caching @SpringBootApplication
@EnableCaching public class Application { } // Service with caching @Service public
class UserService { @Cacheable(value = "users", key = "#id") public User
getUserById(Long id) { // Expensive database call return userRepository.findById(id);
} @CachePut(value = "users", key = "#user.id") public User updateUser(User user) {
return userRepository.save(user); } @CacheEvict(value = "users", key = "#id") public
void deleteUser(Long id) { userRepository.deleteById(id); } @CacheEvict(value =
"users", allEntries = true) public void clearCache() { // Clears all entries } } //
External cache providers # Redis spring.cache.type=redis spring.redis.host=localhost
spring.redis.port=6379 # Caffeine spring.cache.type=caffeine
spring.cache.caffeine.spec=maximumSize=500,expireAfterAccess=600s # Hazelcast
spring.cache.type=hazelcast
```

Cache annotations: @Cacheable (cache result), @CachePut (update cache), @CacheEvict (remove from cache), @Caching (multiple cache operations).

40. How do you do distributed tracing or monitoring in Spring Boot microservices?

Distributed tracing tracks requests across multiple microservices:

```
<!-- Spring Cloud Sleuth for tracing --> <dependency> <groupId>org.springframework.cloud</groupId> <artifactId>spring-cloud-starter-sleuth</artifactId> </dependency> <!-- Zipkin for visualization --> <dependency> <groupId>org.springframework.cloud</groupId> <artifactId>spring-cloud-sleuth-zipkin</artifactId> </dependency> # application.properties spring.sleuth.sampler.probability=1.0 spring.zipkin.base-url=http://localhost:9411 # Logs will include trace and span IDs 2024-01-15 10:30:45 INFO [service-name,trace-id,span-id,true] ...
```

Alternative: OpenTelemetry

```
<dependency> <groupId>io.opentelemetry</groupId> <artifactId>opentelemetry-spring-boot-starter</artifactId> </dependency> # Monitoring stack - Prometheus: Metrics collection - Grafana: Metrics visualization - Jaeger/Zipkin: Distributed tracing - ELK Stack: Log aggregation and analysis
```

Distributed tracing provides end-to-end visibility of request flows, helping identify bottlenecks and debug issues across services.