Spring Boot Mastery: Comprehensive Learning Path (Beginner to Professional)

Table of Contents

A. Spring Boot Fundamentals

Evolution from Traditional Spring to Spring Boot

Traditional Spring Framework Challenges

The traditional Spring Framework, while powerful, came with several challenges:

1. **Complex Configuration**: XML-based configuration was verbose and difficult to maintain
2. **Multiple Dependencies**: Manual management of compatible library versions
3. **Deployment Complexity**: Multiple steps to deploy applications
4. **Boilerplate Code**: Repetitive configuration code for common scenarios

For example, a simple web application in traditional Spring required:

```
<!-- web.xml -->
<web-app>
```

```
    <servlet>
        <servlet-name>dispatcher</servlet-name>
        <servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-
class>
        <init-param>
            <param-name>contextConfigLocation</param-name>
            <param-value>/WEB-INF/spring-mvc-config.xml</param-value>
        </init-param>
        <load-on-startup>1</load-on-startup>
    </servlet>

    <servlet-mapping>
        <servlet-name>dispatcher</servlet-name>
        <url-pattern>/</url-pattern>
    </servlet-mapping>

    <listener>
        <listener-
class>org.springframework.web.context.ContextLoaderListener</listener-class>
    </listener>

    <context-param>
        <param-name>contextConfigLocation</param-name>
        <param-value>/WEB-INF/applicationContext.xml</param-value>
    </context-param>
</web-app>
```

```
<!-- applicationContext.xml -->
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:context="http://www.springframework.org/schema/context"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
                           http://www.springframework.org/schema/beans/spring-
beans.xsd
                           http://www.springframework.org/schema/context
                           http://www.springframework.org/schema/context/spring-
context.xsd">

    <context:component-scan base-package="com.example" />
    <!-- Additional beans and configuration -->
</beans>
```

Spring Boot's Solution

Spring Boot was created to address these challenges. It introduced:

1. **Convention over Configuration**: Sensible defaults based on the classpath
2. **Opinionated Approach**: Pre-configured templates for common use cases
3. **Standalone Applications**: Embedded servers and self-contained JARs

4. **Production-Ready Features**: Metrics, health checks, and externalized configuration

The same web application with Spring Boot:

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

Key Milestones in Spring Evolution

- **2002**: Spring Framework 1.0 released, focusing on dependency injection and AOP
- **2006**: Spring 2.0 introduced custom XML namespaces, simplifying configuration
- **2009**: Spring 3.0 added Java-based configuration and RESTful services
- **2013**: Spring 4.0 enhanced Java 8 support and WebSocket capabilities
- **2014**: Spring Boot 1.0 released, revolutionizing Spring development
- **2017**: Spring 5.0 introduced reactive programming model
- **2018**: Spring Boot 2.0 released with major improvements
- **2022**: Spring Boot 3.0 with native compilation support via GraalVM
- **2023**: Spring Boot 3.2 with improved AOT processing and virtual threads support

Core Principles and Advantages

Spring Boot is built on several core principles that drive its design and functionality:

1. Convention over Configuration

This principle reduces the decisions developers need to make without losing flexibility.

Example: If you add a database dependency, Spring Boot automatically configures a datasource, transaction manager, and JPA repositories with sensible defaults.

2. Opinionated Defaults

Spring Boot provides pre-selected, compatible technology stacks that work well together.

Example: When you include `spring-boot-starter-web`, it automatically configures Tomcat as the embedded server, JSON processing with Jackson, and Spring MVC with sensible defaults.

3. Standalone Operation

Spring Boot applications are self-contained and can run without external dependencies.

Example: The embedded server approach means no need to deploy WAR files to external servers:

```java
@SpringBootApplication
public class Application {
    public static void main(String[] args) {
        // This single line bootstraps the entire application
        SpringApplication.run(Application.class, args);
        // No need for server configuration, deployment descriptors, etc.
    }
}
```

4. Production-Ready Features

Spring Boot includes non-functional requirements like monitoring, health checks, and externalized configuration out of the box.

Example: Adding Actuator dependency adds production-ready monitoring endpoints:

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

Key Advantages of Spring Boot

1. **Rapid Development**: Start new projects in minutes instead of hours or days
2. **Simplified Configuration**: Auto-configuration and sensible defaults
3. **Microservice Ready**: Built to support modern cloud-native applications
4. **Embedded Servers**: No need for external application servers
5. **Production Monitoring**: Built-in metrics, health checks, and management
6. **Easy Testing**: Testing support at various levels of granularity
7. **Ecosystem Integration**: Seamless integration with the broader Spring ecosystem

Project Structure and Conventions

Spring Boot follows a specific project structure and naming conventions that enable its "magic."

Standard Project Structure

```
project-root/
├── src/
│   ├── main/
│   │   ├── java/
│   │   │   └── com/example/myapp/
│   │   │       ├── MyApplication.java        # Main application class
│   │   │       ├── config/                   # Configuration classes
│   │   │       ├── controller/               # Web controllers
│   │   │       ├── model/                    # Domain model classes
```

```
│   │   │       ├── repository/               # Data access components
│   │   │       └── service/                  # Business services
│   │   └── resources/
│   │       ├── static/                       # Static web resources (JS, CSS,
images)
│   │       ├── templates/                    # View templates (Thymeleaf,
FreeMarker, etc.)
│   │       ├── application.properties        # Application configuration
│   │       └── application-{profile}.properties # Profile-specific configurations
│   └── test/
│       ├── java/                             # Test classes
│       └── resources/                        # Test resources
├── pom.xml                                   # Maven project configuration
└── README.md
```

Key Conventions

1. **Main Application Class**: Located at the root package, it contains the `@SpringBootApplication` annotation and `main()` method

```java
package com.example.myapp;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;

@SpringBootApplication  // Combines @Configuration, @EnableAutoConfiguration, and
@ComponentScan
public class MyApplication {
    public static void main(String[] args) {
        // Bootstraps the application
        SpringApplication.run(MyApplication.class, args);
    }
}
```

2. **Component Scanning**: Automatically scans for Spring components in the application's package and sub-packages

```java
// This controller is automatically discovered due to component scanning
package com.example.myapp.controller;

import org.springframework.web.bind.annotation.RestController;
import org.springframework.web.bind.annotation.GetMapping;

@RestController
public class HelloController {
    @GetMapping("/hello")
    public String hello() {
        return "Hello, Spring Boot!";
```

```
    }
}
```

3. **Configuration Classes**: Java-based configuration with `@Configuration` classes instead of XML

```
package com.example.myapp.config;

import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import com.example.myapp.service.MyService;

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

4. **Properties Files**: Configuration in `application.properties` or `application.yml`

```
# application.properties
server.port=8080
spring.datasource.url=jdbc:mysql://localhost/mydb
spring.datasource.username=root
spring.datasource.password=secret
```

The `@SpringBootApplication` Annotation

This annotation combines three annotations:

1. `@Configuration`: Tags the class as a source of bean definitions
2. `@EnableAutoConfiguration`: Enables Spring Boot's auto-configuration mechanism
3. `@ComponentScan`: Scans for components in the current package and sub-packages

Layer Guidelines

- **Controllers**: Handle HTTP requests and delegate to services
- **Services**: Implement business logic and coordinate repositories
- **Repositories**: Handle data access and storage
- **Models/Entities**: Represent business domain objects

Dependency Management and Starter Dependencies

Spring Boot revolutionizes dependency management through:

1. **Version Management**: Compatible versions without explicit specification

2. **Starter Dependencies**: Curated dependencies for common use cases
3. **Transitivity**: Automatic inclusion of necessary dependencies

Parent POM Inheritance

Spring Boot applications typically inherit from the spring-boot-starter-parent:

```
<parent>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-parent</artifactId>
    <version>3.2.0</version>
</parent>
```

This provides:

- Dependency management for Spring libraries
- Default plugin configuration
- Java version defaults
- Resources filtering
- Default configuration for various plugins

Starter Dependencies

Starters are curated sets of dependencies for specific functionality. They follow the naming convention `spring-boot-starter-*`:

```
<!-- For building web applications, including RESTful applications using Spring
MVC -->
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-web</artifactId>
</dependency>

<!-- For accessing data with JPA -->
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-data-jpa</artifactId>
</dependency>
```

Common Starter Dependencies

- **spring-boot-starter-web**: Spring MVC, embedded Tomcat, REST support
- **spring-boot-starter-data-jpa**: Spring Data JPA with Hibernate
- **spring-boot-starter-security**: Spring Security
- **spring-boot-starter-test**: Testing libraries including JUnit, Mockito
- **spring-boot-starter-actuator**: Production-ready features
- **spring-boot-starter-thymeleaf**: Thymeleaf templating engine

- **spring-boot-starter-webflux**: Reactive web applications

Overriding Default Dependencies

When necessary, you can override versions specified by Spring Boot:

```
<properties>
    <!-- Override Spring Boot's default MySQL connector version -->
    <mysql.version>8.0.33</mysql.version>
</properties>
```

Complete Example: Web Application with JPA

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
                             http://maven.apache.org/xsd/maven-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>

    <parent>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-parent</artifactId>
        <version>3.2.0</version>
    </parent>

    <groupId>com.example</groupId>
    <artifactId>my-spring-boot-app</artifactId>
    <version>1.0.0</version>

    <properties>
        <java.version>17</java.version>
    </properties>

    <dependencies>
        <!-- Web application starter -->
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-web</artifactId>
        </dependency>

        <!-- JPA data access -->
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-data-jpa</artifactId>
        </dependency>

        <!-- MySQL database driver -->
        <dependency>
            <groupId>com.mysql</groupId>
```

```
            <artifactId>mysql-connector-j</artifactId>
            <scope>runtime</scope>
        </dependency>

        <!-- Testing dependencies -->
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-test</artifactId>
            <scope>test</scope>
        </dependency>

        <!-- Developer tools -->
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-devtools</artifactId>
            <optional>true</optional>
        </dependency>
    </dependencies>

    <build>
        <plugins>
            <!-- Spring Boot Maven plugin -->
            <plugin>
                <groupId>org.springframework.boot</groupId>
                <artifactId>spring-boot-maven-plugin</artifactId>
            </plugin>
        </plugins>
    </build>
</project>
```

Auto-configuration

Auto-configuration is the "magic" behind Spring Boot's functionality. It automatically configures beans based on:

1. The dependencies on the classpath
2. The properties defined in application.properties/yaml
3. Existing beans in the ApplicationContext

How Auto-configuration Works

Auto-configuration uses conditional processing to decide which beans to create:

1. **@ConditionalOnClass**: Activates when specific classes are present
2. **@ConditionalOnMissingBean**: Activates when certain beans are missing
3. **@ConditionalOnProperty**: Activates based on property values
4. **@ConditionalOnWebApplication**: Activates in web applications

Behind the Scenes

The `@EnableAutoConfiguration` annotation (part of `@SpringBootApplication`) triggers the process:

1. Spring Boot scans for `META-INF/spring.factories` files in the classpath
2. It finds auto-configuration classes registered under `org.springframework.boot.autoconfigure.EnableAutoConfiguration`
3. Each auto-configuration class is evaluated based on its conditions
4. Matching classes contribute beans to the application context

Example of an auto-configuration class:

```
@Configuration
@ConditionalOnClass(DataSource.class)
@ConditionalOnMissingBean(DataSource.class)
@EnableConfigurationProperties(DataSourceProperties.class)
public class DataSourceAutoConfiguration {

    @Bean
    @ConditionalOnProperty(name = "spring.datasource.jmx-enabled", havingValue =
"true")
    public DataSourceMBeanServer dataSourceMBeanServer() {
        return new DataSourceMBeanServer();
    }

    @Configuration
    @ConditionalOnClass(HikariDataSource.class)
    static class Hikari {
        @Bean
        @ConditionalOnMissingBean(DataSource.class)
        @ConditionalOnProperty(name = "spring.datasource.type",
                               havingValue = "com.zaxxer.hikari.HikariDataSource",
                               matchIfMissing = true)
        public HikariDataSource dataSource(DataSourceProperties properties) {
            // Create and configure HikariDataSource
            return properties.initializeDataSourceBuilder()
                    .type(HikariDataSource.class).build();
        }
    }

    // Other configurations for different types of DataSources
}
```

Viewing Auto-configuration Reports

Spring Boot provides ways to see what auto-configuration is being applied:

1. **Debug Logs**: Set `debug=true` in application.properties

```
# application.properties
debug=true
```

2. **Actuator Endpoint**: Use the `conditions` endpoint

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

```
# Enable the conditions endpoint
management.endpoints.web.exposure.include=conditions
```

Then access `http://localhost:8080/actuator/conditions`

Customizing Auto-configuration

You can customize or override auto-configuration in several ways:

1. **Property Configuration**: Customize behavior through properties

```
# Customize embedded server port
server.port=9000

# Customize database connection pool
spring.datasource.hikari.maximum-pool-size=20
spring.datasource.hikari.minimum-idle=5
```

2. **Explicit Bean Definitions**: Override auto-configured beans

```
@Configuration
public class MyDataSourceConfig {

    @Bean
    // This will replace the auto-configured DataSource
    public DataSource dataSource() {
        return new HikariDataSource();
    }
}
```

3. **Exclude Auto-configuration Classes**: Completely disable specific auto-configurations

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

```
    }
}
```

Common Auto-configurations

- **DataSourceAutoConfiguration**: Configures database connections
- **JpaAutoConfiguration**: Sets up JPA EntityManagerFactory and repositories
- **WebMvcAutoConfiguration**: Configures Spring MVC
- **SecurityAutoConfiguration**: Sets up Spring Security
- **JacksonAutoConfiguration**: Configures JSON serialization/deserialization

Practical Example: Web Application with Database

Simply by adding the right dependencies, Spring Boot auto-configures:

```
<dependencies>
    <!-- Web starter triggers WebMvcAutoConfiguration -->
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-web</artifactId>
    </dependency>

    <!-- JPA starter triggers JpaAutoConfiguration -->
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-data-jpa</artifactId>
    </dependency>

    <!-- H2 database triggers H2ConsoleAutoConfiguration -->
    <dependency>
        <groupId>com.h2database</groupId>
        <artifactId>h2</artifactId>
        <scope>runtime</scope>
    </dependency>
</dependencies>
```

With just these dependencies and without additional configuration:

- Embedded Tomcat server is configured and started
- Spring MVC is set up with sensible defaults
- An in-memory H2 database is created
- JPA with Hibernate is configured
- Transaction management is enabled

Application Properties and YAML Configuration

Spring Boot offers flexible configuration through properties files and YAML.

Property Sources Priority

Spring Boot loads properties from multiple sources with a well-defined precedence:

1. Command-line arguments
2. JNDI attributes from `java:comp/env`
3. Java System properties (`System.getProperties()`)
4. OS environment variables
5. Profile-specific properties (`application-{profile}.properties`)
6. Application properties (`application.properties` or YAML)
7. `@PropertySource` annotations
8. Default properties

Properties File Configuration

The standard `application.properties` file:

```
# Server configuration
server.port=8080
server.servlet.context-path=/api

# Database configuration
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

# JPA/Hibernate configuration
spring.jpa.hibernate.ddl-auto=update
spring.jpa.show-sql=true
spring.jpa.properties.hibernate.format_sql=true
spring.jpa.properties.hibernate.dialect=org.hibernate.dialect.MySQLDialect

# Logging configuration
logging.level.org.springframework=INFO
logging.level.com.example=DEBUG

# Custom application properties
app.feature.enabled=true
app.max-upload-size=10MB
app.security.allowed-origins=https://example.com,https://admin.example.com
```

YAML Configuration

Equivalent configuration using `application.yml`:

```
server:
  port: 8080
  servlet:
```

```
    context-path: /api

spring:
  datasource:
    url: jdbc:mysql://localhost:3306/mydb
    username: root
    password: password
    driver-class-name: com.mysql.cj.jdbc.Driver
  jpa:
    hibernate:
      ddl-auto: update
    show-sql: true
    properties:
      hibernate:
        format_sql: true
        dialect: org.hibernate.dialect.MySQLDialect

logging:
  level:
    org.springframework: INFO
    com.example: DEBUG

app:
  feature:
    enabled: true
  max-upload-size: 10MB
  security:
    allowed-origins:
      - https://example.com
      - https://admin.example.com
```

YAML offers advantages for hierarchical configuration and list handling.

Profile-specific Configuration

Spring Boot supports different configurations for different environments using profiles:

```
# application-dev.properties
server.port=8080
spring.datasource.url=jdbc:h2:mem:devdb
```

```
# application-prod.properties
server.port=443
spring.datasource.url=jdbc:mysql://production-db/mydb
```

Activate profiles using:

```
# application.properties
spring.profiles.active=dev
```

Or via command line:

```
java -jar myapp.jar --spring.profiles.active=prod
```

Configuration Properties Classes

For type-safe configuration, use `@ConfigurationProperties`:

```
package com.example.myapp.config;

import org.springframework.boot.context.properties.ConfigurationProperties;
import org.springframework.stereotype.Component;

import java.util.List;

@Component
@ConfigurationProperties(prefix = "app.security")
public class SecurityProperties {

    private List<String> allowedOrigins;
    private int sessionTimeout = 3600; // Default value
    private boolean requireSsl = true; // Default value

    // Getters and setters
    public List<String> getAllowedOrigins() {
        return allowedOrigins;
    }

    public void setAllowedOrigins(List<String> allowedOrigins) {
        this.allowedOrigins = allowedOrigins;
    }

    public int getSessionTimeout() {
        return sessionTimeout;
    }

    public void setSessionTimeout(int sessionTimeout) {
        this.sessionTimeout = sessionTimeout;
    }

    public boolean isRequireSsl() {
        return requireSsl;
    }

    public void setRequireSsl(boolean requireSsl) {
```

```
        this.requireSsl = requireSsl;
    }
}
```

Enable validation for properties:

```
@Configuration
@EnableConfigurationProperties
@ConfigurationPropertiesScan("com.example.myapp.config")
public class AppConfig {
    // Configuration beans
}
```

Injecting Properties

1. Using `@Value`:

```
@RestController
public class MyController {

    @Value("${app.feature.enabled}")
    private boolean featureEnabled;

    @Value("${app.max-upload-size:5MB}") // Default value if property is missing
    private String maxUploadSize;

    @GetMapping("/feature-status")
    public String getFeatureStatus() {
        return "Feature is " + (featureEnabled ? "enabled" : "disabled");
    }
}
```

2. Using `@ConfigurationProperties` bean:

```
@Service
public class SecurityService {

    private final SecurityProperties securityProperties;

    public SecurityService(SecurityProperties securityProperties) {
        this.securityProperties = securityProperties;
    }

    public boolean isOriginAllowed(String origin) {
        return securityProperties.getAllowedOrigins().contains(origin);
    }
```

```
    public int getSessionTimeout() {
        return securityProperties.getSessionTimeout();
    }
}
```

Externalized Configuration

For production environments, externalize configuration:

1. **External Properties File**:

```
java -jar myapp.jar --
spring.config.location=file:/path/to/external/application.properties
```

2. **Environment Variables**:

```
export SPRING_DATASOURCE_URL=jdbc:mysql://prod-db/mydb
export SPRING_DATASOURCE_USERNAME=app_user
export SERVER_PORT=8443

java -jar myapp.jar
```

3. **Cloud Config Server**:

```
# application.properties
spring.config.import=configserver:http://config-server:8888
```

Best Practices

1. Use meaningful property names with hierarchical structure
2. Group related properties under common prefixes
3. Always provide sensible defaults
4. Use `@ConfigurationProperties` for type safety and validation
5. Document properties with metadata
6. Keep sensitive information in environment variables or secret management systems
7. Use profile-specific properties for environment-specific configuration

Spring Boot Actuator

Spring Boot Actuator adds production-ready features to monitor and manage applications.

Setting Up Actuator

Add the dependency:

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

Built-in Endpoints

Actuator provides several HTTP endpoints:

Endpoint ID	Description
`/health`	Application health information
`/info`	Application information
`/metrics`	Metrics information
`/env`	Environment properties
`/configprops`	Configuration properties
`/loggers`	Logger configuration
`/mappings`	Request mapping information
`/beans`	Spring bean list
`/shutdown`	Shutdown the application (disabled by default)

Configuring Endpoints

By default, only `/health` and `/info` are exposed over HTTP. Enable additional endpoints:

```
# Enable specific endpoints
management.endpoints.web.exposure.include=health,info,metrics,env

# Enable all endpoints
management.endpoints.web.exposure.include=*

# Exclude sensitive endpoints
management.endpoints.web.exposure.exclude=env,beans

# Configure base path (default is /actuator)
management.endpoints.web.base-path=/management
```

Health Checks

The `/health` endpoint shows application health status:

```
# Show detailed health information
management.endpoint.health.show-details=always

# Enable/disable specific health indicators
management.health.diskspace.enabled=true
management.health.db.enabled=true
```

Customize health checks by implementing `HealthIndicator`:

```
@Component
public class CustomHealthIndicator implements HealthIndicator {

    @Override
    public Health health() {
        boolean healthy = checkExternalSystem();

        if (healthy) {
            return Health.up()
                    .withDetail("externalSystem", "Available")
                    .withDetail("responseTime", "42ms")
                    .build();
        } else {
            return Health.down()
                    .withDetail("externalSystem", "Unavailable")
                    .withDetail("error", "Connection timeout")
                    .build();
        }
    }

    private boolean checkExternalSystem() {
        // Logic to check external system health
        return true;
    }
}
```

Info Endpoint

Customize the `/info` endpoint:

```
# Static info
info.app.name=My Spring Boot Application
info.app.description=Sample Spring Boot application
info.app.version=1.0.0

# Build info (requires spring-boot-maven-plugin build-info goal)
management.info.build.enabled=true
```

```
# Git info (requires git-commit-id-plugin)
management.info.git.enabled=true
```

Add custom info contributors:

```
@Component
public class CustomInfoContributor implements InfoContributor {

    @Override
    public void contribute(Builder builder) {
        builder.withDetail("environment", getEnvironmentInfo());
    }

    private Map<String, Object> getEnvironmentInfo() {
        Map<String, Object> details = new HashMap<>();
        details.put("region", "us-west");
        details.put("serverType", "production");
        details.put("deploymentTime", new Date().toString());
        return details;
    }
}
```

Metrics

Spring Boot integrates with Micrometer to collect metrics:

```
@RestController
public class ProductController {

    private final MeterRegistry meterRegistry;

    public ProductController(MeterRegistry meterRegistry) {
        this.meterRegistry = meterRegistry;

        // Create a counter
        Counter.builder("api.product.requests")
               .description("Count of product API requests")
               .register(meterRegistry);
    }

    @GetMapping("/products")
    public List<Product> getProducts() {
        // Increment the counter
        meterRegistry.counter("api.product.requests").increment();

        // Record request processing time
        Timer timer = meterRegistry.timer("api.product.request.time");
        return timer.record(() -> productService.findAll());
```

```
    }
}
```

Access metrics at `/actuator/metrics` and specific metrics at `/actuator/metrics/{name}`.

Custom Actuator Endpoints

Create custom endpoints:

```
@Component
@Endpoint(id = "features")
public class FeaturesEndpoint {

    private final Map<String, Boolean> features = new ConcurrentHashMap<>();

    public FeaturesEndpoint() {
        features.put("user-registration", true);
        features.put("premium-content", false);
        features.put("experimental-api", false);
    }

    @ReadOperation
    public Map<String, Boolean> features() {
        return features;
    }

    @ReadOperation
    public Boolean feature(@Selector String name) {
        return features.get(name);
    }

    @WriteOperation
    public void configureFeature(@Selector String name, Boolean enabled) {
        features.put(name, enabled);
    }
}
```

Securing Actuator Endpoints

Actuator endpoints contain sensitive information. Secure them:

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

```
@Configuration
public class ActuatorSecurityConfig extends WebSecurityConfigurerAdapter {

    @Override
    protected void configure(HttpSecurity http) throws Exception {
        http.requestMatcher(EndpointRequest.toAnyEndpoint())
            .authorizeRequests()
            .requestMatchers(EndpointRequest.to("health", "info")).permitAll()
            .anyRequest().hasRole("ACTUATOR")
            .and()
            .httpBasic();
    }
}
```

Production Monitoring Setup

Integrate with monitoring systems:

1. **Prometheus Integration**:

```
<dependency>
    <groupId>io.micrometer</groupId>
    <artifactId>micrometer-registry-prometheus</artifactId>
</dependency>
```

2. **Graphite Integration**:

```
<dependency>
    <groupId>io.micrometer</groupId>
    <artifactId>micrometer-registry-graphite</artifactId>
</dependency>
```

```
management.metrics.export.graphite.host=graphite.example.com
management.metrics.export.graphite.port=2004
management.metrics.export.graphite.step=1m
```

3. **Elastic Stack Integration**:

```
<dependency>
    <groupId>io.micrometer</groupId>
    <artifactId>micrometer-registry-elastic</artifactId>
</dependency>
```

```
management.metrics.export.elastic.host=http://elastic.example.com:9200
management.metrics.export.elastic.step=10s
```

Best Practices

1. Be selective about which endpoints to expose
2. Always secure sensitive endpoints
3. Use appropriate authentication for production
4. Set up monitoring dashboards with Grafana, Kibana, etc.
5. Configure alerts for critical metrics
6. Implement custom health checks for external dependencies
7. Add business-relevant metrics to track application health

B. Building REST APIs with Spring Boot

Controller Setup and Request Mapping

REST controllers in Spring Boot handle HTTP requests and produce responses.

Basic Controller Structure

```
package com.example.myapp.controller;

import org.springframework.web.bind.annotation.RestController;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.GetMapping;

@RestController                    // Combines @Controller and @ResponseBody
@RequestMapping("/api/products") // Base path for all methods in this controller
public class ProductController {

    @GetMapping                    // Handles GET requests to /api/products
    public List<Product> getAllProducts() {
        // Implementation
        return productService.findAll();
    }

    @GetMapping("/{id}")           // Handles GET requests to /api/products/{id}
    public Product getProductById(@PathVariable Long id) {
        // Implementation
        return productService.findById(id);
    }
}
```

Key Annotations

- **@RestController**: Marks the class as a controller where every method returns a domain object instead of a view
- **@RequestMapping**: Maps HTTP requests to handler methods
- **@GetMapping, @PostMapping, @PutMapping, @DeleteMapping, @PatchMapping**: Specialized shortcuts for HTTP methods

Request Mapping Examples

1. **Basic Mappings**:

```
// Method-level mapping for GET /api/users
@GetMapping("/users")
public List<User> getUsers() { /* ... */ }

// Method-level mapping for POST /api/users
@PostMapping("/users")
public User createUser(@RequestBody User user) { /* ... */ }

// Method-level mapping for GET /api/users/{id}
@GetMapping("/users/{id}")
public User getUser(@PathVariable Long id) { /* ... */ }

// Method-level mapping for PUT /api/users/{id}
@PutMapping("/users/{id}")
public User updateUser(@PathVariable Long id, @RequestBody User user) { /* ... */
}

// Method-level mapping for DELETE /api/users/{id}
@DeleteMapping("/users/{id}")
public void deleteUser(@PathVariable Long id) { /* ... */ }
```

2. **Handling Path Variables**:

```
// Single path variable
@GetMapping("/categories/{categoryId}/products")
public List<Product> getProductsByCategory(@PathVariable Long categoryId) {
    return productService.findByCategoryId(categoryId);
}

// Multiple path variables
@GetMapping("/categories/{categoryId}/products/{productId}")
public Product getProductInCategory(
        @PathVariable Long categoryId,
        @PathVariable Long productId) {
    return productService.findByIdAndCategoryId(productId, categoryId);
}

// Custom path variable name
@GetMapping("/users/{userId}")
```

```
public User getUserById(@PathVariable("userId") Long id) {
    return userService.findById(id);
}
```

3. **Request Parameters**:

```
// Required request parameters
@GetMapping("/products/search")
public List<Product> searchProducts(@RequestParam String keyword) {
    return productService.search(keyword);
}

// Optional request parameters
@GetMapping("/products")
public List<Product> getProducts(
        @RequestParam(required = false) String category,
        @RequestParam(defaultValue = "0") int page,
        @RequestParam(defaultValue = "10") int size) {

    if (category != null) {
        return productService.findByCategory(category, page, size);
    } else {
        return productService.findAll(page, size);
    }
}

// Multiple values
@GetMapping("/products/by-tags")
public List<Product> getProductsByTags(@RequestParam List<String> tags) {
    return productService.findByTags(tags);
}
```

4. **Headers and Cookies**:

```
// Header parameters
@GetMapping("/products")
public List<Product> getProductsWithHeader(
        @RequestHeader("X-API-Version") String apiVersion) {

    return productService.findAllWithVersion(apiVersion);
}

// Optional header
@GetMapping("/users/profile")
public User getUserProfile(
        @RequestHeader(value = "Authorization", required = false) String
authHeader) {

    if (authHeader != null && authHeader.startsWith("Bearer ")) {
```

```
            String token = authHeader.substring(7);
            return userService.findByToken(token);
        }
        return null;
    }

    // Cookie values
    @GetMapping("/prefs")
    public UserPreferences getUserPreferences(
            @CookieValue(value = "user-id", required = false) String userId) {

        if (userId != null) {
            return preferencesService.getByUserId(userId);
        }
        return new UserPreferences(); // Default preferences
    }
```

5. **Request Mapping with Consumes and Produces**:

```
// Specifying content types
@PostMapping(
    path = "/products",
    consumes = MediaType.APPLICATION_JSON_VALUE,          // Accepts only JSON
    produces = MediaType.APPLICATION_JSON_VALUE           // Returns JSON
)
public Product createProduct(@RequestBody Product product) {
    return productService.save(product);
}

// Supporting multiple content types
@PostMapping(
    path = "/users",
    consumes = {
        MediaType.APPLICATION_JSON_VALUE,
        MediaType.APPLICATION_XML_VALUE
    },
    produces = {
        MediaType.APPLICATION_JSON_VALUE,
        MediaType.APPLICATION_XML_VALUE
    }
)
public ResponseEntity<User> createUser(@RequestBody User user) {
    User savedUser = userService.save(user);
    return ResponseEntity.status(HttpStatus.CREATED).body(savedUser);
}
```

Using ResponseEntity for Fine-grained Control

```
@GetMapping("/{id}")
public ResponseEntity<Product> getProduct(@PathVariable Long id) {
    Optional<Product> product = productService.findById(id);

    return product
            .map(p -> ResponseEntity.ok().body(p))           // Return 200 OK with
product
            .orElse(ResponseEntity.notFound().build());      // Return 404 Not
Found
}

@PostMapping
public ResponseEntity<Product> createProduct(@RequestBody Product product) {
    Product savedProduct = productService.save(product);

    URI location = ServletUriComponentsBuilder
            .fromCurrentRequest()
            .path("/{id}")
            .buildAndExpand(savedProduct.getId())
            .toUri();

    return ResponseEntity.created(location).body(savedProduct);  // Return 201
Created
}

@DeleteMapping("/{id}")
public ResponseEntity<Void> deleteProduct(@PathVariable Long id) {
    boolean deleted = productService.deleteById(id);

    if (deleted) {
        return ResponseEntity.noContent().build();           // Return 204 No
Content
    } else {
        return ResponseEntity.notFound().build();            // Return 404 Not
Found
    }
}
```

Controller Design Best Practices

1. **Use Hierarchy for Complex APIs**:

```
// Base controller with common methods
@RestController
@RequestMapping("/api/v1/products")
public class ProductController {

    @GetMapping
    public List<Product> getAllProducts() { /* ... */ }
```

```
    @GetMapping("/{productId}")
    public Product getProduct(@PathVariable Long productId) { /* ... */ }

    // Other common methods
}

// Extended controller for specific product operations
@RestController
@RequestMapping("/api/v1/products/{productId}/reviews")
public class ProductReviewController {

    @GetMapping
    public List<Review> getProductReviews(@PathVariable Long productId) { /* ...
*/ }

    @PostMapping
    public Review addProductReview(
            @PathVariable Long productId,
            @RequestBody Review review) { /* ... */ }

    // Other review-specific methods
}
```

2. **Implementing Proper HTTP Status Codes**:

```
@PutMapping("/{id}")
public ResponseEntity<Product> updateProduct(
        @PathVariable Long id,
        @RequestBody Product product) {

    if (!id.equals(product.getId())) {
        return ResponseEntity.badRequest().build();  // 400 Bad Request
    }

    boolean exists = productService.existsById(id);
    if (!exists) {
        return ResponseEntity.notFound().build();    // 404 Not Found
    }

    try {
        Product updated = productService.update(product);
        return ResponseEntity.ok(updated);           // 200 OK
    } catch (OptimisticLockingFailureException e) {
        return ResponseEntity.status(HttpStatus.CONFLICT).build(); // 409 Conflict
    } catch (Exception e) {
        return ResponseEntity.status(HttpStatus.INTERNAL_SERVER_ERROR).build(); //
500
    }
}
```

3. **Using Method-Level Request Mapping**:

```
@RestController
@RequestMapping("/api/orders")
public class OrderController {

    // GET /api/orders
    @GetMapping
    public List<Order> getOrders() { /* ... */ }

    // GET /api/orders/{id}
    @GetMapping("/{id}")
    public Order getOrder(@PathVariable Long id) { /* ... */ }

    // POST /api/orders
    @PostMapping
    public Order createOrder(@RequestBody Order order) { /* ... */ }

    // PUT /api/orders/{id}
    @PutMapping("/{id}")
    public Order updateOrder(@PathVariable Long id, @RequestBody Order order) { /*
... */ }

    // DELETE /api/orders/{id}
    @DeleteMapping("/{id}")
    public void deleteOrder(@PathVariable Long id) { /* ... */ }

    // GET /api/orders/{id}/items
    @GetMapping("/{id}/items")
    public List<OrderItem> getOrderItems(@PathVariable Long id) { /* ... */ }

    // POST /api/orders/{id}/items
    @PostMapping("/{id}/items")
    public OrderItem addOrderItem(
            @PathVariable Long id,
            @RequestBody OrderItem item) { /* ... */ }
}
```

4. **Dependency Injection with Constructor**:

```
@RestController
@RequestMapping("/api/products")
public class ProductController {

    private final ProductService productService;

    // Constructor injection (preferred in Spring Boot)
    @Autowired  // Optional in newer Spring versions
    public ProductController(ProductService productService) {
        this.productService = productService;
```

```
    }

    // Controller methods using productService
}
```

Request/Response Handling

Spring Boot provides robust mechanisms for handling different data formats and complex request/response scenarios.

Request Body Handling

1. **Basic Request Body Mapping**:

```
@PostMapping("/users")
public User createUser(@RequestBody User user) {
    // @RequestBody automatically deserializes JSON to User object
    return userService.save(user);
}
```

2. **Handling Form Data**:

```
@PostMapping(path = "/login", consumes =
MediaType.APPLICATION_FORM_URLENCODED_VALUE)
public ResponseEntity<String> login(
        @RequestParam String username,
        @RequestParam String password) {

    boolean authenticated = authService.authenticate(username, password);
    if (authenticated) {
        return ResponseEntity.ok("Login successful");
    } else {
        return ResponseEntity.status(HttpStatus.UNAUTHORIZED).body("Invalid
credentials");
    }
}
```

3. **Working with DTOs (Data Transfer Objects)**:

```
// DTO for user creation
public class UserCreateDto {
    private String username;
    private String email;
    private String password;

    // Getters and setters
}
```

```
// Controller using DTO
@PostMapping("/users")
public ResponseEntity<UserDto> createUser(@RequestBody UserCreateDto
userCreateDto) {
    // Convert DTO to entity
    User user = new User();
    user.setUsername(userCreateDto.getUsername());
    user.setEmail(userCreateDto.getEmail());
    // Hash password before storing
    user.setPasswordHash(passwordEncoder.encode(userCreateDto.getPassword()));

    User savedUser = userService.save(user);

    // Convert entity to response DTO (without sensitive data)
    UserDto userDto = new UserDto();
    userDto.setId(savedUser.getId());
    userDto.setUsername(savedUser.getUsername());
    userDto.setEmail(savedUser.getEmail());

    return ResponseEntity.status(HttpStatus.CREATED).body(userDto);
}
```

4. **Using ModelMapper for DTO conversion**:

```
@Service
public class UserServiceImpl implements UserService {

    private final UserRepository userRepository;
    private final ModelMapper modelMapper;

    public UserServiceImpl(UserRepository userRepository, ModelMapper modelMapper)
{
        this.userRepository = userRepository;
        this.modelMapper = modelMapper;
    }

    @Override
    public UserDto createUser(UserCreateDto userCreateDto) {
        // Convert DTO to entity
        User user = modelMapper.map(userCreateDto, User.class);

        // Add additional logic (e.g., password encoding)
        user.setPasswordHash(passwordEncoder.encode(userCreateDto.getPassword()));

        // Save entity
        User savedUser = userRepository.save(user);

        // Convert entity to response DTO
        return modelMapper.map(savedUser, UserDto.class);
    }
}
```

Configure ModelMapper:

```
@Configuration
public class ModelMapperConfig {

    @Bean
    public ModelMapper modelMapper() {
        ModelMapper modelMapper = new ModelMapper();

        // Configure custom mappings
        modelMapper.typeMap(User.class, UserDto.class)
                .addMappings(mapper -> mapper.skip(UserDto::setPassword)); // Skip
sensitive fields

        return modelMapper;
    }
}
```

Response Handling

1. **Basic Response Types**:

```
// Returning an object (automatically serialized to JSON)
@GetMapping("/{id}")
public Product getProduct(@PathVariable Long id) {
    return productService.findById(id);
}

// Returning a collection
@GetMapping
public List<Product> getAllProducts() {
    return productService.findAll();
}

// Returning void (204 No Content)
@DeleteMapping("/{id}")
public void deleteProduct(@PathVariable Long id) {
    productService.deleteById(id);
}
```

2. **Using ResponseEntity for HTTP Control**:

```
@GetMapping("/{id}")
public ResponseEntity<Product> getProduct(@PathVariable Long id) {
    Optional<Product> product = productService.findById(id);
```

```
    if (product.isPresent()) {
        return ResponseEntity
                .ok()
                .eTag(String.valueOf(product.get().getVersion()))  // Add ETag
header
                .lastModified(product.get().getUpdatedAt().toEpochMilli())  // Add
Last-Modified
                .body(product.get());
    } else {
        return ResponseEntity.notFound().build();
    }
}
```

3. **Custom Response Headers**:

```
@GetMapping("/download/{fileName}")
public ResponseEntity<Resource> downloadFile(@PathVariable String fileName) {
    Resource resource = fileService.loadAsResource(fileName);

    return ResponseEntity
            .ok()
            .header(HttpHeaders.CONTENT_DISPOSITION,
                    "attachment; filename=\"" + resource.getFilename() + "\"")
            .header(HttpHeaders.CONTENT_TYPE, "application/octet-stream")
            .header(HttpHeaders.CONTENT_LENGTH,
String.valueOf(resource.contentLength()))
            .body(resource);
}
```

4. **Streaming Responses**:

```
@GetMapping(value = "/large-data", produces =
MediaType.APPLICATION_OCTET_STREAM_VALUE)
public ResponseEntity<StreamingResponseBody> getLargeData() {
    StreamingResponseBody responseBody = outputStream -> {
        dataService.writeDataTo(outputStream);
    };

    return ResponseEntity
            .ok()
            .header(HttpHeaders.CONTENT_DISPOSITION, "attachment;
filename=\"large-data.csv\"")
            .body(responseBody);
}
```

5. **Paginated Responses**:

```
@GetMapping
public Page<Product> getProducts(
        @RequestParam(defaultValue = "0") int page,
        @RequestParam(defaultValue = "10") int size,
        @RequestParam(defaultValue = "id") String sortBy,
        @RequestParam(defaultValue = "asc") String direction) {

    Sort sort = direction.equalsIgnoreCase("asc") ?
            Sort.by(sortBy).ascending() :
            Sort.by(sortBy).descending();

    Pageable pageable = PageRequest.of(page, size, sort);
    return productService.findAll(pageable);
}
```

Content Negotiation

Spring Boot supports serving different content types based on request headers or URL parameters:

1. **Configure Content Negotiation**:

```
# application.properties

# Enable content negotiation via URL parameter
spring.mvc.contentnegotiation.favor-parameter=true
spring.mvc.contentnegotiation.parameter-name=format

# Enable media types
spring.mvc.contentnegotiation.media-types.json=application/json
spring.mvc.contentnegotiation.media-types.xml=application/xml
spring.mvc.contentnegotiation.media-types.csv=text/csv
```

2. **Controller Supporting Multiple Formats**:

```
@GetMapping(
    path = "/products",
    produces = {
        MediaType.APPLICATION_JSON_VALUE,
        MediaType.APPLICATION_XML_VALUE,
        "text/csv"
    }
)
public ResponseEntity<List<Product>> getProducts() {
    List<Product> products = productService.findAll();
    return ResponseEntity.ok(products);
}
```

3. **Format-Specific Response Handling**:

```
@GetMapping(path = "/products", produces = "text/csv")
public void getProductsAsCsv(HttpServletResponse response) throws IOException {
    response.setContentType("text/csv");
    response.setHeader(HttpHeaders.CONTENT_DISPOSITION, "attachment;
filename=\"products.csv\"");

    List<Product> products = productService.findAll();

    try (CSVPrinter csvPrinter = new CSVPrinter(response.getWriter(),
CSVFormat.DEFAULT
            .withHeader("ID", "Name", "Description", "Price"))) {

        for (Product product : products) {
            csvPrinter.printRecord(
                    product.getId(),
                    product.getName(),
                    product.getDescription(),
                    product.getPrice()
            );
        }
    }
}
```

File Upload and Download

1. **File Upload**:

```
@PostMapping("/upload")
public ResponseEntity<String> uploadFile(@RequestParam("file") MultipartFile file)
{
    if (file.isEmpty()) {
        return ResponseEntity.badRequest().body("Please select a file to upload");
    }

    try {
        String fileName = fileStorageService.storeFile(file);

        String fileDownloadUri =
ServletUriComponentsBuilder.fromCurrentContextPath()
                .path("/api/files/download/")
                .path(fileName)
                .toUriString();

        return ResponseEntity.ok("File uploaded successfully: " +
fileDownloadUri);
    } catch (Exception e) {
        return ResponseEntity.status(HttpStatus.INTERNAL_SERVER_ERROR)
                .body("Could not upload the file: " + e.getMessage());
```

```
    }
}
```

2. **Multiple File Upload**:

```
@PostMapping("/upload-multiple")
public ResponseEntity<List<String>> uploadMultipleFiles(
        @RequestParam("files") MultipartFile[] files) {

    List<String> fileUrls = new ArrayList<>();

    for (MultipartFile file : files) {
        if (!file.isEmpty()) {
            try {
                String fileName = fileStorageService.storeFile(file);

                String fileDownloadUri =
ServletUriComponentsBuilder.fromCurrentContextPath()
                        .path("/api/files/download/")
                        .path(fileName)
                        .toUriString();

                fileUrls.add(fileDownloadUri);
            } catch (Exception e) {
                // Log error and continue with other files
                log.error("Failed to upload file: {}", file.getOriginalFilename(),
e);
            }
        }
    }

    return ResponseEntity.ok(fileUrls);
}
```

3. **File Download**:

```
@GetMapping("/download/{fileName:.+}")
public ResponseEntity<Resource> downloadFile(@PathVariable String fileName,
                                             HttpServletRequest request) {
    // Load file as Resource
    Resource resource = fileStorageService.loadFileAsResource(fileName);

    // Try to determine file's content type
    String contentType = null;
    try {
        contentType =
request.getServletContext().getMimeType(resource.getFile().getAbsolutePath());
    } catch (IOException ex) {
        log.warn("Could not determine file type.");
```

```
    }

    // Fallback to default content type if type could not be determined
    if (contentType == null) {
        contentType = "application/octet-stream";
    }

    return ResponseEntity.ok()
            .contentType(MediaType.parseMediaType(contentType))
            .header(HttpHeaders.CONTENT_DISPOSITION,
                    "attachment; filename=\"" + resource.getFilename() + "\"")
            .body(resource);
}
```

Advanced Response Techniques

1. **Asynchronous Processing**:

```
@GetMapping("/async-data")
public CompletableFuture<ResponseEntity<List<Product>>> getDataAsync() {
    return CompletableFuture
            .supplyAsync(() -> productService.findAll())
            .thenApply(ResponseEntity::ok);
}
```

2. **Server-Sent Events (SSE)**:

```
@GetMapping(path = "/sse-events", produces = MediaType.TEXT_EVENT_STREAM_VALUE)
public SseEmitter streamEvents() {
    SseEmitter emitter = new SseEmitter(Long.MAX_VALUE);

    // Save the emitter to a service to be used elsewhere
    sseService.addEmitter(emitter);

    // Remove emitter when it times out or completes
    emitter.onCompletion(() -> sseService.removeEmitter(emitter));
    emitter.onTimeout(() -> sseService.removeEmitter(emitter));

    return emitter;
}

// In a service that publishes events
@Service
public class NotificationService {
    private final SseService sseService;

    public NotificationService(SseService sseService) {
        this.sseService = sseService;
    }
```

```
    public void publishEvent(String eventName, Object data) {
        SseEmitter.SseEventBuilder event = SseEmitter.event()
                .name(eventName)
                .data(data)
                .id(UUID.randomUUID().toString());

        sseService.sendToAll(event);
    }
}
```

3. **Streaming JSON**:

```
@GetMapping("/stream-api")
public ResponseEntity<StreamingResponseBody> streamJson() {
    StreamingResponseBody responseBody = outputStream -> {
        JsonGenerator jsonGenerator = new JsonFactory()
                .createGenerator(outputStream, JsonEncoding.UTF8);

        try {
            jsonGenerator.writeStartArray();

            // Stream database results instead of loading all in memory
            productRepository.findAll()
                    .forEach(product -> {
                        try {
                            jsonGenerator.writeStartObject();
                            jsonGenerator.writeNumberField("id", product.getId());
                            jsonGenerator.writeStringField("name",
product.getName());
                            jsonGenerator.writeNumberField("price",
product.getPrice());
                            jsonGenerator.writeEndObject();
                            // Flush after each product to send data immediately
                            jsonGenerator.flush();
                        } catch (IOException e) {
                            throw new RuntimeException("Error writing JSON", e);
                        }
                    });

            jsonGenerator.writeEndArray();
            jsonGenerator.close();
        } catch (IOException e) {
            throw new RuntimeException("Error writing JSON", e);
        }
    };

    return ResponseEntity
            .ok()
            .header(HttpHeaders.CONTENT_TYPE, MediaType.APPLICATION_JSON_VALUE)
            .body(responseBody);
}
```

Exception Handling

Proper exception handling is crucial for building robust and user-friendly REST APIs.

Basic Exception Handling

1. **Using @ExceptionHandler in a Controller**:

```
@RestController
@RequestMapping("/api/products")
public class ProductController {

    // Regular controller methods

    @ExceptionHandler(ProductNotFoundException.class)
    public ResponseEntity<ErrorResponse> handleProductNotFound(
            ProductNotFoundException ex, WebRequest request) {

        ErrorResponse errorResponse = new ErrorResponse(
                HttpStatus.NOT_FOUND.value(),
                ex.getMessage(),
                request.getDescription(false),
                new Date()
        );

        return new ResponseEntity<>(errorResponse, HttpStatus.NOT_FOUND);
    }

    @ExceptionHandler(Exception.class)
    public ResponseEntity<ErrorResponse> handleGlobalException(
            Exception ex, WebRequest request) {

        ErrorResponse errorResponse = new ErrorResponse(
                HttpStatus.INTERNAL_SERVER_ERROR.value(),
                "An unexpected error occurred",
                request.getDescription(false),
                new Date()
        );

        return new ResponseEntity<>(errorResponse,
HttpStatus.INTERNAL_SERVER_ERROR);
    }
}
```

2. **Custom Exception Classes**:

```
public class ProductNotFoundException extends RuntimeException {
```

```
    public ProductNotFoundException(Long id) {
        super("Product not found with id: " + id);
    }
}

public class InvalidProductDataException extends RuntimeException {

    public InvalidProductDataException(String message) {
        super(message);
    }
}
```

Global Exception Handling

Instead of handling exceptions in each controller, use a global exception handler:

```
@RestControllerAdvice
public class GlobalExceptionHandler {

    private static final Logger logger =
LoggerFactory.getLogger(GlobalExceptionHandler.class);

    // Handle specific exceptions
    @ExceptionHandler(ResourceNotFoundException.class)
    public ResponseEntity<ErrorResponse> handleResourceNotFoundException(
            ResourceNotFoundException ex, WebRequest request) {

        ErrorResponse errorResponse = new ErrorResponse(
                HttpStatus.NOT_FOUND.value(),
                ex.getMessage(),
                request.getDescription(false),
                new Date()
        );

        return new ResponseEntity<>(errorResponse, HttpStatus.NOT_FOUND);
    }

    @ExceptionHandler(InvalidDataException.class)
    public ResponseEntity<ErrorResponse> handleInvalidDataException(
            InvalidDataException ex, WebRequest request) {

        ErrorResponse errorResponse = new ErrorResponse(
                HttpStatus.BAD_REQUEST.value(),
                ex.getMessage(),
                request.getDescription(false),
                new Date()
        );

        return new ResponseEntity<>(errorResponse, HttpStatus.BAD_REQUEST);
    }
```

```
    // Handle validation exceptions
    @ExceptionHandler(MethodArgumentNotValidException.class)
    public ResponseEntity<ValidationErrorResponse> handleValidationExceptions(
            MethodArgumentNotValidException ex) {

        ValidationErrorResponse response = new ValidationErrorResponse();
        response.setStatus(HttpStatus.BAD_REQUEST.value());
        response.setTimestamp(new Date());
        response.setMessage("Validation failed");

        ex.getBindingResult().getFieldErrors().forEach(error -> {
            response.addError(error.getField(), error.getDefaultMessage());
        });

        return new ResponseEntity<>(response, HttpStatus.BAD_REQUEST);
    }

    // Handle access denied exceptions
    @ExceptionHandler(AccessDeniedException.class)
    public ResponseEntity<ErrorResponse> handleAccessDeniedException(
            AccessDeniedException ex, WebRequest request) {

        ErrorResponse errorResponse = new ErrorResponse(
                HttpStatus.FORBIDDEN.value(),
                "You don't have permission to access this resource",
                request.getDescription(false),
                new Date()
        );

        return new ResponseEntity<>(errorResponse, HttpStatus.FORBIDDEN);
    }

    // Handle all other exceptions
    @ExceptionHandler(Exception.class)
    public ResponseEntity<ErrorResponse> handleGlobalException(
            Exception ex, WebRequest request) {

        // Log the error
        logger.error("Unhandled exception", ex);

        ErrorResponse errorResponse = new ErrorResponse(
                HttpStatus.INTERNAL_SERVER_ERROR.value(),
                "An unexpected error occurred. Please try again later.",
                request.getDescription(false),
                new Date()
        );

        return new ResponseEntity<>(errorResponse,
HttpStatus.INTERNAL_SERVER_ERROR);
    }
}
```

Error Response DTOs

Create standardized error response objects:

```
public class ErrorResponse {
    private int status;
    private String message;
    private String details;
    private Date timestamp;

    // Constructor, getters, and setters
}

public class ValidationErrorResponse {
    private int status;
    private String message;
    private Date timestamp;
    private Map<String, String> errors = new HashMap<>();

    public void addError(String field, String message) {
        errors.put(field, message);
    }

    // Getters and setters
}
```

Exception Handling with Spring Boot's Error Handling

Configure Spring Boot's default error handling:

```
@Configuration
public class ErrorHandlingConfig {

    @Bean
    public ErrorAttributes errorAttributes() {
        return new DefaultErrorAttributes() {
            @Override
            public Map<String, Object> getErrorAttributes(
                    WebRequest webRequest, ErrorAttributeOptions options) {

                Map<String, Object> errorAttributes =
super.getErrorAttributes(webRequest, options);

                // Add or remove attributes from the error response
                errorAttributes.put("app", "MyApp");
                errorAttributes.remove("trace");  // Remove stack trace in
production

                return errorAttributes;
            }
```

```
        };
    }
}
```

Custom Error Controller

Override Spring Boot's default error page:

```
@RestController
@RequestMapping("${server.error.path:${error.path:/error}}")
public class CustomErrorController implements ErrorController {

    private final ErrorAttributes errorAttributes;

    public CustomErrorController(ErrorAttributes errorAttributes) {
        this.errorAttributes = errorAttributes;
    }

    @RequestMapping
    public ResponseEntity<Map<String, Object>> handleError(WebRequest webRequest)
{
        Map<String, Object> attributes = getErrorAttributes(webRequest);

        HttpStatus status = HttpStatus.valueOf((Integer)
attributes.get("status"));

        return new ResponseEntity<>(attributes, status);
    }

    private Map<String, Object> getErrorAttributes(WebRequest webRequest) {
        ErrorAttributeOptions options = ErrorAttributeOptions
                .defaults()
                .including(ErrorAttributeOptions.Include.MESSAGE);

        // Don't include stack traces in production
        if (Arrays.asList(environment.getActiveProfiles()).contains("dev")) {
            options =
options.including(ErrorAttributeOptions.Include.STACK_TRACE);
        }

        return errorAttributes.getErrorAttributes(webRequest, options);
    }
}
```

Exception Handling Best Practices

1. **Use Custom Exceptions**:

```
// Base exception for all business exceptions
public abstract class BusinessException extends RuntimeException {
    private final String errorCode;

    public BusinessException(String message, String errorCode) {
        super(message);
        this.errorCode = errorCode;
    }

    public String getErrorCode() {
        return errorCode;
    }
}

// Specific business exceptions
public class OrderProcessingException extends BusinessException {
    public OrderProcessingException(String message) {
        super(message, "ORDER-001");
    }
}

public class PaymentFailedException extends BusinessException {
    public PaymentFailedException(String message) {
        super(message, "PAYMENT-001");
    }
}
```

2. **Add Contextual Information**:

```
@ExceptionHandler(BusinessException.class)
public ResponseEntity<ErrorResponse> handleBusinessException(BusinessException ex)
{
    ErrorResponse response = new ErrorResponse();
    response.setStatus(HttpStatus.BAD_REQUEST.value());
    response.setMessage(ex.getMessage());
    response.setErrorCode(ex.getErrorCode());
    response.setTimestamp(new Date());

    return new ResponseEntity<>(response, HttpStatus.BAD_REQUEST);
}
```

3. **Different Responses by Environment**:

```
@ExceptionHandler(Exception.class)
public ResponseEntity<ErrorResponse> handleGlobalException(
        Exception ex, WebRequest request, HttpServletRequest servletRequest) {

    ErrorResponse response = new ErrorResponse();
    response.setStatus(HttpStatus.INTERNAL_SERVER_ERROR.value());
```

```
        response.setTimestamp(new Date());
        response.setPath(servletRequest.getRequestURI());

        // In development, include more details
        if (Arrays.asList(environment.getActiveProfiles()).contains("dev")) {
            response.setMessage(ex.getMessage());
            response.setException(ex.getClass().getName());

            StringWriter sw = new StringWriter();
            ex.printStackTrace(new PrintWriter(sw));
            response.setTrace(sw.toString());
        } else {
            // In production, just a generic message
            response.setMessage("An unexpected error occurred. Please try again
later.");
        }

        return new ResponseEntity<>(response, HttpStatus.INTERNAL_SERVER_ERROR);
    }
```

4. **Structured Exception Hierarchy**:

```
// Base exception
public abstract class ApiException extends RuntimeException {
    public ApiException(String message) {
        super(message);
    }
}

// Client errors (4xx)
public abstract class ClientException extends ApiException {
    public ClientException(String message) {
        super(message);
    }
}

// Server errors (5xx)
public abstract class ServerException extends ApiException {
    public ServerException(String message) {
        super(message);
    }
}

// Specific client exceptions
public class ResourceNotFoundException extends ClientException {
    public ResourceNotFoundException(String resource, String id) {
        super(String.format("%s not found with id: %s", resource, id));
    }
}

public class InvalidRequestException extends ClientException {
    public InvalidRequestException(String message) {
```

```
        super(message);
    }
}

// Specific server exceptions
public class DatabaseException extends ServerException {
    public DatabaseException(String message) {
        super(message);
    }
}

public class ExternalServiceException extends ServerException {
    public ExternalServiceException(String service, String message) {
        super(String.format("Error from %s: %s", service, message));
    }
}
```

5. **Implementing Problem Details (RFC 7807)**:

```
public class ProblemDetail {
    private String type;
    private String title;
    private int status;
    private String detail;
    private String instance;
    private Map<String, Object> properties = new HashMap<>();

    // Constructors, getters, and setters

    public void addProperty(String key, Object value) {
        properties.put(key, value);
    }
}

@RestControllerAdvice
public class ProblemDetailsExceptionHandler {

    @ExceptionHandler(ResourceNotFoundException.class)
    public ResponseEntity<ProblemDetail> handleResourceNotFoundException(
            ResourceNotFoundException ex, HttpServletRequest request) {

        ProblemDetail problem = new ProblemDetail();
        problem.setType("https://api.myapp.com/errors/not-found");
        problem.setTitle("Resource Not Found");
        problem.setStatus(HttpStatus.NOT_FOUND.value());
        problem.setDetail(ex.getMessage());
        problem.setInstance(request.getRequestURI());

        return ResponseEntity
                .status(HttpStatus.NOT_FOUND)
                .contentType(MediaType.APPLICATION_PROBLEM_JSON)
                .body(problem);
```

```
    }

    // Other exception handlers
}
```

Validation and Data Binding

Data validation ensures the integrity and consistency of input data in Spring Boot applications.

Bean Validation (JSR-380)

1. **Add Validation Dependency**:

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

2. **Validating Request Bodies**:

```
public class ProductDto {

    @NotNull(message = "Product name is required")
    @Size(min = 2, max = 100, message = "Product name must be between 2 and 100
characters")
    private String name;

    @Size(max = 500, message = "Description cannot exceed 500 characters")
    private String description;

    @NotNull(message = "Price is required")
    @DecimalMin(value = "0.01", message = "Price must be greater than 0")
    private BigDecimal price;

    @NotNull(message = "Category is required")
    private Long categoryId;

    @Min(value = 0, message = "Stock quantity cannot be negative")
    private Integer stockQuantity;

    // Getters and setters
}
```

3. **Controller Validation**:

```
@RestController
@RequestMapping("/api/products")
```

```
public class ProductController {

    private final ProductService productService;

    public ProductController(ProductService productService) {
        this.productService = productService;
    }

    @PostMapping
    public ResponseEntity<ProductDto> createProduct(
            @Valid @RequestBody ProductDto productDto) {

        // If validation fails, MethodArgumentNotValidException is thrown
        // and handled by the global exception handler

        ProductDto savedProduct = productService.createProduct(productDto);
        return ResponseEntity.status(HttpStatus.CREATED).body(savedProduct);
    }

    @PutMapping("/{id}")
    public ResponseEntity<ProductDto> updateProduct(
            @PathVariable Long id,
            @Valid @RequestBody ProductDto productDto) {

        ProductDto updatedProduct = productService.updateProduct(id, productDto);
        return ResponseEntity.ok(updatedProduct);
    }
}
```

Custom Validation Annotations

1. **Creating a Custom Validator**:

```
// Custom annotation
@Documented
@Constraint(validatedBy = EmailDomainValidator.class)
@Target({ElementType.FIELD})
@Retention(RetentionPolicy.RUNTIME)
public @interface ValidEmailDomain {
    String message() default "Invalid email domain";
    Class<?>[] groups() default {};
    Class<? extends Payload>[] payload() default {};
    String[] allowedDomains() default {};
}

// Validator implementation
public class EmailDomainValidator implements ConstraintValidator<ValidEmailDomain,
String> {

    private String[] allowedDomains;
```

```
    @Override
    public void initialize(ValidEmailDomain constraintAnnotation) {
        this.allowedDomains = constraintAnnotation.allowedDomains();
    }

    @Override
    public boolean isValid(String email, ConstraintValidatorContext context) {
        if (email == null) {
            return true; // Let @NotNull handle null validation
        }

        // Simple validation - in real scenarios, use a proper email validator
first
        String[] parts = email.split("@");
        if (parts.length != 2) {
            return false;
        }

        String domain = parts[1];

        for (String allowedDomain : allowedDomains) {
            if (domain.equals(allowedDomain)) {
                return true;
            }
        }

        return false;
    }
}
```

2. **Using the Custom Validator**:

```
public class UserRegistrationDto {

    @NotBlank(message = "Name is required")
    private String name;

    @NotBlank(message = "Email is required")
    @Email(message = "Email should be valid")
    @ValidEmailDomain(
        allowedDomains = {"example.com", "company.org"},
        message = "Only corporate email addresses are allowed"
    )
    private String email;

    @NotBlank(message = "Password is required")
    @Size(min = 8, message = "Password must be at least 8 characters long")
    private String password;

    // Getters and setters
}
```

Cross-Field Validation

1. **Class-Level Validation Annotations**:

```
// Custom annotation
@Documented
@Constraint(validatedBy = PasswordMatchesValidator.class)
@Target({ElementType.TYPE})
@Retention(RetentionPolicy.RUNTIME)
public @interface PasswordMatches {
    String message() default "Passwords don't match";
    Class<?>[] groups() default {};
    Class<? extends Payload>[] payload() default {};
}

// Validator implementation
public class PasswordMatchesValidator implements
ConstraintValidator<PasswordMatches, Object> {

    @Override
    public boolean isValid(Object obj, ConstraintValidatorContext context) {
        if (obj instanceof RegisterRequest) {
            RegisterRequest request = (RegisterRequest) obj;
            return request.getPassword().equals(request.getConfirmPassword());
        }
        return true;
    }
}
```

2. **Using Class-Level Validation**:

```
@PasswordMatches(message = "Password confirmation does not match")
public class RegisterRequest {

    @NotBlank(message = "Username is required")
    private String username;

    @NotBlank(message = "Email is required")
    @Email(message = "Email should be valid")
    private String email;

    @NotBlank(message = "Password is required")
    @Size(min = 8, message = "Password must be at least 8 characters long")
    private String password;

    @NotBlank(message = "Password confirmation is required")
    private String confirmPassword;
```

```
    // Getters and setters
}
```

Method-Level Validation

Enable validation for service methods:

```
@Configuration
public class ValidationConfig {

    @Bean
    public MethodValidationPostProcessor methodValidationPostProcessor() {
        return new MethodValidationPostProcessor();
    }
}
```

```
@Service
@Validated
public class UserService {

    public User createUser(@Valid UserDto userDto) {
        // Implementation
    }

    public User updateUser(
            @NotNull(message = "User ID cannot be null") Long id,
            @Valid UserDto userDto) {
        // Implementation
    }

    @Size(min = 1, message = "At least one user must be provided")
    public List<User> findByIds(@NotEmpty(message = "IDs cannot be empty")
List<Long> ids) {
        // Implementation
    }
}
```

Programmatic Validation

Manually trigger validation:

```
@Service
public class ProductService {

    private final Validator validator;
```

```
    public ProductService(Validator validator) {
        this.validator = validator;
    }

    public Product createProductWithValidation(ProductDto productDto) {
        // Manual validation
        Set<ConstraintViolation<ProductDto>> violations =
validator.validate(productDto);

        if (!violations.isEmpty()) {
            StringBuilder sb = new StringBuilder();
            for (ConstraintViolation<ProductDto> violation : violations) {
                sb.append(violation.getPropertyPath())
                  .append(": ")
                  .append(violation.getMessage())
                  .append("; ");
            }
            throw new ValidationException(sb.toString());
        }

        // Continue with product creation
        return mapAndSave(productDto);
    }

    // Other methods
}
```

Handling Validation Errors

Custom validation error response:

```
@RestControllerAdvice
public class ValidationExceptionHandler {

    @ExceptionHandler(MethodArgumentNotValidException.class)
    public ResponseEntity<ValidationErrorResponse> handleValidationExceptions(
            MethodArgumentNotValidException ex) {

        ValidationErrorResponse response = new ValidationErrorResponse();
        response.setStatus(HttpStatus.BAD_REQUEST.value());
        response.setMessage("Validation failed");
        response.setTimestamp(new Date());

        // Field errors
        ex.getBindingResult().getFieldErrors().forEach(error -> {
            response.addFieldError(
                    error.getField(),
                    error.getRejectedValue(),
                    error.getDefaultMessage()
            );
        });
```

```
        // Global errors (class-level validation)
        ex.getBindingResult().getGlobalErrors().forEach(error -> {
            response.addGlobalError(
                    error.getObjectName(),
                    error.getDefaultMessage()
            );
        });

        return new ResponseEntity<>(response, HttpStatus.BAD_REQUEST);
    }

    @ExceptionHandler(ConstraintViolationException.class)
    public ResponseEntity<ValidationErrorResponse> handleConstraintViolation(
            ConstraintViolationException ex) {

        ValidationErrorResponse response = new ValidationErrorResponse();
        response.setStatus(HttpStatus.BAD_REQUEST.value());
        response.setMessage("Validation failed");
        response.setTimestamp(new Date());

        ex.getConstraintViolations().forEach(violation -> {
            String propertyPath = violation.getPropertyPath().toString();
            // For method parameters, remove the method name
            if (propertyPath.contains(".")) {
                propertyPath =
propertyPath.substring(propertyPath.lastIndexOf(".") + 1);
            }

            response.addFieldError(
                    propertyPath,
                    violation.getInvalidValue(),
                    violation.getMessage()
            );
        });

        return new ResponseEntity<>(response, HttpStatus.BAD_REQUEST);
    }
}
```

Data Binding and Type Conversion

Configure custom data binding:

```
@Configuration
public class WebConfig implements WebMvcConfigurer {

    @Override
    public void addFormatters(FormatterRegistry registry) {
        // Add custom converters
        registry.addConverter(new StringToEnumConverter());
```

```
        registry.addConverter(new StringToLocalDateConverter());

        // Add custom formatters
        registry.addFormatter(new PhoneNumberFormatter());
    }
}

// String to Enum converter
public class StringToEnumConverter implements Converter<String, OrderStatus> {

    @Override
    public OrderStatus convert(String source) {
        try {
            return OrderStatus.valueOf(source.toUpperCase());
        } catch (IllegalArgumentException e) {
            throw new InvalidDataException("Invalid order status: " + source);
        }
    }
}

// String to LocalDate converter
public class StringToLocalDateConverter implements Converter<String, LocalDate> {

    private final DateTimeFormatter formatter = DateTimeFormatter.ofPattern("yyyy-
MM-dd");

    @Override
    public LocalDate convert(String source) {
        try {
            return LocalDate.parse(source, formatter);
        } catch (DateTimeParseException e) {
            throw new InvalidDataException("Invalid date format. Use yyyy-MM-dd");
        }
    }
}

// Phone number formatter
public class PhoneNumberFormatter implements Formatter<PhoneNumber> {

    @Override
    public PhoneNumber parse(String text, Locale locale) {
        // Remove non-digits
        String digitsOnly = text.replaceAll("\\D", "");

        if (digitsOnly.length() < 10) {
            throw new ParseException(text, 0);
        }

        PhoneNumber phoneNumber = new PhoneNumber();
        phoneNumber.setCountryCode(digitsOnly.substring(0, digitsOnly.length() -
10));
        phoneNumber.setAreaCode(digitsOnly.substring(digitsOnly.length() - 10,
digitsOnly.length() - 7));
        phoneNumber.setPrefix(digitsOnly.substring(digitsOnly.length() - 7,
```

```
digitsOnly.length() - 4));
        phoneNumber.setLineNumber(digitsOnly.substring(digitsOnly.length() - 4));

        return phoneNumber;
    }

    @Override
    public String print(PhoneNumber phoneNumber, Locale locale) {
        return String.format("+%s (%s) %s-%s",
                phoneNumber.getCountryCode(),
                phoneNumber.getAreaCode(),
                phoneNumber.getPrefix(),
                phoneNumber.getLineNumber());
    }
}
```

Best Practices for Validation

1. **Use the Right Constraints**:

```
public class ProductDto {
    @NotBlank                    // For string not null, not empty, not just
whitespace
    @Size(max = 100)             // Constrain string length
    private String name;

    @NotNull                     // For objects that shouldn't be null
    @Min(0) @Max(1000)           // For numeric ranges
    private Integer quantity;

    @Email                       // For email format validation
    private String contactEmail;

    @Pattern(regexp = "^[A-Z]{2}\\d{9}$")  // For specific pattern matching
    private String serialNumber;

    @Past                        // For dates in the past
    private LocalDate manufacturingDate;

    @FutureOrPresent             // For dates in the present or future
    private LocalDate expiryDate;

    @Digits(integer = 6, fraction = 2) // For numeric precision
    private BigDecimal price;
}
```

2. **Validation Groups**:

```
// Define validation groups
public interface ValidationGroups {
    interface Create {}
    interface Update {}
}

// Use validation groups on DTO
public class UserDto {

    @Null(groups = ValidationGroups.Create, message = "ID must be null for
creation")
    @NotNull(groups = ValidationGroups.Update, message = "ID is required for
updates")
    private Long id;

    @NotBlank(message = "Name is required")
    private String name;

    @NotBlank(groups = ValidationGroups.Create, message = "Password is required
for creation")
    @Size(min = 8, message = "Password must be at least 8 characters")
    private String password;

    // Getters and setters
}

// Controller using validation groups
@RestController
@RequestMapping("/api/users")
public class UserController {

    @PostMapping
    public ResponseEntity<UserDto> createUser(
            @Validated(ValidationGroups.Create.class) @RequestBody UserDto
userDto) {
        // Creation logic
    }

    @PutMapping("/{id}")
    public ResponseEntity<UserDto> updateUser(
            @PathVariable Long id,
            @Validated(ValidationGroups.Update.class) @RequestBody UserDto
userDto) {
        // Update logic
    }
}
```

3. **Conditional Validation with GroupSequenceProvider**:

```
// Define the provider
public class UserGroupSequenceProvider implements
```

```
DefaultGroupSequenceProvider<User> {

    @Override
    public List<Class<?>> getValidationGroups(User user) {
        List<Class<?>> groups = new ArrayList<>();

        // Add default group
        groups.add(User.class);

        // Add premium validation if user is premium
        if (user != null && Boolean.TRUE.equals(user.getPremium())) {
            groups.add(PremiumValidation.class);
        }

        return groups;
    }
}

// Use the provider in the entity
@Entity
@GroupSequenceProvider(UserGroupSequenceProvider.class)
public class User {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    @NotBlank(message = "Name is required")
    private String name;

    private Boolean premium;

    // Only validated for premium users
    @NotNull(groups = PremiumValidation.class, message = "Premium users must have
a subscription type")
    private SubscriptionType subscriptionType;

    // Getters and setters

    // Validation group marker interface
    public interface PremiumValidation {}
}
```

REST API Documentation

Good documentation is essential for API usability. Spring Boot integrates with Swagger/OpenAPI for automatic API documentation.

OpenAPI 3.0 Integration

1. **Add OpenAPI Dependency**:

```
<dependency>
    <groupId>org.springdoc</groupId>
    <artifactId>springdoc-openapi-starter-webmvc-ui</artifactId>
    <version>2.3.0</version>
</dependency>
```

2. **Basic API Information**:

```
@Configuration
public class OpenApiConfig {

    @Bean
    public OpenAPI customOpenAPI() {
        return new OpenAPI()
                .info(new Info()
                        .title("Product Management API")
                        .description("API for managing products, categories, and
inventory")
                        .version("1.0")
                        .contact(new Contact()
                                .name("API Support")
                                .email("api@example.com")
                                .url("https://example.com/support"))
                        .license(new License()
                                .name("Apache 2.0")
                                .url("https://www.apache.org/licenses/LICENSE-
2.0")))
                .externalDocs(new ExternalDocumentation()
                        .description("API Documentation")
                        .url("https://example.com/docs"));
    }
}
```

3. **Security Schemes**:

```
@Configuration
public class OpenApiConfig {

    @Bean
    public OpenAPI customOpenAPI() {
        return new OpenAPI()
                .info(/* API info */)
                .components(new Components()
                        .addSecuritySchemes("bearer-jwt", new SecurityScheme()
                                .type(SecurityScheme.Type.HTTP)
                                .scheme("bearer")
                                .bearerFormat("JWT")
                                .description("Enter JWT token"))
                        .addSecuritySchemes("basic", new SecurityScheme()
```

```
                                .type(SecurityScheme.Type.HTTP)
                                .scheme("basic")))
                .addSecurityItem(new SecurityRequirement().addList("bearer-jwt"));
    }
}
```

4. **Document API Controllers**:

```
@RestController
@RequestMapping("/api/products")
@Tag(name = "Product Management", description = "APIs for managing products")
public class ProductController {

    private final ProductService productService;

    public ProductController(ProductService productService) {
        this.productService = productService;
    }

    @Operation(
        summary = "Get all products",
        description = "Retrieves a list of all products with optional filtering
and pagination",
        responses = {
            @ApiResponse(
                responseCode = "200",
                description = "Successfully retrieved products",
                content = @Content(
                    mediaType = "application/json",
                    array = @ArraySchema(schema = @Schema(implementation =
ProductDto.class))
                )
            ),
            @ApiResponse(
                responseCode = "401",
                description = "Unauthorized",
                content = @Content(schema = @Schema(implementation =
ErrorResponse.class))
            )
        }
    )
    @GetMapping
    public ResponseEntity<List<ProductDto>> getAllProducts(
            @Parameter(description = "Filter by category ID")
            @RequestParam(required = false) Long categoryId,

            @Parameter(description = "Page number (zero-based)")
            @RequestParam(defaultValue = "0") int page,

            @Parameter(description = "Number of items per page")
            @RequestParam(defaultValue = "10") int size) {
```

```
        List<ProductDto> products = productService.findAll(categoryId, page,
size);
        return ResponseEntity.ok(products);
    }

        @Operation(
        summary = "Get product by ID",
        description = "Retrieves a specific product by its ID",
        responses = {
            @ApiResponse(
                responseCode = "200",
                description = "Successfully retrieved product",
                content = @Content(schema = @Schema(implementation =
ProductDto.class))
            ),
            @ApiResponse(
                responseCode = "404",
                description = "Product not found",
                content = @Content(schema = @Schema(implementation =
ErrorResponse.class))
            ),
            @ApiResponse(
                responseCode = "401",
                description = "Unauthorized",
                content = @Content
            )
        }
    )
    @GetMapping("/{id}")
    public ResponseEntity<ProductDto> getProductById(
            @Parameter(description = "Product ID", required = true)
            @PathVariable Long id) {

        ProductDto product = productService.findById(id);
        return ResponseEntity.ok(product);
    }

    @Operation(
        summary = "Create new product",
        description = "Creates a new product with the provided details",
        responses = {
            @ApiResponse(
                responseCode = "201",
                description = "Product successfully created",
                content = @Content(schema = @Schema(implementation =
ProductDto.class))
            ),
            @ApiResponse(
                responseCode = "400",
                description = "Invalid product data",
                content = @Content(schema = @Schema(implementation =
ValidationErrorResponse.class))
            )
        }
```

```
    )
    @PostMapping
    public ResponseEntity<ProductDto> createProduct(
            @Parameter(description = "Product details", required = true)
            @Valid @RequestBody ProductDto productDto) {

        ProductDto createdProduct = productService.create(productDto);

        URI location = ServletUriComponentsBuilder.fromCurrentRequest()
                .path("/{id}")
                .buildAndExpand(createdProduct.getId())
                .toUri();

        return ResponseEntity.created(location).body(createdProduct);
    }
}
```

5. **Document DTOs**:

```
@Schema(description = "Product data transfer object")
public class ProductDto {

    @Schema(description = "Unique product identifier", example = "1")
    private Long id;

    @Schema(description = "Product name", example = "Smartphone X1", required =
true)
    @NotBlank(message = "Name is required")
    private String name;

    @Schema(description = "Product description", example = "Latest smartphone with
advanced features")
    private String description;

    @Schema(description = "Product price in USD", example = "999.99", required =
true)
    @NotNull(message = "Price is required")
    private BigDecimal price;

    @Schema(description = "Stock quantity", example = "100")
    private Integer stockQuantity;

    @Schema(description = "Category ID", example = "5", required = true)
    @NotNull(message = "Category is required")
    private Long categoryId;

    @Schema(description = "Product image URLs")
    private List<String> imageUrls;

    // Getters and setters
}
```

6. **Document Enumerations**:

```
@Schema(description = "Product status")
public enum ProductStatus {

    @Schema(description = "Product is available for purchase")
    AVAILABLE,

    @Schema(description = "Product is out of stock")
    OUT_OF_STOCK,

    @Schema(description = "Product has been discontinued")
    DISCONTINUED
}
```

7. **Customizing OpenAPI Configuration**:

```
@Configuration
public class OpenApiConfig {

    @Bean
    public OpenAPI customOpenAPI() {
        // Basic OpenAPI information
        return new OpenAPI()
                .info(apiInfo())
                .servers(apiServers())
                .components(securityComponents())
                .addSecurityItem(new SecurityRequirement().addList("bearer-jwt"))
                .tags(apiTags());
    }

    private Info apiInfo() {
        return new Info()
                .title("Product Management API")
                .description("RESTful API for managing products and categories")
                .version("1.0.0")
                .contact(new Contact()
                        .name("API Support")
                        .email("support@example.com")
                        .url("https://example.com/support"))
                .license(new License()
                        .name("Apache 2.0")
                        .url("https://www.apache.org/licenses/LICENSE-2.0"));
    }

    private List<Server> apiServers() {
        return List.of(
                new Server()
                        .url("https://api.example.com")
```

```
                        .description("Production server"),
                new Server()
                        .url("https://staging-api.example.com")
                        .description("Staging server"),
                new Server()
                        .url("http://localhost:8080")
                        .description("Development server")
        );
    }

    private Components securityComponents() {
        return new Components()
                .addSecuritySchemes("bearer-jwt", new SecurityScheme()
                        .type(SecurityScheme.Type.HTTP)
                        .scheme("bearer")
                        .bearerFormat("JWT")
                        .description("JWT token authentication"))
                .addSecuritySchemes("api-key", new SecurityScheme()
                        .type(SecurityScheme.Type.APIKEY)
                        .in(SecurityScheme.In.HEADER)
                        .name("X-API-KEY")
                        .description("API key authentication"));
    }

    private List<Tag> apiTags() {
        return List.of(
                new Tag().name("Product Management").description("APIs for
managing products"),
                new Tag().name("Category Management").description("APIs for
managing categories"),
                new Tag().name("Order Management").description("APIs for managing
orders"),
                new Tag().name("User Management").description("APIs for managing
users")
        );
    }
}
```

8. **Property-level Configuration**:

```
# application.properties

# OpenAPI configuration
springdoc.api-docs.path=/api-docs
springdoc.swagger-ui.path=/swagger-ui.html
springdoc.swagger-ui.operationsSorter=method
springdoc.swagger-ui.tagsSorter=alpha
springdoc.swagger-ui.filter=true
springdoc.swagger-ui.deepLinking=true
springdoc.swagger-ui.displayRequestDuration=true
springdoc.packages-to-scan=com.example.myapp.controller
springdoc.paths-to-match=/api/**
```

Generating API Documentation

1. **Access Swagger UI**:

After configuring OpenAPI, Swagger UI is available at:

- http://localhost:8080/swagger-ui.html (default)
- http://localhost:8080/swagger-ui/index.html (alternative path)

2. **Access OpenAPI JSON/YAML**:

Raw OpenAPI documentation is available at:

- http://localhost:8080/v3/api-docs (JSON format)
- http://localhost:8080/v3/api-docs.yaml (YAML format)

3. **Generating Static Documentation**:

```
<!-- Add to pom.xml -->
<plugin>
    <groupId>org.springdoc</groupId>
    <artifactId>springdoc-openapi-maven-plugin</artifactId>
    <version>1.4</version>
    <executions>
        <execution>
            <id>generate-api-docs</id>
            <phase>prepare-package</phase>
            <goals>
                <goal>generate</goal>
            </goals>
        </execution>
    </executions>
    <configuration>
        <apiDocsUrl>http://localhost:8080/v3/api-docs</apiDocsUrl>
        <outputFileName>openapi.json</outputFileName>
        <outputDir>${project.build.directory}/classes/static/docs</outputDir>
    </configuration>
</plugin>
```

Best Practices for API Documentation

1. **Use Descriptive Operation Summaries and Descriptions**:

```
@Operation(
    summary = "Update product inventory",
    description = "Updates the inventory quantity for a specific product. " +
                  "If the quantity is set to 0, the product status is
automatically " +
```

```
                    "changed to OUT_OF_STOCK. Negative values are not allowed."
)
```

2. **Document All Possible Responses**:

```
@ApiResponses({
    @ApiResponse(
        responseCode = "200",
        description = "Inventory updated successfully"
    ),
    @ApiResponse(
        responseCode = "400",
        description = "Invalid request (e.g., negative quantity)",
        content = @Content(schema = @Schema(implementation = ErrorResponse.class))
    ),
    @ApiResponse(
        responseCode = "404",
        description = "Product not found",
        content = @Content(schema = @Schema(implementation = ErrorResponse.class))
    ),
    @ApiResponse(
        responseCode = "409",
        description = "Conflict with current state (e.g., product discontinued)",
        content = @Content(schema = @Schema(implementation = ErrorResponse.class))
    )
})
```

3. **Provide Examples**:

```
@Schema(
    description = "Order creation request",
    example = """
        {
          "customerId": 123,
          "items": [
            {
              "productId": 456,
              "quantity": 2
            },
            {
              "productId": 789,
              "quantity": 1
            }
          ],
          "shippingAddress": {
            "street": "123 Main St",
            "city": "Anytown",
            "state": "CA",
            "zipCode": "12345",
```

```
            "country": "USA"
          },
          "paymentMethod": "CREDIT_CARD"
        }
        """
)
public class OrderRequest {
    // Properties, getters, and setters
}
```

4. **Use Consistent Naming and Patterns**:

- Use noun plurals for collection resources (e.g., `/products`, `/users`)
- Use noun singulars for specific resources (e.g., `/products/{id}`, `/users/{id}`)
- Use verb phrases for operations (e.g., `/orders/{id}/cancel`, `/users/verify-email`)

5. **Document Authentication Requirements**:

```
@SecurityRequirement(name = "bearer-jwt")
@Tag(name = "User Management")
@RestController
@RequestMapping("/api/users")
public class UserController {
    // Controller methods
}
```

Versioning Strategies

API versioning ensures backward compatibility while allowing for evolution. Spring Boot supports multiple versioning approaches.

URI Path Versioning

Using different URI paths for different API versions:

```
@RestController
@RequestMapping("/api/v1/products")
public class ProductControllerV1 {

    @GetMapping("/{id}")
    public ResponseEntity<ProductDtoV1> getProductV1(@PathVariable Long id) {
        ProductDtoV1 product = productService.findProductV1(id);
        return ResponseEntity.ok(product);
    }

    // Other V1 endpoints
}

@RestController
```

```
@RequestMapping("/api/v2/products")
public class ProductControllerV2 {

    @GetMapping("/{id}")
    public ResponseEntity<ProductDtoV2> getProductV2(@PathVariable Long id) {
        ProductDtoV2 product = productService.findProductV2(id);
        return ResponseEntity.ok(product);
    }

    // Other V2 endpoints
}
```

Advantages:

- Simple and explicit
- Easy to understand and document
- Good browser and cache support

Disadvantages:

- URI should ideally represent a resource, not its version
- Requires maintaining separate controllers for each version
- Can lead to code duplication

Request Parameter Versioning

Using a query parameter to specify the API version:

```
@RestController
@RequestMapping("/api/products")
public class ProductController {

    @GetMapping(value = "/{id}", params = "version=1")
    public ResponseEntity<ProductDtoV1> getProductV1(@PathVariable Long id) {
        ProductDtoV1 product = productService.findProductV1(id);
        return ResponseEntity.ok(product);
    }

    @GetMapping(value = "/{id}", params = "version=2")
    public ResponseEntity<ProductDtoV2> getProductV2(@PathVariable Long id) {
        ProductDtoV2 product = productService.findProductV2(id);
        return ResponseEntity.ok(product);
    }
}
```

Advantages:

- Keeps URI clean for representing resources
- Easy to use and test
- Good cache support if implemented correctly

Disadvantages:

- Not ideal for path-based routing
- Can complicate URI construction
- Documentation may be less clear

HTTP Header Versioning

Using a custom HTTP header to specify the version:

```
@RestController
@RequestMapping("/api/products")
public class ProductController {

    @GetMapping(value = "/{id}", headers = "X-API-Version=1")
    public ResponseEntity<ProductDtoV1> getProductV1(@PathVariable Long id) {
        ProductDtoV1 product = productService.findProductV1(id);
        return ResponseEntity.ok(product);
    }

    @GetMapping(value = "/{id}", headers = "X-API-Version=2")
    public ResponseEntity<ProductDtoV2> getProductV2(@PathVariable Long id) {
        ProductDtoV2 product = productService.findProductV2(id);
        return ResponseEntity.ok(product);
    }
}
```

Advantages:

- Keeps URI clean and resource-focused
- Follows HTTP protocol design
- Separates versioning from resource identification

Disadvantages:

- Not visible in the URI, which can make debugging harder
- May require additional client configuration
- Potential caching issues

Media Type Versioning (Content Negotiation)

Using the Accept header with a custom media type and version:

```
@RestController
@RequestMapping("/api/products")
public class ProductController {

    @GetMapping(
        value = "/{id}",
```

```
        produces = "application/vnd.company.app-v1+json"
    )
    public ResponseEntity<ProductDtoV1> getProductV1(@PathVariable Long id) {
        ProductDtoV1 product = productService.findProductV1(id);
        return ResponseEntity.ok(product);
    }

    @GetMapping(
        value = "/{id}",
        produces = "application/vnd.company.app-v2+json"
    )
    public ResponseEntity<ProductDtoV2> getProductV2(@PathVariable Long id) {
        ProductDtoV2 product = productService.findProductV2(id);
        return ResponseEntity.ok(product);
    }
}
```

Advantages:

- Follows HTTP content negotiation standards
- Clean URIs that represent resources
- Most REST-compliant approach

Disadvantages:

- More complex for API consumers
- May be difficult to test in browsers
- Potential caching complexities

Implementing Versioning with a Facade Pattern

To reduce code duplication, use a facade pattern:

```
// Common interface for all versions
public interface ProductFacade {
    ResponseEntity<?> getProduct(Long id);
    ResponseEntity<?> getAllProducts();
    ResponseEntity<?> createProduct(Object productDto);
    ResponseEntity<?> updateProduct(Long id, Object productDto);
    ResponseEntity<?> deleteProduct(Long id);
}

// V1 implementation
@Component
public class ProductFacadeV1 implements ProductFacade {

    private final ProductServiceV1 productService;

    public ProductFacadeV1(ProductServiceV1 productService) {
        this.productService = productService;
    }
```

```
    @Override
    public ResponseEntity<?> getProduct(Long id) {
        ProductDtoV1 product = productService.findById(id);
        return ResponseEntity.ok(product);
    }

    // Other method implementations
}

// V2 implementation
@Component
public class ProductFacadeV2 implements ProductFacade {

    private final ProductServiceV2 productService;

    public ProductFacadeV2(ProductServiceV2 productService) {
        this.productService = productService;
    }

    @Override
    public ResponseEntity<?> getProduct(Long id) {
        ProductDtoV2 product = productService.findById(id);
        return ResponseEntity.ok(product);
    }

    // Other method implementations
}

// Controller delegating to facades
@RestController
@RequestMapping("/api/products")
public class ProductController {

    private final ProductFacadeV1 productFacadeV1;
    private final ProductFacadeV2 productFacadeV2;

    public ProductController(
            ProductFacadeV1 productFacadeV1,
            ProductFacadeV2 productFacadeV2) {
        this.productFacadeV1 = productFacadeV1;
        this.productFacadeV2 = productFacadeV2;
    }

    @GetMapping(value = "/{id}", headers = "X-API-Version=1")
    public ResponseEntity<?> getProductV1(@PathVariable Long id) {
        return productFacadeV1.getProduct(id);
    }

    @GetMapping(value = "/{id}", headers = "X-API-Version=2")
    public ResponseEntity<?> getProductV2(@PathVariable Long id) {
        return productFacadeV2.getProduct(id);
    }
```

```
    // Other versioned endpoints
}
```

Version Mapping with Custom Annotations

Create custom annotations for cleaner code:

```
// Custom annotations for API versions
@Target({ElementType.METHOD, ElementType.TYPE})
@Retention(RetentionPolicy.RUNTIME)
@Documented
@RequestMapping(produces = "application/vnd.company.app-v1+json")
public @interface ApiV1 {
}

@Target({ElementType.METHOD, ElementType.TYPE})
@Retention(RetentionPolicy.RUNTIME)
@Documented
@RequestMapping(produces = "application/vnd.company.app-v2+json")
public @interface ApiV2 {
}

// Using the annotations in controllers
@RestController
@RequestMapping("/api/products")
public class ProductController {

    private final ProductService productService;

    public ProductController(ProductService productService) {
        this.productService = productService;
    }

    @ApiV1
    @GetMapping("/{id}")
    public ResponseEntity<ProductDtoV1> getProductV1(@PathVariable Long id) {
        ProductDtoV1 product = productService.findProductV1(id);
        return ResponseEntity.ok(product);
    }

    @ApiV2
    @GetMapping("/{id}")
    public ResponseEntity<ProductDtoV2> getProductV2(@PathVariable Long id) {
        ProductDtoV2 product = productService.findProductV2(id);
        return ResponseEntity.ok(product);
    }
}
```

Best Practices for API Versioning

1. **Choose a Consistent Strategy**: Pick one versioning approach and use it consistently across all APIs.

2. **Document Version Lifecycle**:
    - When new versions are released
    - How long previous versions will be supported
    - Deprecation notices for older versions

3. **Make Backward Compatible Changes When Possible**:
    - Adding new optional fields doesn't require a new version
    - Adding new endpoints doesn't require a new version
    - Relaxing validation rules doesn't require a new version

4. **Version for Breaking Changes Only**:
    - Removing or renaming fields
    - Changing field types
    - Adding required fields
    - Changing response structure
    - Changing error codes or formats

5. **Version Your DTOs**:
    - Keep separate DTO classes for different versions
    - Use mappers to convert between versions

```
// Version-specific DTOs
public class ProductDtoV1 {
    private Long id;
    private String name;
    private String description;
    private BigDecimal price;
    // Other V1 fields, getters, and setters
}

public class ProductDtoV2 {
    private Long id;
    private String name;
    private String description;
    private BigDecimal price;
    private String sku;            // New in V2
    private List<String> images;   // New in V2
    private Map<String, String> attributes; // New in V2
    // Other V2 fields, getters, and setters
}

// DTO mapper
@Component
public class ProductMapper {

    public ProductDtoV1 toV1Dto(Product product) {
```

```
        ProductDtoV1 dto = new ProductDtoV1();
        dto.setId(product.getId());
        dto.setName(product.getName());
        dto.setDescription(product.getDescription());
        dto.setPrice(product.getPrice());
        return dto;
    }

    public ProductDtoV2 toV2Dto(Product product) {
        ProductDtoV2 dto = new ProductDtoV2();
        dto.setId(product.getId());
        dto.setName(product.getName());
        dto.setDescription(product.getDescription());
        dto.setPrice(product.getPrice());
        dto.setSku(product.getSku());
        dto.setImages(product.getImageUrls());
        dto.setAttributes(product.getAttributes());
        return dto;
    }
}
```

HATEOAS Implementation

HATEOAS (Hypermedia as the Engine of Application State) enables clients to navigate API resources through hypermedia links.

Setting Up HATEOAS

1. **Add Dependency**:

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

2. **Basic HATEOAS Implementation**:

```
@RestController
@RequestMapping("/api/products")
public class ProductController {

    private final ProductService productService;

    public ProductController(ProductService productService) {
        this.productService = productService;
    }

    @GetMapping("/{id}")
    public EntityModel<ProductDto> getProduct(@PathVariable Long id) {
```

```
        ProductDto product = productService.findById(id);

        // Create EntityModel with links
        return EntityModel.of(product,

linkTo(methodOn(ProductController.class).getProduct(id)).withSelfRel(),

linkTo(methodOn(ProductController.class).getAllProducts()).withRel("products"),

linkTo(methodOn(CategoryController.class).getCategory(product.getCategoryId())).wi
thRel("category")
        );
    }

    @GetMapping
    public CollectionModel<EntityModel<ProductDto>> getAllProducts() {
        List<ProductDto> products = productService.findAll();

        // Create EntityModel for each product with links
        List<EntityModel<ProductDto>> productResources = products.stream()
                .map(product -> EntityModel.of(product,

linkTo(methodOn(ProductController.class).getProduct(product.getId())).withSelfRel(
),

linkTo(methodOn(CategoryController.class).getCategory(product.getCategoryId())).wi
thRel("category")
                ))
                .collect(Collectors.toList());

        // Create CollectionModel with links
        return CollectionModel.of(productResources,

linkTo(methodOn(ProductController.class).getAllProducts()).withSelfRel());
    }
}
```

Creating Custom Representations

1. **Define Resource Models**:

```
// Resource model with embedded links
public class ProductModel extends RepresentationModel<ProductModel> {

    private Long id;
    private String name;
    private String description;
    private BigDecimal price;
    private Long categoryId;

    // Constructors, getters, and setters
```

```
    public static ProductModel fromDto(ProductDto dto) {
        ProductModel model = new ProductModel();
        model.setId(dto.getId());
        model.setName(dto.getName());
        model.setDescription(dto.getDescription());
        model.setPrice(dto.getPrice());
        model.setCategoryId(dto.getCategoryId());
        return model;
    }
}
```

2. **Using Resource Assemblers**:

```
@Component
public class ProductModelAssembler implements
RepresentationModelAssembler<ProductDto, ProductModel> {

    @Override
    public ProductModel toModel(ProductDto product) {
        ProductModel model = ProductModel.fromDto(product);

        model.add(linkTo(methodOn(ProductController.class)
                .getProduct(product.getId())).withSelfRel());

        model.add(linkTo(methodOn(ProductController.class)
                .getAllProducts()).withRel("products"));

        model.add(linkTo(methodOn(CategoryController.class)
                .getCategory(product.getCategoryId())).withRel("category"));

        model.add(linkTo(methodOn(ReviewController.class)
                .getProductReviews(product.getId())).withRel("reviews"));

        return model;
    }

    @Override
    public CollectionModel<ProductModel> toCollectionModel(Iterable<? extends
ProductDto> entities) {
        CollectionModel<ProductModel> productModels = RepresentationModelAssembler
                .super.toCollectionModel(entities);

        productModels.add(linkTo(methodOn(ProductController.class)
                .getAllProducts()).withSelfRel());

        return productModels;
    }
}
```

3. **Controller with Assembler**:

```
@RestController
@RequestMapping("/api/products")
public class ProductController {

    private final ProductService productService;
    private final ProductModelAssembler assembler;

    public ProductController(
            ProductService productService,
            ProductModelAssembler assembler) {
        this.productService = productService;
        this.assembler = assembler;
    }

    @GetMapping("/{id}")
    public ProductModel getProduct(@PathVariable Long id) {
        ProductDto product = productService.findById(id);
        return assembler.toModel(product);
    }

    @GetMapping
    public CollectionModel<ProductModel> getAllProducts() {
        List<ProductDto> products = productService.findAll();
        return assembler.toCollectionModel(products);
    }
}
```

Affordances (Advanced HATEOAS)

Affordances provide hints about possible actions on resources:

```
@RestController
@RequestMapping("/api/orders")
public class OrderController {

    private final OrderService orderService;

    public OrderController(OrderService orderService) {
        this.orderService = orderService;
    }

    @GetMapping("/{id}")
    public EntityModel<OrderDto> getOrder(@PathVariable Long id) {
        OrderDto order = orderService.findById(id);

        // Build links with affordances
        Link selfLink =
linkTo(methodOn(OrderController.class).getOrder(id)).withSelfRel();
```

```
        if (order.getStatus() == OrderStatus.CREATED) {
            // Add affordance for payment action
            selfLink =
selfLink.andAffordance(afford(methodOn(OrderController.class)
                    .payOrder(id, null)));

            // Add affordance for cancellation
            selfLink =
selfLink.andAffordance(afford(methodOn(OrderController.class)
                    .cancelOrder(id)));
        }

        if (order.getStatus() == OrderStatus.PAID) {
            // Add affordance for shipping
            selfLink =
selfLink.andAffordance(afford(methodOn(OrderController.class)
                    .shipOrder(id, null)));
        }

        return EntityModel.of(order, selfLink);
    }

    @PostMapping("/{id}/payment")
    public EntityModel<OrderDto> payOrder(
            @PathVariable Long id,
            @RequestBody PaymentDto paymentDto) {
        OrderDto updatedOrder = orderService.processPayment(id, paymentDto);
        return EntityModel.of(updatedOrder,

linkTo(methodOn(OrderController.class).getOrder(id)).withSelfRel());
    }

    @PostMapping("/{id}/cancel")
    public EntityModel<OrderDto> cancelOrder(@PathVariable Long id) {
        OrderDto cancelledOrder = orderService.cancelOrder(id);
        return EntityModel.of(cancelledOrder,

linkTo(methodOn(OrderController.class).getOrder(id)).withSelfRel());
    }

    @PostMapping("/{id}/ship")
    public EntityModel<OrderDto> shipOrder(
            @PathVariable Long id,
            @RequestBody ShippingDetailsDto shippingDetails) {
        OrderDto shippedOrder = orderService.shipOrder(id, shippingDetails);
        return EntityModel.of(shippedOrder,

linkTo(methodOn(OrderController.class).getOrder(id)).withSelfRel());
    }
}
```

Paging with HATEOAS

```
@RestController
@RequestMapping("/api/products")
public class ProductController {

    private final ProductService productService;
    private final PagedResourcesAssembler<ProductDto> pagedResourcesAssembler;
    private final ProductModelAssembler productAssembler;

    public ProductController(
            ProductService productService,
            PagedResourcesAssembler<ProductDto> pagedResourcesAssembler,
            ProductModelAssembler productAssembler) {
        this.productService = productService;
        this.pagedResourcesAssembler = pagedResourcesAssembler;
        this.productAssembler = productAssembler;
    }

    @GetMapping
    public PagedModel<ProductModel> getAllProducts(
            @RequestParam(defaultValue = "0") int page,
            @RequestParam(defaultValue = "10") int size,
            @RequestParam(defaultValue = "id") String sort) {

        // Create pageable request
        Pageable pageable = PageRequest.of(
                page,
                size,
                Sort.by(sort));

        // Get paged response from service
        Page<ProductDto> productPage = productService.findAll(pageable);

        // Convert to paged model with links
        return pagedResourcesAssembler.toModel(
                productPage,
                productAssembler);
    }
}
```

Best Practices for HATEOAS

1. **Consistent Link Relations**:

Use standard link relations when possible:

- `self`: Link to the current resource
- `collection`: Link to the collection containing this resource
- `create`: Link to create a new resource

- `edit` or `update`: Link to edit the current resource
- `delete`: Link to delete the current resource
- `next`, `prev`, `first`, `last`: For pagination

2. **Use Templated Links for Query Parameters**:

```
Link searchLink = Link.of(

UriTemplate.of(linkTo(methodOn(ProductController.class).searchProducts(null,
null))
                .toUri().toString(),
                Map.of(
                        "query", TemplateVariable.of("query",
TemplateVariable.VariableType.REQUEST_PARAM),
                        "category", TemplateVariable.of("category",
TemplateVariable.VariableType.REQUEST_PARAM)
                )),
        "search");

productModels.add(searchLink);
```

3. **Include State-dependent Links**:

```
@Component
public class OrderModelAssembler implements RepresentationModelAssembler<OrderDto,
EntityModel<OrderDto>> {

    @Override
    public EntityModel<OrderDto> toModel(OrderDto order) {
        // Always include self link
        List<Link> links = new ArrayList<>();

links.add(linkTo(methodOn(OrderController.class).getOrder(order.getId())).withSelf
Rel());

        // Add links based on order state
        switch (order.getStatus()) {
            case CREATED:

links.add(linkTo(methodOn(OrderController.class).cancelOrder(order.getId()))
                        .withRel("cancel"));

links.add(linkTo(methodOn(OrderController.class).payOrder(order.getId(), null))
                        .withRel("pay"));
                break;

            case PAID:

links.add(linkTo(methodOn(OrderController.class).shipOrder(order.getId(), null))
                        .withRel("ship"));
```

```
                break;

            case SHIPPED:

links.add(linkTo(methodOn(OrderController.class).deliverOrder(order.getId()))
                        .withRel("deliver"));
                break;

            case DELIVERED:

links.add(linkTo(methodOn(OrderController.class).getOrderReceipt(order.getId()))
                        .withRel("receipt"));
                break;
        }

        return EntityModel.of(order, links);
    }
}
```

4. **Include Embedded Resources When Appropriate**:

```
@Component
public class ProductModelAssembler implements
RepresentationModelAssembler<ProductDto, EntityModel<ProductDto>> {

    private final CategoryService categoryService;
    private final ReviewService reviewService;

    public ProductModelAssembler(
            CategoryService categoryService,
            ReviewService reviewService) {
        this.categoryService = categoryService;
        this.reviewService = reviewService;
    }

    @Override
    public EntityModel<ProductDto> toModel(ProductDto product) {
        EntityModel<ProductDto> model = EntityModel.of(product);

        // Add standard links

model.add(linkTo(methodOn(ProductController.class).getProduct(product.getId())).wi
thSelfRel());

model.add(linkTo(methodOn(CategoryController.class).getCategory(product.getCategor
yId())).withRel("category"));

model.add(linkTo(methodOn(ReviewController.class).getProductReviews(product.getId(
))).withRel("reviews"));

        // Add embedded resources (use sparingly and only for small, frequently
accessed data)
```

```
        // Category data is embedded for convenience
        CategoryDto category = categoryService.findById(product.getCategoryId());
        model.add(EntityModel.of(category,

linkTo(methodOn(CategoryController.class).getCategory(category.getId())).withSelfR
el())
                .withRel("_embedded.category"));

        // Top reviews are embedded (limited to avoid large responses)
        List<ReviewDto> topReviews =
reviewService.findTopReviewsByProductId(product.getId(), 2);
        List<EntityModel<ReviewDto>> reviewModels = topReviews.stream()
                .map(review -> EntityModel.of(review,

linkTo(methodOn(ReviewController.class).getReview(review.getId())).withSelfRel()))
                .collect(Collectors.toList());

        model.add(CollectionModel.of(reviewModels,

linkTo(methodOn(ReviewController.class).getProductReviews(product.getId())).withSe
lfRel())
                .withRel("_embedded.topReviews"));

        return model;
    }
}
```

C. Data Access Layer

Spring Data JPA Comprehensive Guide

Spring Data JPA simplifies data access with JPA (Java Persistence API) by reducing boilerplate code and providing powerful abstractions.

Setting Up Spring Data JPA

1. **Add Dependencies**:

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

<dependency>
    <groupId>com.h2database</groupId>
    <artifactId>h2</artifactId>
    <scope>runtime</scope>
</dependency>
```

2. **Configure Database Connection**:

```
# application.properties

# H2 Database
spring.datasource.url=jdbc:h2:mem:testdb
spring.datasource.driverClassName=org.h2.Driver
spring.datasource.username=sa
spring.datasource.password=password
spring.jpa.database-platform=org.hibernate.dialect.H2Dialect
spring.h2.console.enabled=true

# JPA/Hibernate properties
spring.jpa.show-sql=true
spring.jpa.hibernate.ddl-auto=update
spring.jpa.properties.hibernate.format_sql=true
```

Entity Mapping

1. **Basic Entity**:

```
@Entity
@Table(name = "products")
public class Product {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    @Column(nullable = false, length = 100)
    private String name;

    @Column(length = 500)
    private String description;

    @Column(nullable = false, precision = 10, scale = 2)
    private BigDecimal price;

    @Column(name = "stock_quantity")
    private Integer stockQuantity;

    @Column(name = "created_at")
    private LocalDateTime createdAt;

    @Column(name = "updated_at")
    private LocalDateTime updatedAt;

    // Getters and setters
}
```

2. **Entity Relationships**:

```
// One-to-Many relationship
@Entity
@Table(name = "categories")
public class Category {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    @Column(nullable = false, unique = true)
    private String name;

    @OneToMany(mappedBy = "category", cascade = CascadeType.ALL, orphanRemoval =
true)
    private List<Product> products = new ArrayList<>();

    // Getters and setters
}

@Entity
@Table(name = "products")
public class Product {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    // Other fields

    @ManyToOne(fetch = FetchType.LAZY)
    @JoinColumn(name = "category_id", nullable = false)
    private Category category;

    // Getters and setters
}

// Many-to-Many relationship
@Entity
@Table(name = "products")
public class Product {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    // Other fields

    @ManyToMany
    @JoinTable(
        name = "product_tag",
        joinColumns = @JoinColumn(name = "product_id"),
        inverseJoinColumns = @JoinColumn(name = "tag_id")
```

```
    )
    private Set<Tag> tags = new HashSet<>();

    // Methods to manage the relationship
    public void addTag(Tag tag) {
        tags.add(tag);
        tag.getProducts().add(this);
    }

    public void removeTag(Tag tag) {
        tags.remove(tag);
        tag.getProducts().remove(this);
    }

    // Getters and setters
}

@Entity
@Table(name = "tags")
public class Tag {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    @Column(nullable = false, unique = true)
    private String name;

    @ManyToMany(mappedBy = "tags")
    private Set<Product> products = new HashSet<>();

    // Getters and setters
}

// One-to-One relationship
@Entity
@Table(name = "users")
public class User {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    // Other fields

    @OneToOne(mappedBy = "user", cascade = CascadeType.ALL,
              fetch = FetchType.LAZY, optional = false)
    private UserProfile profile;

    // Convenience method to set both sides of the relationship
    public void setProfile(UserProfile profile) {
        if (profile == null) {
            if (this.profile != null) {
                this.profile.setUser(null);
```

```
            }
        } else {
            profile.setUser(this);
        }
        this.profile = profile;
    }

    // Getters and setters
}

@Entity
@Table(name = "user_profiles")
public class UserProfile {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    // Other fields

    @OneToOne(fetch = FetchType.LAZY)
    @JoinColumn(name = "user_id", nullable = false)
    private User user;

    // Getters and setters
}
```

3. **Inheritance Strategies**:

```
// Single Table Strategy (default)
@Entity
@Inheritance(strategy = InheritanceType.SINGLE_TABLE)
@DiscriminatorColumn(name = "payment_type", discriminatorType =
DiscriminatorType.STRING)
public abstract class Payment {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    @Column(nullable = false)
    private BigDecimal amount;

    @Column(name = "payment_date")
    private LocalDateTime paymentDate;

    // Getters and setters
}

@Entity
@DiscriminatorValue("CREDIT_CARD")
public class CreditCardPayment extends Payment {
```

```
    @Column(name = "card_number_last_four")
    private String cardNumberLastFour;

    @Column(name = "card_type")
    @Enumerated(EnumType.STRING)
    private CardType cardType;

    // Getters and setters
}

@Entity
@DiscriminatorValue("PAYPAL")
public class PayPalPayment extends Payment {

    @Column(name = "paypal_transaction_id")
    private String paypalTransactionId;

    @Column(name = "payer_email")
    private String payerEmail;

    // Getters and setters
}

// Joined Table Strategy
@Entity
@Inheritance(strategy = InheritanceType.JOINED)
public abstract class Employee {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    @Column(nullable = false)
    private String name;

    @Column(nullable = false, unique = true)
    private String email;

    // Getters and setters
}

@Entity
@Table(name = "managers")
public class Manager extends Employee {

    @Column(name = "department")
    private String department;

    @OneToMany(mappedBy = "manager")
    private Set<Developer> subordinates = new HashSet<>();

    // Getters and setters
}
```

```java
@Entity
@Table(name = "developers")
public class Developer extends Employee {

    @Column(name = "programming_language")
    private String programmingLanguage;

    @ManyToOne
    @JoinColumn(name = "manager_id")
    private Manager manager;

    // Getters and setters
}

// Table Per Class Strategy
@Entity
@Inheritance(strategy = InheritanceType.TABLE_PER_CLASS)
public abstract class Vehicle {

    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    private Long id;

    @Column(nullable = false)
    private String manufacturer;

    @Column(nullable = false)
    private String model;

    @Column(name = "production_year")
    private Integer productionYear;

    // Getters and setters
}

@Entity
@Table(name = "cars")
public class Car extends Vehicle {

    @Column(name = "num_doors")
    private Integer numDoors;

    @Column(name = "fuel_type")
    @Enumerated(EnumType.STRING)
    private FuelType fuelType;

    // Getters and setters
}

@Entity
@Table(name = "motorcycles")
public class Motorcycle extends Vehicle {
```

```
    @Column(name = "engine_capacity")
    private Integer engineCapacity;

    @Column(name = "bike_type")
    @Enumerated(EnumType.STRING)
    private BikeType bikeType;

    // Getters and setters
}
```

4. **Entity Lifecycle Callbacks**:

```
@Entity
@Table(name = "products")
@EntityListeners(AuditingEntityListener.class)
public class Product {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    // Other fields

    @CreatedDate
    @Column(name = "created_at", nullable = false, updatable = false)
    private LocalDateTime createdAt;

    @LastModifiedDate
    @Column(name = "updated_at")
    private LocalDateTime updatedAt;

    @CreatedBy
    @Column(name = "created_by")
    private String createdBy;

    @LastModifiedBy
    @Column(name = "updated_by")
    private String updatedBy;

    @Version
    private Long version;

    @PrePersist
    public void prePersist() {
        // Custom logic before saving a new entity
    }

    @PostPersist
    public void postPersist() {
        // Custom logic after saving a new entity
    }
```

```
    @PreUpdate
    public void preUpdate() {
        // Custom logic before updating an entity
    }

    @PostUpdate
    public void postUpdate() {
        // Custom logic after updating an entity
    }

    @PreRemove
    public void preRemove() {
        // Custom logic before removing an entity
    }

    @PostRemove
    public void postRemove() {
        // Custom logic after removing an entity
    }

    @PostLoad
    public void postLoad() {
        // Custom logic after loading an entity
    }

    // Getters and setters
}
```

Enable JPA Auditing:

```
@Configuration
@EnableJpaAuditing
public class JpaConfig {

    @Bean
    public AuditorAware<String> auditorProvider() {
        return () -> Optional.ofNullable(SecurityContextHolder.getContext())
                .map(SecurityContext::getAuthentication)
                .filter(Authentication::isAuthenticated)
                .map(Authentication::getName);
    }
}
```

Spring Data Repositories

1. **Basic Repository**:

```
public interface ProductRepository extends JpaRepository<Product, Long> {
    // Spring Data JPA provides basic CRUD operations out of the box
```

```
}
```

2. **Custom Query Methods**:

```
public interface ProductRepository extends JpaRepository<Product, Long> {

    // Find by field
    Optional<Product> findByName(String name);

    // Find by multiple fields
    List<Product> findByPriceBetween(BigDecimal minPrice, BigDecimal maxPrice);

    // Find with ordering
    List<Product> findByCategoryIdOrderByPriceAsc(Long categoryId);

    // Count by criteria
    long countByStockQuantityLessThan(Integer threshold);

    // Exists by criteria
    boolean existsByNameIgnoreCase(String name);

    // Custom JPQL query
    @Query("SELECT p FROM Product p WHERE p.price > :minPrice AND p.stockQuantity
> :minStock")
    List<Product> findExpensiveInStockProducts(
            @Param("minPrice") BigDecimal minPrice,
            @Param("minStock") Integer minStock);

    // Native SQL query
    @Query(
            value = "SELECT * FROM products p WHERE p.price > :minPrice ORDER BY
p.price DESC LIMIT :limit",
            nativeQuery = true
    )
    List<Product> findTopExpensiveProducts(
            @Param("minPrice") BigDecimal minPrice,
            @Param("limit") int limit);

    // Query with projection
    @Query("SELECT new com.example.dto.ProductSummary(p.id, p.name, p.price) FROM
Product p")
    List<ProductSummary> findAllProductSummaries();

    // Modifying query
    @Modifying
    @Query("UPDATE Product p SET p.stockQuantity = :newQuantity WHERE p.id = :id")
    int updateStockQuantity(@Param("id") Long id, @Param("newQuantity") Integer
newQuantity);

    // Delete by criteria
    void deleteByCreatedAtBefore(LocalDateTime date);
}
```

3. **Query By Example**:

```
@Service
public class ProductService {

    private final ProductRepository productRepository;

    public ProductService(ProductRepository productRepository) {
        this.productRepository = productRepository;
    }

    public List<Product> findProductsByCriteria(String name, String description,
BigDecimal minPrice) {
        // Create a probe entity with the search criteria
        Product probe = new Product();

        if (name != null) {
            probe.setName(name);
        }

        if (description != null) {
            probe.setDescription(description);
        }

        // Create the Example with custom matching
        ExampleMatcher matcher = ExampleMatcher.matching()
                .withIgnoreCase()
                .withStringMatcher(ExampleMatcher.StringMatcher.CONTAINING)
                .withIgnorePaths("price", "stockQuantity", "createdAt",
"updatedAt");

        Example<Product> example = Example.of(probe, matcher);

        // Find all matching the example
        List<Product> products = productRepository.findAll(example);

        // Filter by minPrice (since it can't be included in the Example)
        if (minPrice != null) {
            products = products.stream()
                    .filter(p -> p.getPrice().compareTo(minPrice) >= 0)
                    .collect(Collectors.toList());
        }

        return products;
    }
}
```

4. **Specifications**:

```
public class ProductSpecifications {

    public static Specification<Product> nameContains(String name) {
        return (root, query, cb) -> {
            if (name == null) {
                return cb.conjunction();
            }
            return cb.like(cb.lower(root.get("name")), "%" + name.toLowerCase() +
"%");
        };
    }

    public static Specification<Product> priceGreaterThan(BigDecimal price) {
        return (root, query, cb) -> {
            if (price == null) {
                return cb.conjunction();
            }
            return cb.greaterThan(root.get("price"), price);
        };
    }

    public static Specification<Product> inCategory(Long categoryId) {
        return (root, query, cb) -> {
            if (categoryId == null) {
                return cb.conjunction();
            }
            return cb.equal(root.get("category").get("id"), categoryId);
        };
    }

    public static Specification<Product> inStock() {
        return (root, query, cb) -> cb.greaterThan(root.get("stockQuantity"), 0);
    }
}

// Repository with Specification support
public interface ProductRepository extends JpaRepository<Product, Long>,
JpaSpecificationExecutor<Product> {
    // Additional custom methods
}

// Service using Specifications
@Service
public class ProductService {

    private final ProductRepository productRepository;

    public ProductService(ProductRepository productRepository) {
        this.productRepository = productRepository;
    }

    public List<Product> findProductsByAdvancedCriteria(
            String name, BigDecimal minPrice, Long categoryId, boolean
```

```
inStockOnly) {

        Specification<Product> spec = Specification.where(null);

        if (name != null) {
            spec = spec.and(ProductSpecifications.nameContains(name));
        }

        if (minPrice != null) {
            spec = spec.and(ProductSpecifications.priceGreaterThan(minPrice));
        }

        if (categoryId != null) {
            spec = spec.and(ProductSpecifications.inCategory(categoryId));
        }

        if (inStockOnly) {
            spec = spec.and(ProductSpecifications.inStock());
        }

        return productRepository.findAll(spec);
    }
}
```

5. **Custom Repository Implementation**:

```
// Custom repository interface
public interface CustomProductRepository {
    List<Product> findByCategoryWithFilters(Long categoryId, String nameFilter,
BigDecimal minPrice);
    Map<String, Long> getProductCountsByCategory();
}

// Implementation of custom methods
public class CustomProductRepositoryImpl implements CustomProductRepository {

    @PersistenceContext
    private EntityManager entityManager;

    @Override
    public List<Product> findByCategoryWithFilters(Long categoryId, String
nameFilter, BigDecimal minPrice) {
        CriteriaBuilder cb = entityManager.getCriteriaBuilder();
        CriteriaQuery<Product> query = cb.createQuery(Product.class);
        Root<Product> root = query.from(Product.class);

        // Build predicates
        List<Predicate> predicates = new ArrayList<>();

        if (categoryId != null) {
            predicates.add(cb.equal(root.get("category").get("id"), categoryId));
        }
```

```
        if (nameFilter != null && !nameFilter.trim().isEmpty()) {
            predicates.add(cb.like(cb.lower(root.get("name")),
                    "%" + nameFilter.toLowerCase() + "%"));
        }

        if (minPrice != null) {
            predicates.add(cb.greaterThanOrEqualTo(root.get("price"), minPrice));
        }

        // Add predicates to query
        query.where(predicates.toArray(new Predicate[0]));

        // Order by name
        query.orderBy(cb.asc(root.get("name")));

        return entityManager.createQuery(query).getResultList();
    }

    @Override
    public Map<String, Long> getProductCountsByCategory() {
        // Using JPQL for complex aggregation
        String jpql = "SELECT c.name, COUNT(p) " +
                      "FROM Category c LEFT JOIN c.products p " +
                      "GROUP BY c.name " +
                      "ORDER BY c.name";

        List<Object[]> results = entityManager.createQuery(jpql,
Object[].class).getResultList();

        Map<String, Long> countsByCategory = new LinkedHashMap<>();
        for (Object[] result : results) {
            String categoryName = (String) result[0];
            Long count = (Long) result[1];
            countsByCategory.put(categoryName, count);
        }

        return countsByCategory;
    }
}

// Combined repository interface
public interface ProductRepository extends JpaRepository<Product, Long>,
                                           JpaSpecificationExecutor<Product>,
                                           CustomProductRepository {
    // Standard and derived query methods
}
```

Pagination and Sorting

```java
@RestController
@RequestMapping("/api/products")
public class ProductController {

    private final ProductService productService;

    public ProductController(ProductService productService) {
        this.productService = productService;
    }

    @GetMapping
    public ResponseEntity<Page<ProductDto>> getProducts(
            @RequestParam(defaultValue = "0") int page,
            @RequestParam(defaultValue = "10") int size,
            @RequestParam(defaultValue = "id") String sort,
            @RequestParam(defaultValue = "asc") String direction) {

        // Convert direction string to Sort.Direction enum
        Sort.Direction dir = direction.equalsIgnoreCase("desc") ?
                Sort.Direction.DESC : Sort.Direction.ASC;

        // Support sorting by multiple fields (comma-separated)
        String[] sortFields = sort.split(",");
        List<Sort.Order> orders = new ArrayList<>();

        for (String field : sortFields) {
            orders.add(new Sort.Order(dir, field.trim()));
        }

        Pageable pageable = PageRequest.of(page, size, Sort.by(orders));

        Page<ProductDto> products = productService.findAll(pageable);
        return ResponseEntity.ok(products);
    }
}

@Service
public class ProductService {

    private final ProductRepository productRepository;
    private final ProductMapper productMapper;

    public ProductService(ProductRepository productRepository, ProductMapper
productMapper) {
        this.productRepository = productRepository;
        this.productMapper = productMapper;
    }

    public Page<ProductDto> findAll(Pageable pageable) {
        // Find products with pagination and sorting
        Page<Product> productPage = productRepository.findAll(pageable);

        // Map to DTOs preserving pagination information
```

```
        return productPage.map(productMapper::toDto);
    }

    public Page<ProductDto> findByCategoryId(Long categoryId, Pageable pageable) {
        Page<Product> productPage = productRepository.findByCategoryId(categoryId,
pageable);
        return productPage.map(productMapper::toDto);
    }

    public Page<ProductDto> search(String keyword, Pageable pageable) {
        Page<Product> productPage;

        if (keyword == null || keyword.trim().isEmpty()) {
            productPage = productRepository.findAll(pageable);
        } else {
            // Using Specification for dynamic search
            Specification<Product> spec = Specification
                    .where(ProductSpecifications.nameContains(keyword))
                    .or(ProductSpecifications.descriptionContains(keyword));

            productPage = productRepository.findAll(spec, pageable);
        }

        return productPage.map(productMapper::toDto);
    }
}
```

Projections and DTOs

1. **Interface-based Projections**:

```
// Closed projection (only specific properties)
public interface ProductSummary {
    Long getId();
    String getName();
    BigDecimal getPrice();
}

// Open projection (computed properties)
public interface ProductDetail {
    Long getId();
    String getName();
    BigDecimal getPrice();
    Integer getStockQuantity();

    @Value("#{target.name + ' - $' + target.price}")
    String getDisplayName();

    @Value("#{target.stockQuantity > 0 ? 'In Stock' : 'Out of Stock'}")
    String getStockStatus();
}
```

```
// Dynamic projection support in repository
public interface ProductRepository extends JpaRepository<Product, Long> {
    <T> List<T> findByCategory_Id(Long categoryId, Class<T> type);
    <T> Page<T> findByNameContaining(String name, Pageable pageable, Class<T>
type);
}

// Usage in service
@Service
public class ProductService {

    private final ProductRepository productRepository;

    public ProductService(ProductRepository productRepository) {
        this.productRepository = productRepository;
    }

    public List<ProductSummary> getProductSummariesByCategory(Long categoryId) {
        return productRepository.findByCategory_Id(categoryId,
ProductSummary.class);
    }

    public Page<ProductDetail> searchProductDetails(String name, Pageable
pageable) {
        return productRepository.findByNameContaining(name, pageable,
ProductDetail.class);
    }
}
```

2. **Class-based Projections (DTOs)**:

```
// DTO class
public class ProductDto {
    private Long id;
    private String name;
    private String description;
    private BigDecimal price;
    private Integer stockQuantity;
    private String categoryName;

    // Constructors, getters, and setters
}

// Using constructor in JPQL
@Query("SELECT new com.example.dto.ProductDto(p.id, p.name, p.description,
p.price, p.stockQuantity, p.category.name) " +
       "FROM Product p WHERE p.category.id = :categoryId")
List<ProductDto> findProductDtosByCategoryId(@Param("categoryId") Long
categoryId);

// Manual mapping with mapper
```

```
@Service
public class ProductService {

    private final ProductRepository productRepository;
    private final ProductMapper productMapper;

    public ProductService(ProductRepository productRepository, ProductMapper
productMapper) {
        this.productRepository = productRepository;
        this.productMapper = productMapper;
    }

    public List<ProductDto> findAll() {
        List<Product> products = productRepository.findAll();
        return products.stream()
                .map(productMapper::toDto)
                .collect(Collectors.toList());
    }
}

// Mapper interface using MapStruct
@Mapper(componentModel = "spring")
public interface ProductMapper {

    @Mapping(source = "category.name", target = "categoryName")
    ProductDto toDto(Product product);

    @Mapping(target = "category", ignore = true)
    Product toEntity(ProductDto dto);
}
```

Auditing and Versioning

1. **JPA Auditing**:

```
@Configuration
@EnableJpaAuditing
public class AuditingConfig {

    @Bean
    public AuditorAware<String> auditorProvider() {
        return () -> Optional.ofNullable(SecurityContextHolder.getContext())
                .map(SecurityContext::getAuthentication)
                .filter(Authentication::isAuthenticated)
                .map(Authentication::getName);
    }
}

@MappedSuperclass
@EntityListeners(AuditingEntityListener.class)
public abstract class BaseEntity {
```

```
    @CreatedBy
    @Column(name = "created_by", nullable = false, updatable = false)
    private String createdBy;

    @CreatedDate
    @Column(name = "created_date", nullable = false, updatable = false)
    private LocalDateTime createdDate;

    @LastModifiedBy
    @Column(name = "last_modified_by")
    private String lastModifiedBy;

    @LastModifiedDate
    @Column(name = "last_modified_date")
    private LocalDateTime lastModifiedDate;

    // Getters and setters
}

@Entity
@Table(name = "products")
public class Product extends BaseEntity {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    // Other fields and relationships

    @Version
    private Long version;

    // Getters and setters
}
```

2. **Custom Auditing**:

```
@Component
public class CustomAuditingHandler {

    private final AuditLogger auditLogger;

    public CustomAuditingHandler(AuditLogger auditLogger) {
        this.auditLogger = auditLogger;
    }

    @PrePersist
    public void prePersist(Object entity) {
        if (entity instanceof Auditable) {
            Auditable auditable = (Auditable) entity;
            String user = getCurrentUser();
```

```
            auditable.setCreatedBy(user);
            auditable.setCreatedDate(LocalDateTime.now());
            auditLogger.logCreation(entity.getClass().getSimpleName(),
auditable.getId(), user);
        }
    }

    @PreUpdate
    public void preUpdate(Object entity) {
        if (entity instanceof Auditable) {
            Auditable auditable = (Auditable) entity;
            String user = getCurrentUser();
            auditable.setLastModifiedBy(user);
            auditable.setLastModifiedDate(LocalDateTime.now());
            auditLogger.logUpdate(entity.getClass().getSimpleName(),
auditable.getId(), user);
        }
    }

    @PreRemove
    public void preRemove(Object entity) {
        if (entity instanceof Auditable) {
            Auditable auditable = (Auditable) entity;
            String user = getCurrentUser();
            auditLogger.logDeletion(entity.getClass().getSimpleName(),
auditable.getId(), user);
        }
    }

    private String getCurrentUser() {
        Authentication auth =
SecurityContextHolder.getContext().getAuthentication();
        return auth != null ? auth.getName() : "system";
    }
}

// Interface for entities that should be audited
public interface Auditable {
    Serializable getId();
    String getCreatedBy();
    void setCreatedBy(String createdBy);
    LocalDateTime getCreatedDate();
    void setCreatedDate(LocalDateTime createdDate);
    String getLastModifiedBy();
    void setLastModifiedBy(String lastModifiedBy);
    LocalDateTime getLastModifiedDate();
    void setLastModifiedDate(LocalDateTime lastModifiedDate);
}
```

3. **Envers Auditing**:

```
<dependency>
    <groupId>org.springframework.data</groupId>
    <artifactId>spring-data-envers</artifactId>
</dependency>
```

```
@Configuration
@EnableJpaRepositories(
    basePackages = "com.example.repository",
    repositoryFactoryBeanClass = EnversRevisionRepositoryFactoryBean.class
)
public class EnversConfig {
}

@Entity
@Table(name = "products")
@Audited
public class Product {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    @Column(nullable = false)
    private String name;

    @Column(nullable = false, precision = 10, scale = 2)
    private BigDecimal price;

    @Column(name = "stock_quantity")
    private Integer stockQuantity;

    @NotAudited  // Field that doesn't need auditing
    @Column(name = "internal_notes")
    private String internalNotes;

    // Getters and setters
}

// Revision repository interface
public interface ProductRepository extends RevisionRepository<Product, Long,
Integer>,
                                           JpaRepository<Product, Long> {
}

// Service using revision repository
@Service
public class ProductAuditService {

    private final ProductRepository productRepository;

    public ProductAuditService(ProductRepository productRepository) {
```

```
        this.productRepository = productRepository;
    }

    public List<Revision<Integer, Product>> getProductRevisions(Long productId) {
        return productRepository.findRevisions(productId).getContent();
    }

    public Product getProductAtRevision(Long productId, Integer revisionNumber) {
        return productRepository.findRevision(productId, revisionNumber)
                .orElseThrow(() -> new RevisionNotFoundException(
                        "Revision " + revisionNumber + " not found for product " +
productId));
    }

    public Product revertToPreviousRevision(Long productId) {
        // Get all revisions ordered by revision number
        List<Revision<Integer, Product>> revisions =
productRepository.findRevisions(productId)
                .getContent();

        if (revisions.size() < 2) {
            throw new IllegalStateException("Not enough revisions to revert");
        }

        // Get the previous revision
        Revision<Integer, Product> previousRevision =
revisions.get(revisions.size() - 2);
        Product previousProduct = previousRevision.getEntity();

        // Update the current product with data from the previous revision
        Product currentProduct = productRepository.findById(productId)
                .orElseThrow(() -> new ResourceNotFoundException("Product not
found"));

        // Copy attributes but keep the ID
        currentProduct.setName(previousProduct.getName());
        currentProduct.setPrice(previousProduct.getPrice());
        currentProduct.setStockQuantity(previousProduct.getStockQuantity());

        // Save the reverted product
        return productRepository.save(currentProduct);
    }
}
```

Best Practices for Spring Data JPA

1. **Use Appropriate Fetch Types**:

```
@Entity
public class Order {
```

```
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    // EAGER for small, frequently accessed collections
    @OneToMany(cascade = CascadeType.ALL, orphanRemoval = true, fetch =
FetchType.EAGER)
    @JoinColumn(name = "order_id")
    private Set<OrderItem> items = new HashSet<>();

    // LAZY for potentially large collections or rarely accessed data
    @OneToMany(mappedBy = "order", cascade = CascadeType.ALL, fetch =
FetchType.LAZY)
    private List<OrderNote> notes = new ArrayList<>();

    // LAZY for complex entities to avoid unnecessary joins
    @ManyToOne(fetch = FetchType.LAZY)
    @JoinColumn(name = "customer_id")
    private Customer customer;

    // Getters and setters
}
```

2. **Handle N+1 Query Problem with Entity Graphs**:

```
@Entity
@NamedEntityGraph(
    name = "Order.withDetails",
    attributeNodes = {
        @NamedAttributeNode("items"),
        @NamedAttributeNode(value = "customer", subgraph = "customer-details")
    },
    subgraphs = {
        @NamedSubgraph(
            name = "customer-details",
            attributeNodes = {
                @NamedAttributeNode("addresses")
            }
        )
    }
)
public class Order {
    // Entity definition
}

public interface OrderRepository extends JpaRepository<Order, Long> {

    // Use named entity graph
    @EntityGraph(value = "Order.withDetails")
    List<Order> findByOrderDateBetween(LocalDate startDate, LocalDate endDate);

    // Use dynamic entity graph
```

```
    @EntityGraph(attributePaths = {"items", "customer"})
    Optional<Order> findById(Long id);
}
```

3. **Use Native Queries Judiciously**:

```
public interface ProductRepository extends JpaRepository<Product, Long> {

    // Use JPQL when possible for database independence
    @Query("SELECT p FROM Product p WHERE p.price > (SELECT AVG(p2.price) FROM
Product p2)")
    List<Product> findProductsAboveAveragePrice();

    // Use native queries for database-specific features or performance
optimization
    @Query(
        value = "SELECT * FROM products p " +
                "JOIN product_inventory pi ON p.id = pi.product_id " +
                "WHERE pi.stock_quantity > 0 " +
                "ORDER BY p.created_at DESC " +
                "LIMIT 10",
        nativeQuery = true
    )
    List<Product> findRecentInStockProducts();
}
```

4. **Use Transactions Appropriately**:

```
@Service
@Transactional(readOnly = true)  // Default to read-only
public class OrderService {

    private final OrderRepository orderRepository;
    private final InventoryService inventoryService;
    private final PaymentService paymentService;
    private final NotificationService notificationService;

    // Constructor injection

    @Transactional  // Override to use read-write transaction
    public Order createOrder(OrderDto orderDto) {
        // Validate order
        validateOrder(orderDto);

        // Create order entity
        Order order = new Order();
        order.setCustomer(customerRepository.findById(orderDto.getCustomerId())
                .orElseThrow(() -> new ResourceNotFoundException("Customer not
found")));
```

```
        order.setOrderDate(LocalDate.now());
        order.setStatus(OrderStatus.PENDING);

        // Add order items
        for (OrderItemDto itemDto : orderDto.getItems()) {
            Product product = productRepository.findById(itemDto.getProductId())
                    .orElseThrow(() -> new ResourceNotFoundException("Product not
found"));

            OrderItem item = new OrderItem();
            item.setProduct(product);
            item.setQuantity(itemDto.getQuantity());
            item.setPrice(product.getPrice());
            order.addItem(item);

            // Update inventory (this might also be transactional)
            inventoryService.reserveStock(product.getId(), itemDto.getQuantity());
        }

        // Save order
        Order savedOrder = orderRepository.save(order);

        // Process payment (this might throw exceptions)
        paymentService.processPayment(savedOrder.getId(),
orderDto.getPaymentDetails());

        // Update order status
        savedOrder.setStatus(OrderStatus.PAID);

        // Notify customer (not part of transaction - can fail independently)
        try {
            notificationService.sendOrderConfirmation(savedOrder);
        } catch (Exception e) {
            // Log but don't roll back transaction
            log.error("Failed to send order confirmation", e);
        }

        return savedOrder;
    }

    @Transactional(timeout = 5)  // Set timeout to 5 seconds
    public Order updateOrderStatus(Long orderId, OrderStatus status) {
        Order order = orderRepository.findById(orderId)
                .orElseThrow(() -> new ResourceNotFoundException("Order not
found"));
        order.setStatus(status);
        return orderRepository.save(order);
    }

    @Transactional(noRollbackFor = InventoryWarningException.class)
    public void processBackorder(Long orderId) {
        // Process backorder logic
        try {
            inventoryService.requestBackorderedItems(orderId);
```

```
        } catch (InventoryWarningException e) {
            // Log warning but continue transaction
            log.warn("Inventory warning during backorder", e);
        }
    }
}
```

5. **Use Batch Operations for Performance**:

```
@Service
public class ProductBatchService {

    private final EntityManager entityManager;
    private final ProductRepository productRepository;

    public ProductBatchService(EntityManager entityManager, ProductRepository
productRepository) {
        this.entityManager = entityManager;
        this.productRepository = productRepository;
    }

    @Transactional
    public void batchUpdatePrices(Map<Long, BigDecimal> priceUpdates) {
        int batchSize = 50;
        int count = 0;

        for (Map.Entry<Long, BigDecimal> entry : priceUpdates.entrySet()) {
            // Use JPA directly for better control
            Product product = entityManager.find(Product.class, entry.getKey());
            if (product != null) {
                product.setPrice(entry.getValue());
                entityManager.persist(product);

                // Flush and clear in batches to avoid memory issues
                if (++count % batchSize == 0) {
                    entityManager.flush();
                    entityManager.clear();
                }
            }
        }
    }

    @Transactional
    public void batchInsertProducts(List<Product> products) {
        int batchSize = 25;
        for (int i = 0; i < products.size(); i++) {
            entityManager.persist(products.get(i));

            if ((i + 1) % batchSize == 0 || i == products.size() - 1) {
                entityManager.flush();
                entityManager.clear();
            }
```

```
        }
    }

    @Transactional
    public void batchDeleteProducts(List<Long> productIds) {
        // Using custom query for efficient batch deletion
        // Be careful with cascades and relationships
        productRepository.deleteAllByIdInBatch(productIds);
    }
}
```

6. **Use Derived Queries Wisely**:

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

    // Good: Simple and clear
    Optional<User> findByEmail(String email);

    // Good: Straightforward query with a few conditions
    List<User> findByActiveIsTrueAndCreatedDateAfter(LocalDate date);

    // Avoid: Too complex for a derived query, use @Query instead
    // List<User>
findByFirstNameContainingOrLastNameContainingAndActiveIsTrueAndRoleInAndCreatedDat
eBetweenOrderByLastLoginDateDesc(...);

    // Better: Use @Query for complex queries
    @Query("SELECT u FROM User u WHERE (LOWER(u.firstName) LIKE LOWER(CONCAT('%',
:name, '%')) " +
           "OR LOWER(u.lastName) LIKE LOWER(CONCAT('%', :name, '%'))) " +
           "AND u.active = true AND u.role IN :roles AND u.createdDate BETWEEN
:startDate AND :endDate " +
           "ORDER BY u.lastLoginDate DESC")
    List<User> findActiveUsersByNameAndRoles(
            @Param("name") String name,
            @Param("roles") List<Role> roles,
            @Param("startDate") LocalDate startDate,
            @Param("endDate") LocalDate endDate);
}
```

Working with Different Databases

Spring Boot supports various databases, including SQL and NoSQL options.

SQL Databases

1. **MySQL Configuration**:

```
<dependency>
    <groupId>com.mysql</groupId>
    <artifactId>mysql-connector-j</artifactId>
    <scope>runtime</scope>
</dependency>
```

```
# application.properties
spring.datasource.url=jdbc:mysql://localhost:3306/mydatabase?
useSSL=false&serverTimezone=UTC
spring.datasource.username=root
spring.datasource.password=password
spring.datasource.driver-class-name=com.mysql.cj.jdbc.Driver

# Hibernate properties
spring.jpa.properties.hibernate.dialect=org.hibernate.dialect.MySQLDialect
spring.jpa.hibernate.ddl-auto=update
spring.jpa.show-sql=true

# Connection pool properties
spring.datasource.hikari.maximum-pool-size=20
spring.datasource.hikari.minimum-idle=5
spring.datasource.hikari.idle-timeout=300000
spring.datasource.hikari.connection-timeout=20000
```

2. **PostgreSQL Configuration**:

```
<dependency>
    <groupId>org.postgresql</groupId>
    <artifactId>postgresql</artifactId>
    <scope>runtime</scope>
</dependency>
```

```
# application.properties
spring.datasource.url=jdbc:postgresql://localhost:5432/mydatabase
spring.datasource.username=postgres
spring.datasource.password=password
spring.datasource.driver-class-name=org.postgresql.Driver

# Hibernate properties
spring.jpa.properties.hibernate.dialect=org.hibernate.dialect.PostgreSQLDialect
spring.jpa.hibernate.ddl-auto=update
spring.jpa.show-sql=true

# PostgreSQL-specific properties
spring.jpa.properties.hibernate.jdbc.lob.non_contextual_creation=true
```

3. **Oracle Configuration**:

```
<dependency>
    <groupId>com.oracle.database.jdbc</groupId>
    <artifactId>ojdbc8</artifactId>
    <scope>runtime</scope>
</dependency>
```

```
# application.properties
spring.datasource.url=jdbc:oracle:thin:@localhost:1521:XE
spring.datasource.username=system
spring.datasource.password=password
spring.datasource.driver-class-name=oracle.jdbc.OracleDriver

# Hibernate properties
spring.jpa.properties.hibernate.dialect=org.hibernate.dialect.Oracle12cDialect
spring.jpa.hibernate.ddl-auto=update
spring.jpa.show-sql=true
```

4. **SQL Server Configuration**:

```
<dependency>
    <groupId>com.microsoft.sqlserver</groupId>
    <artifactId>mssql-jdbc</artifactId>
    <scope>runtime</scope>
</dependency>
```

```
# application.properties
spring.datasource.url=jdbc:sqlserver://localhost:1433;databaseName=mydatabase;encr
ypt=false
spring.datasource.username=sa
spring.datasource.password=password
spring.datasource.driver-class-name=com.microsoft.sqlserver.jdbc.SQLServerDriver

# Hibernate properties
spring.jpa.properties.hibernate.dialect=org.hibernate.dialect.SQLServerDialect
spring.jpa.hibernate.ddl-auto=update
spring.jpa.show-sql=true
```

5. **H2 Database (In-memory)**:

```
<dependency>
    <groupId>com.h2database</groupId>
    <artifactId>h2</artifactId>
```

```
    <scope>runtime</scope>
</dependency>
```

```
# application.properties
spring.datasource.url=jdbc:h2:mem:testdb
spring.datasource.driverClassName=org.h2.Driver
spring.datasource.username=sa
spring.datasource.password=password
spring.jpa.database-platform=org.hibernate.dialect.H2Dialect

# Enable H2 Console
spring.h2.console.enabled=true
spring.h2.console.path=/h2-console
```

NoSQL Databases

1. **MongoDB Configuration**:

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

```
# application.properties
spring.data.mongodb.host=localhost
spring.data.mongodb.port=27017
spring.data.mongodb.database=mydb
spring.data.mongodb.username=admin
spring.data.mongodb.password=password
spring.data.mongodb.authentication-database=admin
```

```
@Configuration
@EnableMongoRepositories(basePackages = "com.example.repository.mongo")
public class MongoConfig {

    @Bean
    public MongoTemplate mongoTemplate(MongoDbFactory mongoDbFactory) {
        return new MongoTemplate(mongoDbFactory);
    }

    @Bean
    public MappingMongoConverter mappingMongoConverter(
            MongoDbFactory factory, MongoMappingContext context, BeanFactory
beanFactory) {
```

```
        DbRefResolver dbRefResolver = new DefaultDbRefResolver(factory);
        MappingMongoConverter mappingConverter = new
MappingMongoConverter(dbRefResolver, context);

        // Don't save _class to mongo
        mappingConverter.setTypeMapper(new DefaultMongoTypeMapper(null));

        return mappingConverter;
    }
}

// MongoDB document
@Document(collection = "products")
public class Product {

    @Id
    private String id;

    @Indexed(unique = true)
    private String sku;

    private String name;
    private String description;
    private BigDecimal price;
    private Integer stockQuantity;

    @DBRef
    private Category category;

    @DBRef(lazy = true)
    private List<Review> reviews;

    private Map<String, String> attributes;

    @CreatedDate
    private Date createdDate;

    @LastModifiedDate
    private Date lastModifiedDate;

    // Getters and setters
}

// MongoDB repository
public interface ProductRepository extends MongoRepository<Product, String> {

    Optional<Product> findBySku(String sku);

    List<Product> findByCategory(Category category);

    @Query("{ 'price': { $lt: ?0 } }")
    List<Product> findByPriceLessThan(BigDecimal price);
```

```
    @Query("{ 'name': { $regex: ?0, $options: 'i' } }")
    List<Product> findByNameLike(String name);

    @Query(value = "{ 'category.$id': ?0 }", count = true)
    Long countByCategory(String categoryId);
}
```

2. **Redis Configuration**:

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

<!-- For connection pooling -->
<dependency>
    <groupId>org.apache.commons</groupId>
    <artifactId>commons-pool2</artifactId>
</dependency>
```

```
# application.properties
spring.redis.host=localhost
spring.redis.port=6379
spring.redis.password=password
spring.redis.database=0

# Connection pool settings
spring.redis.lettuce.pool.max-active=8
spring.redis.lettuce.pool.max-idle=8
spring.redis.lettuce.pool.min-idle=0
spring.redis.lettuce.pool.max-wait=-1ms
```

```
@Configuration
@EnableRedisRepositories
public class RedisConfig {

    @Bean
    public RedisTemplate<String, Object> redisTemplate(RedisConnectionFactory
connectionFactory) {
        RedisTemplate<String, Object> template = new RedisTemplate<>();
        template.setConnectionFactory(connectionFactory);

        // Set up serializers
        Jackson2JsonRedisSerializer<Object> jackson2JsonRedisSerializer =
                new Jackson2JsonRedisSerializer<>(Object.class);

        ObjectMapper objectMapper = new ObjectMapper();
```

```
        objectMapper.setVisibility(PropertyAccessor.ALL,
JsonAutoDetect.Visibility.ANY);
        objectMapper.activateDefaultTyping(
                LaissezFaireSubTypeValidator.instance,
                ObjectMapper.DefaultTyping.NON_FINAL,
                JsonTypeInfo.As.PROPERTY
        );

        jackson2JsonRedisSerializer.setObjectMapper(objectMapper);

        // String serializer for keys
        template.setKeySerializer(new StringRedisSerializer());
        // JSON serializer for values
        template.setValueSerializer(jackson2JsonRedisSerializer);
        // JSON serializer for hash values
        template.setHashValueSerializer(jackson2JsonRedisSerializer);

        template.afterPropertiesSet();

        return template;
    }
}

// Redis entity
@RedisHash("product")
public class Product {

    @Id
    private String id;

    @Indexed
    private String sku;

    private String name;
    private String description;
    private BigDecimal price;
    private Integer stockQuantity;

    @Reference
    private Category category;

    private Map<String, String> attributes;

    @TimeToLive
    private Long ttl; // Time-to-live in seconds

    // Getters and setters
}

// Redis repository
public interface ProductRedisRepository extends CrudRepository<Product, String> {

    Optional<Product> findBySku(String sku);
```

```
    List<Product> findByCategory(Category category);
}

// Redis operations service
@Service
public class ProductRedisService {

    private final StringRedisTemplate stringRedisTemplate;
    private final RedisTemplate<String, Object> redisTemplate;
    private final ProductRedisRepository productRepository;

    public ProductRedisService(
            StringRedisTemplate stringRedisTemplate,
            RedisTemplate<String, Object> redisTemplate,
            ProductRedisRepository productRepository) {
        this.stringRedisTemplate = stringRedisTemplate;
        this.redisTemplate = redisTemplate;
        this.productRepository = productRepository;
    }

    // Working with String keys and values
    public void incrementProductViews(String productId) {
        stringRedisTemplate.opsForValue().increment("product:views:" + productId);
    }

    public Long getProductViews(String productId) {
        String views = stringRedisTemplate.opsForValue().get("product:views:" +
productId);
        return views != null ? Long.parseLong(views) : 0;
    }

    // Working with List data structure
    public void addToRecentlyViewed(String userId, String productId) {
        String key = "user:recent:" + userId;

        // Add to start of list
        redisTemplate.opsForList().leftPush(key, productId);

        // Trim list to keep only the 10 most recent items
        redisTemplate.opsForList().trim(key, 0, 9);

        // Set expiry on key if not already set
        if (redisTemplate.getExpire(key) < 0) {
            redisTemplate.expire(key, 30, TimeUnit.DAYS);
        }
    }

    public List<String> getRecentlyViewed(String userId) {
        String key = "user:recent:" + userId;
        Long size = redisTemplate.opsForList().size(key);

        if (size == null || size == 0) {
            return Collections.emptyList();
        }
```

```
        return redisTemplate.opsForList().range(key, 0, size - 1)
                .stream()
                .map(Object::toString)
                .collect(Collectors.toList());
    }

    // Working with Hash data structure
    public void updateProductStock(String productId, int quantity) {
        String key = "product:stock";
        redisTemplate.opsForHash().put(key, productId, quantity);
    }

    public Integer getProductStock(String productId) {
        String key = "product:stock";
        Object value = redisTemplate.opsForHash().get(key, productId);
        return value != null ? (Integer) value : null;
    }

    // Working with Set data structure
    public void addProductTag(String productId, String tag) {
        String key = "product:tags:" + productId;
        redisTemplate.opsForSet().add(key, tag);
    }

    public Set<String> getProductTags(String productId) {
        String key = "product:tags:" + productId;
        Set<Object> tags = redisTemplate.opsForSet().members(key);

        if (tags == null) {
            return Collections.emptySet();
        }

        return tags.stream()
                .map(Object::toString)
                .collect(Collectors.toSet());
    }

    // Working with Sorted Set data structure
    public void addProductRating(String productId, String userId, double rating) {
        String key = "product:ratings:" + productId;
        redisTemplate.opsForZSet().add(key, userId, rating);
    }

    public Double getAverageRating(String productId) {
        String key = "product:ratings:" + productId;

        Set<Object> userIds = redisTemplate.opsForZSet().range(key, 0, -1);
        if (userIds == null || userIds.isEmpty()) {
            return 0.0;
        }

        double sum = 0;
        for (Object userId : userIds) {
```

```
            Double score = redisTemplate.opsForZSet().score(key, userId);
            if (score != null) {
                sum += score;
            }
        }

        return sum / userIds.size();
    }
}
```

3. **Elasticsearch Configuration**:

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

```
# application.properties
spring.elasticsearch.uris=http://localhost:9200
spring.elasticsearch.username=elastic
spring.elasticsearch.password=password
```

```
@Configuration
@EnableElasticsearchRepositories(basePackages =
"com.example.repository.elasticsearch")
public class ElasticsearchConfig extends AbstractElasticsearchConfiguration {

    @Value("${spring.elasticsearch.uris}")
    private String elasticsearchUrl;

    @Value("${spring.elasticsearch.username}")
    private String username;

    @Value("${spring.elasticsearch.password}")
    private String password;

    @Override
    public RestHighLevelClient elasticsearchClient() {
        final ClientConfiguration clientConfiguration =
ClientConfiguration.builder()
                .connectedTo(elasticsearchUrl.replace("http://", ""))
                .withBasicAuth(username, password)
                .build();

        return RestClients.create(clientConfiguration).rest();
    }
}
```

```
// Elasticsearch document
@Document(indexName = "products")
public class ProductDocument {

    @Id
    private String id;

    @Field(type = FieldType.Text, name = "name")
    private String name;

    @Field(type = FieldType.Text, name = "description")
    private String description;

    @Field(type = FieldType.Keyword, name = "sku")
    private String sku;

    @Field(type = FieldType.Double, name = "price")
    private BigDecimal price;

    @Field(type = FieldType.Integer, name = "stock_quantity")
    private Integer stockQuantity;

    @Field(type = FieldType.Keyword, name = "category")
    private String category;

    @Field(type = FieldType.Nested, name = "attributes")
    private Map<String, String> attributes;

    @Field(type = FieldType.Date, name = "created_date")
    private Date createdDate;

    // Getters and setters
}

// Elasticsearch repository
public interface ProductElasticsearchRepository
        extends ElasticsearchRepository<ProductDocument, String> {

    List<ProductDocument> findByName(String name);

    List<ProductDocument> findByCategory(String category);

    @Query("{\"bool\": {\"must\": [{\"match\": {\"name\": \"?0\"}}]}}")
    List<ProductDocument> findByNameCustomQuery(String name);

    @Query("{\"bool\": {\"must\": [{\"range\": {\"price\": {\"gte\": ?0, \"lte\":
?1}}}]}}")
    List<ProductDocument> findByPriceBetween(double minPrice, double maxPrice);
}

// Elasticsearch service
@Service
public class ProductSearchService {
```

```
    private final ProductElasticsearchRepository repository;
    private final ElasticsearchOperations operations;

    public ProductSearchService(
            ProductElasticsearchRepository repository,
            ElasticsearchOperations operations) {
        this.repository = repository;
        this.operations = operations;
    }

    // Basic repository methods
    public ProductDocument save(ProductDocument product) {
        return repository.save(product);
    }

    public Optional<ProductDocument> findById(String id) {
        return repository.findById(id);
    }

    public List<ProductDocument> findByName(String name) {
        return repository.findByName(name);
    }

    // Advanced search with query builder
    public List<ProductDocument> searchProducts(String keyword, String category,
                                                Double minPrice, Double maxPrice,
                                                int page, int size) {

        // Create a Boolean query
        BoolQueryBuilder boolQuery = QueryBuilders.boolQuery();

        // Add query conditions if parameters are provided
        if (keyword != null && !keyword.isEmpty()) {
            boolQuery.must(QueryBuilders.multiMatchQuery(keyword, "name",
"description")
                    .field("name", 2.0f) // Boost name field
                    .type(MultiMatchQueryBuilder.Type.BEST_FIELDS));
        }

        if (category != null && !category.isEmpty()) {
            boolQuery.filter(QueryBuilders.termQuery("category", category));
        }

        if (minPrice != null || maxPrice != null) {
            RangeQueryBuilder rangeQuery = QueryBuilders.rangeQuery("price");

            if (minPrice != null) {
                rangeQuery.gte(minPrice);
            }

            if (maxPrice != null) {
                rangeQuery.lte(maxPrice);
            }
```

```
            boolQuery.filter(rangeQuery);
        }

        // Create a search query
        NativeSearchQueryBuilder searchQueryBuilder = new
NativeSearchQueryBuilder()
                .withQuery(boolQuery)
                .withSort(SortBuilders.fieldSort("price").order(SortOrder.ASC))
                .withPageable(PageRequest.of(page, size));

        // Execute search
        SearchHits<ProductDocument> searchHits = operations.search(
                searchQueryBuilder.build(),
                ProductDocument.class);

        // Convert search hits to list of products
        return searchHits.getSearchHits().stream()
                .map(SearchHit::getContent)
                .collect(Collectors.toList());
    }

    // Aggregation example
    public Map<String, Long> getProductCountsByCategory() {
        // Create aggregation
        TermsAggregationBuilder aggregation = AggregationBuilders
                .terms("categories")
                .field("category")
                .size(100);

        // Create search query with aggregation
        NativeSearchQueryBuilder searchQueryBuilder = new
NativeSearchQueryBuilder()
                .addAggregation(aggregation)
                .withMaxResults(0); // We only want aggregation results

        // Execute search
        SearchHits<ProductDocument> searchHits = operations.search(
                searchQueryBuilder.build(),
                ProductDocument.class);

        // Get aggregation results
        Aggregations aggregations = searchHits.getAggregations();
        Terms categoriesAgg = aggregations.get("categories");

        // Convert to Map
        Map<String, Long> result = new HashMap<>();
        for (Terms.Bucket bucket : categoriesAgg.getBuckets()) {
            result.put(bucket.getKeyAsString(), bucket.getDocCount());
        }

        return result;
    }
```

```
    // Full-text search with highlighting
    public List<ProductSearchResult> searchProductsWithHighlight(String query) {
        // Create highlight fields
        HighlightBuilder highlightBuilder = new HighlightBuilder()
                .field("name")
                .field("description")
                .preTags("<strong>")
                .postTags("</strong>")
                .fragmentSize(150)
                .numOfFragments(3);

        // Create search query
        NativeSearchQueryBuilder searchQueryBuilder = new
NativeSearchQueryBuilder()
                .withQuery(QueryBuilders.multiMatchQuery(query, "name",
"description"))
                .withHighlightBuilder(highlightBuilder)
                .withPageable(PageRequest.of(0, 20));

        // Execute search
        SearchHits<ProductDocument> searchHits = operations.search(
                searchQueryBuilder.build(),
                ProductDocument.class);

        // Convert to ProductSearchResult with highlights
        return searchHits.getSearchHits().stream()
                .map(hit -> {
                    ProductDocument product = hit.getContent();
                    Map<String, List<String>> highlights =
hit.getHighlightFields();

                    return new ProductSearchResult(
                            product.getId(),
                            product.getName(),
                            product.getDescription(),
                            product.getPrice(),
                            highlights.get("name"),
                            highlights.get("description")
                    );
                })
                .collect(Collectors.toList());
    }

    // Custom DTO for search results with highlighting
    public static class ProductSearchResult {
        private String id;
        private String name;
        private String description;
        private BigDecimal price;
        private List<String> nameHighlights;
        private List<String> descriptionHighlights;

        // Constructor, getters, and setters
        public ProductSearchResult(String id, String name, String description,
```

```
                                    BigDecimal price, List<String> nameHighlights,
                                    List<String> descriptionHighlights) {
            this.id = id;
            this.name = name;
            this.description = description;
            this.price = price;
            this.nameHighlights = nameHighlights;
            this.descriptionHighlights = descriptionHighlights;
        }

        // Getters and setters
    }
}
```

4. **Cassandra Configuration**:

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

```
# application.properties
spring.cassandra.contact-points=localhost
spring.cassandra.port=9042
spring.cassandra.keyspace-name=mykeyspace
spring.cassandra.username=cassandra
spring.cassandra.password=cassandra
spring.cassandra.schema-action=create_if_not_exists
```

```
@Configuration
@EnableCassandraRepositories(basePackages = "com.example.repository.cassandra")
public class CassandraConfig extends AbstractCassandraConfiguration {

    @Value("${spring.cassandra.contact-points}")
    private String contactPoints;

    @Value("${spring.cassandra.port}")
    private int port;

    @Value("${spring.cassandra.keyspace-name}")
    private String keyspaceName;

    @Value("${spring.cassandra.username}")
    private String username;

    @Value("${spring.cassandra.password}")
    private String password;
```

```
    @Override
    protected String getKeyspaceName() {
        return keyspaceName;
    }

    @Override
    protected String getContactPoints() {
        return contactPoints;
    }

    @Override
    protected int getPort() {
        return port;
    }

    @Override
    public SchemaAction getSchemaAction() {
        return SchemaAction.CREATE_IF_NOT_EXISTS;
    }

    @Override
    protected List<CreateKeyspaceSpecification> getKeyspaceCreations() {
        CreateKeyspaceSpecification specification = CreateKeyspaceSpecification
                .createKeyspace(keyspaceName)
                .ifNotExists()
                .withSimpleReplication(3);

        return List.of(specification);
    }

    @Override
    protected SessionBuilderConfigurer getSessionBuilderConfigurer() {
        return new SessionBuilderConfigurer() {
            @Override
            public CqlSessionBuilder configure(CqlSessionBuilder
cqlSessionBuilder) {
                return cqlSessionBuilder
                        .withAuthCredentials(username, password);
            }
        };
    }
}

// Cassandra entity
@Table("products")
public class ProductCassandra {

    @PrimaryKey
    private ProductKey key;

    @Column("name")
    private String name;
```

```
    @Column("description")
    private String description;

    @Column("price")
    private BigDecimal price;

    @Column("stock_quantity")
    private Integer stockQuantity;

    @Column("category")
    private String category;

    @Column("created_date")
    private LocalDateTime createdDate;

    // Composite key class
    @PrimaryKeyClass
    public static class ProductKey {

        @PrimaryKeyColumn(name = "id", type = PrimaryKeyType.PARTITIONED)
        private UUID id;

        @PrimaryKeyColumn(name = "sku", type = PrimaryKeyType.CLUSTERED, ordinal =
0)
        private String sku;

        // Constructors, getters, and setters
    }

    // Constructors, getters, and setters
}

// Cassandra repository
public interface ProductCassandraRepository extends
CassandraRepository<ProductCassandra, ProductCassandra.ProductKey> {

    List<ProductCassandra> findByKeySkuStartingWith(String skuPrefix);

    @AllowFiltering
    List<ProductCassandra> findByCategory(String category);

    @AllowFiltering
    List<ProductCassandra> findByPriceLessThan(BigDecimal price);

    @Query("SELECT * FROM products WHERE category = ?0 AND price < ?1 ALLOW
FILTERING")
    List<ProductCassandra> findByCategoryAndPriceLessThan(String category,
BigDecimal price);
}
```

Hybrid Data Access Approaches

In real-world applications, you may need to use multiple databases for different purposes:

```
@Service
public class ProductService {

    private final ProductRepository jpaRepository;
    private final ProductRedisRepository redisRepository;
    private final ProductElasticsearchRepository elasticsearchRepository;
    private final ProductMapper mapper;

    public ProductService(
            ProductRepository jpaRepository,
            ProductRedisRepository redisRepository,
            ProductElasticsearchRepository elasticsearchRepository,
            ProductMapper mapper) {
        this.jpaRepository = jpaRepository;
        this.redisRepository = redisRepository;
        this.elasticsearchRepository = elasticsearchRepository;
        this.mapper = mapper;
    }

    public ProductDto findById(Long id) {
        // Try to get from cache first
        Optional<com.example.redis.Product> cachedProduct =
                redisRepository.findById(id.toString());

        if (cachedProduct.isPresent()) {
            return mapper.fromRedisToDto(cachedProduct.get());
        }

        // If not in cache, get from database
        com.example.entity.Product product = jpaRepository.findById(id)
                .orElseThrow(() -> new ResourceNotFoundException("Product not
found"));

        // Cache the product
        redisRepository.save(mapper.fromEntityToRedis(product));

        return mapper.toDto(product);
    }

    public List<ProductDto> search(String keyword, String category, Double
minPrice) {
        // Use Elasticsearch for full-text search
        List<com.example.elasticsearch.ProductDocument> searchResults =
                elasticsearchRepository.searchProducts(keyword, category,
minPrice, null, 0, 50);

        // Get full data from database for top results
        List<Long> productIds = searchResults.stream()
                .map(p -> Long.parseLong(p.getId()))
                .collect(Collectors.toList());
```

```
        List<com.example.entity.Product> products =
jpaRepository.findAllById(productIds);

        // Sort according to search results order
        Map<Long, Integer> orderMap = new HashMap<>();
        for (int i = 0; i < productIds.size(); i++) {
            orderMap.put(productIds.get(i), i);
        }

        products.sort(Comparator.comparing(p -> orderMap.getOrDefault(p.getId(),
Integer.MAX_VALUE)));

        return products.stream()
                .map(mapper::toDto)
                .collect(Collectors.toList());
    }

    @Transactional
    public ProductDto create(ProductDto productDto) {
        // Save to relational database
        com.example.entity.Product product = mapper.toEntity(productDto);
        com.example.entity.Product savedProduct = jpaRepository.save(product);

        // Update search index
        elasticsearchRepository.save(mapper.toElasticsearch(savedProduct));

        // Cache the product
        redisRepository.save(mapper.fromEntityToRedis(savedProduct));

        return mapper.toDto(savedProduct);
    }

    @Transactional
    public void delete(Long id) {
        // Delete from database
        jpaRepository.deleteById(id);

        // Delete from search index
        elasticsearchRepository.deleteById(id.toString());

        // Delete from cache
        redisRepository.deleteById(id.toString());
    }
}
```

Database Migration Pattern

For seamless migration between different database technologies:

```
@Component
public class DatabaseMigrationService {
```

```
    private final ProductRepository sqlRepository;
    private final ProductDocumentRepository mongoRepository;
    private final ProductMapper mapper;

    public DatabaseMigrationService(
            ProductRepository sqlRepository,
            ProductDocumentRepository mongoRepository,
            ProductMapper mapper) {
        this.sqlRepository = sqlRepository;
        this.mongoRepository = mongoRepository;
        this.mapper = mapper;
    }

    @Transactional(readOnly = true)
    public void migrateFromSqlToMongo() {
        // Process in batches to avoid memory issues
        int batchSize = 100;
        int page = 0;

        Pageable pageable = PageRequest.of(page, batchSize);
        Page<Product> productPage;

        do {
            // Fetch a batch from SQL
            productPage = sqlRepository.findAll(pageable);

            // Convert and save to MongoDB
            List<ProductDocument> documents = productPage.getContent().stream()
                    .map(mapper::toDocument)
                    .collect(Collectors.toList());

            mongoRepository.saveAll(documents);

            // Move to next page
            pageable = productPage.nextPageable();
        } while (productPage.hasNext());
    }

    @Transactional
    public void syncDatabases() {
        // Get all IDs from SQL database
        List<Long> sqlIds = sqlRepository.findAllIds();

        // Get all IDs from MongoDB
        List<String> mongoIds = mongoRepository.findAllIds();

        // Find IDs in SQL but not in MongoDB (need to add to MongoDB)
        Set<String> mongoIdSet = new HashSet<>(mongoIds);
        List<Long> idsToAddToMongo = sqlIds.stream()
                .filter(id -> !mongoIdSet.contains(id.toString()))
                .collect(Collectors.toList());

        // Find IDs in MongoDB but not in SQL (need to remove from MongoDB)
```

```
        Set<Long> sqlIdSet = new HashSet<>(sqlIds);
        List<String> idsToRemoveFromMongo = mongoIds.stream()
                .filter(id -> !sqlIdSet.contains(Long.parseLong(id)))
                .collect(Collectors.toList());

        // Process additions
        for (Long id : idsToAddToMongo) {
            sqlRepository.findById(id).ifPresent(product -> {
                ProductDocument document = mapper.toDocument(product);
                mongoRepository.save(document);
            });
        }

        // Process removals
        mongoRepository.deleteAllByIdIn(idsToRemoveFromMongo);
    }
}
```

Transaction Management

Proper transaction management ensures data consistency and integrity in Spring Boot applications.

Understanding Transactions

A transaction is a sequence of operations that is treated as a single unit of work. Transactions have four key properties (ACID):

1. **Atomicity**: All operations within a transaction succeed or fail together.
2. **Consistency**: A transaction transforms the system from one consistent state to another.
3. **Isolation**: Concurrent transactions do not interfere with each other.
4. **Durability**: Once a transaction is committed, its effects persist even in case of system failure.

Spring's Declarative Transaction Management

Spring Boot makes transaction management easy through annotations:

```
@Service
public class OrderService {

    private final OrderRepository orderRepository;
    private final ProductRepository productRepository;
    private final PaymentService paymentService;
    private final InventoryService inventoryService;

    // Constructor injection

    @Transactional
    public Order createOrder(OrderRequest request) {
        // 1. Create and save the order
        Order order = new Order();
```

```
        order.setCustomerId(request.getCustomerId());
        order.setOrderDate(LocalDateTime.now());
        order.setStatus(OrderStatus.PENDING);

        // 2. Add order items
        for (OrderItemRequest itemRequest : request.getItems()) {
            Product product =
productRepository.findById(itemRequest.getProductId())
                    .orElseThrow(() -> new ResourceNotFoundException("Product not
found"));

            // Check if there's enough stock
            if (product.getStockQuantity() < itemRequest.getQuantity()) {
                throw new InsufficientStockException(
                        "Not enough stock for product: " + product.getName());
            }

            OrderItem item = new OrderItem();
            item.setProduct(product);
            item.setQuantity(itemRequest.getQuantity());
            item.setPrice(product.getPrice());
            order.addItem(item);

            // Update product stock
            product.setStockQuantity(product.getStockQuantity() -
itemRequest.getQuantity());
            productRepository.save(product);
        }

        Order savedOrder = orderRepository.save(order);

        // 3. Process payment
        // This might throw an exception, causing transaction rollback
        PaymentResult paymentResult = paymentService.processPayment(
                savedOrder.getId(),
                request.getPaymentDetails());

        if (!paymentResult.isSuccessful()) {
            throw new PaymentFailedException(paymentResult.getErrorMessage());
        }

        // 4. Update order status
        savedOrder.setStatus(OrderStatus.PAID);

        return orderRepository.save(savedOrder);
    }
}
```

Transaction Attributes

Fine-tune transaction behavior with attributes:

```java
@Service
public class ProductService {

    private final ProductRepository productRepository;
    private final CategoryRepository categoryRepository;
    private final ProductAuditService auditService;

    // Constructor injection

    // Default transaction behavior
    @Transactional
    public Product createProduct(ProductDto productDto) {
        // Implementation
    }

    // Read-only transaction (optimization)
    @Transactional(readOnly = true)
    public List<Product> findByCategory(Long categoryId) {
        return productRepository.findByCategoryId(categoryId);
    }

    // Custom isolation level
    @Transactional(isolation = Isolation.SERIALIZABLE)
    public Product updateStock(Long id, int quantity) {
        Product product = productRepository.findById(id)
                .orElseThrow(() -> new ResourceNotFoundException("Product not
found"));

        product.setStockQuantity(quantity);
        return productRepository.save(product);
    }

    // Custom propagation behavior
    @Transactional(propagation = Propagation.REQUIRES_NEW)
    public void logProductAccess(Long productId, String userId) {
        // This always starts a new transaction, even if called from
        // within an existing transaction
        auditService.logAccess(productId, userId);
    }

    // Specify timeout
    @Transactional(timeout = 5)
    public void batchUpdate(List<Product> products) {
        productRepository.saveAll(products);
    }

    // Specify which exceptions trigger rollback
    @Transactional(rollbackFor = {DataIntegrityException.class},
                   noRollbackFor = {NotFoundException.class})
    public Product updateProductWithValidation(Long id, ProductDto productDto) {
        Product product = productRepository.findById(id)
                .orElseThrow(() -> new NotFoundException("Product not found"));
```

```
        // If this throws NotFoundException, transaction won't roll back
        // If this throws DataIntegrityException, transaction will roll back
        validateProduct(productDto);

        // Update product
        product.setName(productDto.getName());
        product.setPrice(productDto.getPrice());

        return productRepository.save(product);
    }
}
```

Understanding Propagation Behaviors

Transaction propagation defines how transactions relate to each other:

```
@Service
public class OrderService {

    private final OrderRepository orderRepository;
    private final PaymentService paymentService;
    private final NotificationService notificationService;

    // REQUIRED (default): Use current transaction or create a new one if none
exists
    @Transactional(propagation = Propagation.REQUIRED)
    public Order createOrder(OrderRequest request) {
        // All operations within this method run in the same transaction
        // If any operation fails, all operations are rolled back

        Order order = new Order();
        // Set order properties

        orderRepository.save(order);

        // This uses the same transaction
        paymentService.processPayment(order.getId(), request.getPaymentDetails());

        return order;
    }

    // REQUIRES_NEW: Always create a new transaction
    @Transactional(propagation = Propagation.REQUIRES_NEW)
    public void sendOrderConfirmation(Long orderId) {
        // This runs in a new transaction, regardless of caller's transaction
        Order order = orderRepository.findById(orderId)
                .orElseThrow(() -> new ResourceNotFoundException("Order not
found"));

        notificationService.sendEmail(order.getCustomerEmail(), "Order
Confirmation",
```

```
                "Your order #" + order.getId() + " has been placed.");

        order.setNotificationSent(true);
        orderRepository.save(order);
    }

    // SUPPORTS: Use current transaction if one exists, otherwise non-
transactional
    @Transactional(propagation = Propagation.SUPPORTS)
    public Order getOrderDetails(Long orderId) {
        // If called from a transactional context, this method runs in that
transaction
        // If called from a non-transactional context, this method runs without a
transaction
        return orderRepository.findById(orderId)
                .orElseThrow(() -> new ResourceNotFoundException("Order not
found"));
    }

    // MANDATORY: Use current transaction, throw exception if none exists
    @Transactional(propagation = Propagation.MANDATORY)
    public void updateOrderStatus(Long orderId, OrderStatus status) {
        // This method must be called from a transactional context
        // If called without a transaction, it throws TransactionRequiredException
        Order order = orderRepository.findById(orderId)
                .orElseThrow(() -> new ResourceNotFoundException("Order not
found"));

        order.setStatus(status);
        orderRepository.save(order);
    }

    // NOT_SUPPORTED: Execute non-transactionally, suspend current transaction if
one exists
    @Transactional(propagation = Propagation.NOT_SUPPORTED)
    public OrderSummary generateOrderSummary(Long orderId) {
        // This method always runs without a transaction
        // If a transaction exists, it's suspended during this method's execution
        Order order = orderRepository.findById(orderId)
                .orElseThrow(() -> new ResourceNotFoundException("Order not
found"));

        return new OrderSummary(order);
    }

    // NEVER: Execute non-transactionally, throw exception if a transaction exists
    @Transactional(propagation = Propagation.NEVER)
    public void auditOrderAccess(Long orderId, String userId) {
        // This method must be called from a non-transactional context
        // If a transaction exists, it throws IllegalTransactionStateException
        OrderAudit audit = new OrderAudit();
        audit.setOrderId(orderId);
        audit.setUserId(userId);
        audit.setAccessTime(LocalDateTime.now());
```

```
        orderAuditRepository.save(audit);
    }

    // NESTED: Execute within a nested transaction if a current transaction exists
    @Transactional(propagation = Propagation.NESTED)
    public void addOrderNotes(Long orderId, String notes) {
        // If a transaction exists, this runs in a nested transaction
        // If the nested transaction fails, it can be rolled back without
        // affecting the outer transaction
        // If no transaction exists, this behaves like REQUIRED
        Order order = orderRepository.findById(orderId)
                .orElseThrow(() -> new ResourceNotFoundException("Order not
found"));

        order.setNotes(notes);
        orderRepository.save(order);
    }
}
```

Understanding Isolation Levels

Isolation levels determine how transactions interact:

```
@Service
public class InventoryService {

    private final ProductRepository productRepository;

    // READ_UNCOMMITTED: Allows dirty reads, non-repeatable reads, and phantom
reads
    // Highest performance, lowest isolation
    @Transactional(isolation = Isolation.READ_UNCOMMITTED)
    public List<Product> getLowStockProductsReport() {
        // This can read uncommitted changes from other transactions
        return productRepository.findByStockQuantityLessThan(10);
    }

    // READ_COMMITTED: Prevents dirty reads, allows non-repeatable reads and
phantom reads
    // Default in many databases
    @Transactional(isolation = Isolation.READ_COMMITTED)
    public Product reserveStock(Long productId, int quantity) {
        Product product = productRepository.findById(productId)
                .orElseThrow(() -> new ResourceNotFoundException("Product not
found"));

        if (product.getStockQuantity() < quantity) {
            throw new InsufficientStockException("Not enough stock");
        }
```

```
        product.setStockQuantity(product.getStockQuantity() - quantity);
        return productRepository.save(product);
    }

    // REPEATABLE_READ: Prevents dirty reads and non-repeatable reads, allows
phantom reads
    @Transactional(isolation = Isolation.REPEATABLE_READ)
    public void processStockAdjustment(Long productId, int adjustment) {
        // First read - stock quantity will be the same in subsequent reads
        // within this transaction, even if other transactions modify it
        Product product = productRepository.findById(productId)
                .orElseThrow(() -> new ResourceNotFoundException("Product not
found"));

        // Some business logic that takes time
        performTimeConsumingCalculation();

        // Second read - will see the same value as the first read
        Product productAgain = productRepository.findById(productId).get();

        productAgain.setStockQuantity(productAgain.getStockQuantity() +
adjustment);
        productRepository.save(productAgain);
    }

    // SERIALIZABLE: Prevents dirty reads, non-repeatable reads, and phantom reads
    // Highest isolation, lowest performance
    @Transactional(isolation = Isolation.SERIALIZABLE)
    public void processCriticalInventoryOperation() {
        // This provides complete isolation from other transactions
        // Other transactions cannot modify data that this transaction reads
        List<Product> products = productRepository.findAll();

        for (Product product : products) {
            // Complex inventory adjustments
        }
    }
}
```

Programmatic Transaction Management

Sometimes declarative transactions aren't enough. Use programmatic transactions for more control:

```
@Service
public class ComplexTransactionService {

    private final PlatformTransactionManager transactionManager;
    private final OrderRepository orderRepository;
    private final ProductRepository productRepository;

    public ComplexTransactionService(
```

```
            PlatformTransactionManager transactionManager,
            OrderRepository orderRepository,
            ProductRepository productRepository) {
        this.transactionManager = transactionManager;
        this.orderRepository = orderRepository;
        this.productRepository = productRepository;
    }

    public void processOrderWithCustomTransaction(OrderRequest request) {
        // Define transaction attributes
        TransactionDefinition txDef = new DefaultTransactionDefinition(
                TransactionDefinition.PROPAGATION_REQUIRED);

        // Start a transaction
        TransactionStatus txStatus = transactionManager.getTransaction(txDef);

        try {
            // Perform transactional operations
            Order order = new Order();
            order.setCustomerId(request.getCustomerId());
            order.setOrderDate(LocalDateTime.now());

            // Add order items
            for (OrderItemRequest itemRequest : request.getItems()) {
                Product product =
productRepository.findById(itemRequest.getProductId())
                        .orElseThrow(() -> new ResourceNotFoundException("Product
not found"));

                OrderItem item = new OrderItem();
                item.setProduct(product);
                item.setQuantity(itemRequest.getQuantity());
                item.setPrice(product.getPrice());
                order.addItem(item);
            }

            orderRepository.save(order);

            // Create a save point
            Object savepoint = txStatus.createSavepoint();

            try {
                // Risky operation that might fail
                processPayment(order.getId(), request.getPaymentDetails());
            } catch (PaymentException e) {
                // Rollback to the savepoint instead of the whole transaction
                txStatus.rollbackToSavepoint(savepoint);

                // Mark order as payment failed but still save it
                order.setStatus(OrderStatus.PAYMENT_FAILED);
                order.setFailureReason(e.getMessage());
                orderRepository.save(order);
            }
```

```
            // Commit the transaction
            transactionManager.commit(txStatus);
        } catch (Exception e) {
            // Rollback the transaction
            transactionManager.rollback(txStatus);
            throw e;
        }
    }

    public void processBatchWithChunkedTransactions(List<ProductUpdateRequest>
updates) {
        int chunkSize = 100;

        for (int i = 0; i < updates.size(); i += chunkSize) {
            int endIndex = Math.min(i + chunkSize, updates.size());
            List<ProductUpdateRequest> chunk = updates.subList(i, endIndex);

            // Process each chunk in its own transaction
            TransactionDefinition txDef = new DefaultTransactionDefinition(
                    TransactionDefinition.PROPAGATION_REQUIRES_NEW);
            TransactionStatus txStatus = transactionManager.getTransaction(txDef);

            try {
                for (ProductUpdateRequest update : chunk) {
                    Product product =
productRepository.findById(update.getProductId())
                            .orElseThrow(() -> new
ResourceNotFoundException("Product not found"));

                    product.setPrice(update.getNewPrice());
                    product.setStockQuantity(update.getNewStockQuantity());

                    productRepository.save(product);
                }

                transactionManager.commit(txStatus);
            } catch (Exception e) {
                transactionManager.rollback(txStatus);
                // Log failure and continue with next chunk
                log.error("Failed to process chunk starting at index " + i, e);
            }
        }
    }
}
```

Distributed Transactions

Handling transactions across multiple services:

```
@Configuration
public class XaTransactionConfig {
```

```
    @Bean
    public DataSource dataSource() {
        AtomikosDataSourceBean ds = new AtomikosDataSourceBean();
        ds.setUniqueResourceName("xaDataSource");
        ds.setXaDataSourceClassName("com.mysql.cj.jdbc.MysqlXADataSource");

        Properties props = new Properties();
        props.setProperty("URL", "jdbc:mysql://localhost:3306/mydb");
        props.setProperty("user", "root");
        props.setProperty("password", "password");

        ds.setXaProperties(props);
        return ds;
    }

    @Bean
    public JdbcTemplate jdbcTemplate(DataSource dataSource) {
        return new JdbcTemplate(dataSource);
    }

    @Bean
    public JtaTransactionManager transactionManager() {
        return new JtaTransactionManager();
    }
}

@Service
public class DistributedTransactionService {

    private final OrderRepository orderRepository;
    private final RestTemplate restTemplate;

    public DistributedTransactionService(
            OrderRepository orderRepository,
            RestTemplate restTemplate) {
        this.orderRepository = orderRepository;
        this.restTemplate = restTemplate;
    }

    // For XA transactions (two-phase commit)
    @Transactional
    public Order createOrderWithExternalPayment(OrderRequest request) {
        // Create local order
        Order order = new Order();
        order.setCustomerId(request.getCustomerId());
        order.setOrderDate(LocalDateTime.now());
        order.setStatus(OrderStatus.PENDING);

        Order savedOrder = orderRepository.save(order);

        // Call external payment service
        // This is part of the same XA transaction if the payment service
        // is also configured to use XA transactions
```

```
        PaymentRequest paymentRequest = new PaymentRequest(
                savedOrder.getId(),
                request.getPaymentDetails());

        PaymentResponse response = restTemplate.postForObject(
                "http://payment-service/api/payments",
                paymentRequest,
                PaymentResponse.class);

        if (!response.isSuccessful()) {
            throw new PaymentFailedException(response.getErrorMessage());
        }

        savedOrder.setStatus(OrderStatus.PAID);
        return orderRepository.save(savedOrder);
    }

    // For Saga pattern (compensating transactions)
    public Order createOrderWithSaga(OrderRequest request) {
        // 1. Create order
        Order order = new Order();
        order.setCustomerId(request.getCustomerId());
        order.setOrderDate(LocalDateTime.now());
        order.setStatus(OrderStatus.PENDING);

        Order savedOrder = orderRepository.save(order);

        try {
            // 2. Reserve inventory
            InventoryRequest inventoryRequest = new InventoryRequest(
                    savedOrder.getId(),
                    savedOrder.getItems().stream()
                            .map(item -> new InventoryItemRequest(
                                    item.getProduct().getId(),
                                    item.getQuantity()))
                            .collect(Collectors.toList()));

            InventoryResponse inventoryResponse = restTemplate.postForObject(
                    "http://inventory-service/api/inventory/reserve",
                    inventoryRequest,
                    InventoryResponse.class);

            if (!inventoryResponse.isSuccessful()) {
                // Compensating transaction
                orderRepository.delete(savedOrder);
                throw new InventoryException(inventoryResponse.getErrorMessage());
            }

            // 3. Process payment
            PaymentRequest paymentRequest = new PaymentRequest(
                    savedOrder.getId(),
                    request.getPaymentDetails());

            PaymentResponse paymentResponse = restTemplate.postForObject(
```

```
                    "http://payment-service/api/payments",
                    paymentRequest,
                    PaymentResponse.class);

            if (!paymentResponse.isSuccessful()) {
                // Compensating transactions
                restTemplate.postForObject(
                        "http://inventory-service/api/inventory/release",
                        inventoryRequest,
                        InventoryResponse.class);

                orderRepository.delete(savedOrder);
                throw new
PaymentFailedException(paymentResponse.getErrorMessage());
            }

            // 4. Update order status
            savedOrder.setStatus(OrderStatus.PAID);
            return orderRepository.save(savedOrder);
        } catch (Exception e) {
            // Ensure order is deleted in case of unexpected errors
            orderRepository.delete(savedOrder);
            throw e;
        }
    }
}
```

Best Practices for Transaction Management

1. **Keep Transactions Short**: Long-running transactions can lead to contention and deadlocks.

```
// Bad practice: too many operations in one transaction
@Transactional
public void processEndOfDayOrders() {
    List<Order> orders = orderRepository.findByStatus(OrderStatus.PENDING);

    for (Order order : orders) {
        // Process each order
        // Update inventory
        // Send notifications
        // Generate reports
    }
}

// Better approach: smaller transactions
public void processEndOfDayOrders() {
    List<Long> orderIds = orderRepository.findPendingOrderIds();

    for (Long orderId : orderIds) {
        try {
            processIndividualOrder(orderId);
```

```
        } catch (Exception e) {
            log.error("Failed to process order: " + orderId, e);
            // Continue with next order
        }
    }
}

@Transactional
public void processIndividualOrder(Long orderId) {
    // Process a single order in its own transaction
}
```

2. **Use Appropriate Isolation Levels**: Higher isolation levels provide more consistency but reduce concurrency.

```
// Choose isolation based on requirements
@Transactional(isolation = Isolation.READ_COMMITTED)
public void updateProductPrice(Long productId, BigDecimal newPrice) {
    // READ_COMMITTED is usually sufficient for simple updates
}

@Transactional(isolation = Isolation.SERIALIZABLE)
public void transferFunds(Long fromAccountId, Long toAccountId, BigDecimal amount)
{
    // SERIALIZABLE for critical financial operations
}
```

3. **Consider Read-Only Transactions**: Mark read-only operations to allow optimizations.

```
@Transactional(readOnly = true)
public List<ProductDto> findProductsByCategory(Long categoryId) {
    // Database can optimize read-only transactions
    return productRepository.findByCategoryId(categoryId).stream()
            .map(productMapper::toDto)
            .collect(Collectors.toList());
}
```

4. **Handle Exceptions Properly**: Be clear about which exceptions should trigger rollback.

```
@Transactional(rollbackFor = DataAccessException.class,
               noRollbackFor = ResourceNotFoundException.class)
public Product updateProductStock(Long productId, int quantity) {
    // Specific exception handling
}
```

5. **Avoid Unnecessary Nesting**: Be careful with transaction propagation in nested service calls.

```
@Service
public class OrderService {

    private final OrderRepository orderRepository;
    private final NotificationService notificationService;

    @Transactional
    public Order createOrder(OrderRequest request) {
        Order order = new Order();
        // Set order properties

        Order savedOrder = orderRepository.save(order);

        // Use REQUIRES_NEW to ensure notification is sent in a separate
transaction
        notificationService.sendOrderConfirmation(savedOrder.getId());

        return savedOrder;
    }
}

@Service
public class NotificationService {

    @Transactional(propagation = Propagation.REQUIRES_NEW)
    public void sendOrderConfirmation(Long orderId) {
        // This runs in a new transaction
        // If this fails, the order transaction will still commit
    }
}
```

Advanced Query Techniques

Spring Data JPA offers several advanced query techniques for complex data access requirements.

Dynamic Queries with Specifications

Specifications allow building dynamic queries based on runtime conditions:

```
public class ProductSpecifications {

    public static Specification<Product> hasName(String name) {
        return (root, query, cb) -> {
            if (name == null || name.isEmpty()) {
                return cb.conjunction(); // Always true predicate
            }
            return cb.like(cb.lower(root.get("name")), "%" + name.toLowerCase() +
"%");
        };
    }
```

```
    public static Specification<Product> hasCategory(Long categoryId) {
        return (root, query, cb) -> {
            if (categoryId == null) {
                return cb.conjunction();
            }
            return cb.equal(root.get("category").get("id"), categoryId);
        };
    }

    public static Specification<Product> priceBetween(BigDecimal min, BigDecimal
max) {
        return (root, query, cb) -> {
            if (min == null && max == null) {
                return cb.conjunction();
            }

            if (min == null) {
                return cb.lessThanOrEqualTo(root.get("price"), max);
            }

            if (max == null) {
                return cb.greaterThanOrEqualTo(root.get("price"), min);
            }

            return cb.between(root.get("price"), min, max);
        };
    }

    public static Specification<Product> isInStock() {
        return (root, query, cb) -> cb.greaterThan(root.get("stockQuantity"), 0);
    }

    public static Specification<Product> hasAttributes(Map<String, String>
attributes) {
        return (root, query, cb) -> {
            if (attributes == null || attributes.isEmpty()) {
                return cb.conjunction();
            }

            Join<Product, ProductAttribute> attributeJoin =
root.join("attributes");

            List<Predicate> predicates = new ArrayList<>();

            for (Map.Entry<String, String> entry : attributes.entrySet()) {
                predicates.add(
                        cb.and(
                                cb.equal(attributeJoin.get("name"),
entry.getKey()),
                                cb.equal(attributeJoin.get("value"),
entry.getValue())
                        )
                );
```

```
            }

            return cb.and(predicates.toArray(new Predicate[0]));
        };
    }
}

@Repository
public interface ProductRepository extends JpaRepository<Product, Long>,
                                           JpaSpecificationExecutor<Product> {
}

@Service
public class ProductService {

    private final ProductRepository productRepository;

    public Page<Product> findProducts(
            String name,
            Long categoryId,
            BigDecimal minPrice,
            BigDecimal maxPrice,
            boolean inStockOnly,
            Map<String, String> attributes,
            Pageable pageable) {

        Specification<Product> spec = Specification.where(null);

        spec = spec.and(hasName(name));
        spec = spec.and(hasCategory(categoryId));
        spec = spec.and(priceBetween(minPrice, maxPrice));

        if (inStockOnly) {
            spec = spec.and(isInStock());
        }

        if (attributes != null && !attributes.isEmpty()) {
            spec = spec.and(hasAttributes(attributes));
        }

        return productRepository.findAll(spec, pageable);
    }
}
```

Complex Joins and Fetch Strategies

Optimize database access with smart join and fetch strategies:

```
public interface OrderRepository extends JpaRepository<Order, Long> {

    // Fetch join to avoid N+1 problem
```

```
    @Query("SELECT o FROM Order o " +
           "JOIN FETCH o.items i " +
           "JOIN FETCH i.product " +
           "WHERE o.customer.id = :customerId")
    List<Order> findByCustomerIdWithItems(@Param("customerId") Long customerId);

    // Selective fetch join based on conditions
    @Query("SELECT DISTINCT o FROM Order o " +
           "LEFT JOIN FETCH o.items i " +
           "LEFT JOIN FETCH i.product p " +
           "WHERE o.orderDate >= :startDate AND " +
           "(:categoryId IS NULL OR p.category.id = :categoryId)")
    List<Order> findRecentOrdersWithProducts(
            @Param("startDate") LocalDateTime startDate,
            @Param("categoryId") Long categoryId);

    // Using entity graph for dynamic fetch
    @EntityGraph(attributePaths = {"customer", "items.product"})
    List<Order> findByStatusIn(List<OrderStatus> statuses);
}

@Service
public class OrderService {

    private final OrderRepository orderRepository;
    private final EntityManager entityManager;

    public List<Order> findComplexOrders(
            Long customerId,
            List<OrderStatus> statuses,
            boolean includeItems,
            boolean includePayments) {

        // Use Criteria API for dynamic fetch joins
        CriteriaBuilder cb = entityManager.getCriteriaBuilder();
        CriteriaQuery<Order> query = cb.createQuery(Order.class);
        Root<Order> root = query.from(Order.class);

        // Apply conditions
        List<Predicate> predicates = new ArrayList<>();

        if (customerId != null) {
            predicates.add(cb.equal(root.get("customer").get("id"), customerId));
        }

        if (statuses != null && !statuses.isEmpty()) {
            predicates.add(root.get("status").in(statuses));
        }

        query.where(predicates.toArray(new Predicate[0]));

        // Apply dynamic fetch joins
        if (includeItems) {
            root.fetch("items", JoinType.LEFT)
```

```
                .fetch("product", JoinType.LEFT);
        }

        if (includePayments) {
            root.fetch("payments", JoinType.LEFT);
        }

        // Execute query
        return entityManager.createQuery(query)
                .setHint(QueryHints.HINT_PASS_DISTINCT_THROUGH, false)
                .getResultList();
    }
}
```

Native SQL Queries for Complex Operations

Use native SQL when JPA capabilities are insufficient:

```
public interface ProductRepository extends JpaRepository<Product, Long> {

    // Simple native query
    @Query(value = "SELECT * FROM products WHERE date(created_at) = CURDATE()",
           nativeQuery = true)
    List<Product> findProductsCreatedToday();

    // Native query with pagination
    @Query(value = "SELECT * FROM products p " +
                   "JOIN product_categories pc ON p.category_id = pc.id " +
                   "WHERE pc.name = :categoryName " +
                   "ORDER BY p.created_at DESC",
           countQuery = "SELECT COUNT(*) FROM products p " +
                        "JOIN product_categories pc ON p.category_id = pc.id " +
                        "WHERE pc.name = :categoryName",
           nativeQuery = true)
    Page<Product> findByCategoryNameNative(
            @Param("categoryName") String categoryName,
            Pageable pageable);

    // Native query with custom projection
    @Query(value = "SELECT p.id as id, p.name as name, " +
                   "p.price as price, SUM(oi.quantity) as totalSold " +
                   "FROM products p " +
                   "JOIN order_items oi ON p.id = oi.product_id " +
                   "JOIN orders o ON oi.order_id = o.id " +
                   "WHERE o.status = 'COMPLETED' " +
                   "AND o.order_date BETWEEN :startDate AND :endDate " +
                   "GROUP BY p.id, p.name, p.price " +
                   "ORDER BY totalSold DESC " +
                   "LIMIT :limit",
           nativeQuery = true)
    List<ProductSalesProjection> findTopSellingProducts(
```

```
            @Param("startDate") Date startDate,
            @Param("endDate") Date endDate,
            @Param("limit") int limit);

    // Custom projection interface
    interface ProductSalesProjection {
        Long getId();
        String getName();
        BigDecimal getPrice();
        Integer getTotalSold();
    }

    // Modifying native query
    @Modifying
    @Query(value = "UPDATE products SET stock_quantity = stock_quantity -
:quantity " +
                   "WHERE id = :id AND stock_quantity >= :quantity",
           nativeQuery = true)
    int decreaseStockIfAvailable(@Param("id") Long id, @Param("quantity") int
quantity);
}

@Service
public class AdvancedQueryService {

    private final EntityManager entityManager;
    private final JdbcTemplate jdbcTemplate;

    // Using EntityManager for dynamic native queries
    public List<Map<String, Object>> getCustomSalesReport(
            Date startDate, Date endDate, List<String> categories) {

        StringBuilder sql = new StringBuilder();
        sql.append("SELECT c.name as category, ")
           .append("       MONTH(o.order_date) as month, ")
           .append("       SUM(oi.quantity * oi.price) as total_sales ")
           .append("FROM orders o ")
           .append("JOIN order_items oi ON o.id = oi.order_id ")
           .append("JOIN products p ON oi.product_id = p.id ")
           .append("JOIN product_categories c ON p.category_id = c.id ")
           .append("WHERE o.status = 'COMPLETED' ")
           .append("AND o.order_date BETWEEN ? AND ? ");

        if (categories != null && !categories.isEmpty()) {
            sql.append("AND c.name IN (");
            for (int i = 0; i < categories.size(); i++) {
                if (i > 0) {
                    sql.append(", ");
                }
                sql.append("?");
            }
            sql.append(") ");
        }
```

```
        sql.append("GROUP BY c.name, MONTH(o.order_date) ")
           .append("ORDER BY c.name, MONTH(o.order_date)");

        Query query = entityManager.createNativeQuery(sql.toString());
        query.setParameter(1, startDate);
        query.setParameter(2, endDate);

        if (categories != null && !categories.isEmpty()) {
            for (int i = 0; i < categories.size(); i++) {
                query.setParameter(i + 3, categories.get(i));
            }
        }

        List<Object[]> results = query.getResultList();

        return results.stream()
                .map(row -> {
                    Map<String, Object> map = new HashMap<>();
                    map.put("category", row[0]);
                    map.put("month", row[1]);
                    map.put("totalSales", row[2]);
                    return map;
                })
                .collect(Collectors.toList());
    }

    // Using JdbcTemplate for complex queries
    public List<ProductInventoryDto> getProductInventorySummary() {
        String sql = "WITH inventory_summary AS (" +
                     "    SELECT p.id, p.name, p.stock_quantity, " +
                     "           SUM(CASE WHEN o.status = 'PENDING' THEN
oi.quantity ELSE 0 END) as reserved, " +
                     "           SUM(CASE WHEN o.status IN ('SHIPPED', 'DELIVERED')
THEN oi.quantity ELSE 0 END) as sold_30days " +
                     "    FROM products p " +
                     "    LEFT JOIN order_items oi ON p.id = oi.product_id " +
                     "    LEFT JOIN orders o ON oi.order_id = o.id AND o.order_date
>= DATE_SUB(CURDATE(), INTERVAL 30 DAY) " +
                     "    GROUP BY p.id, p.name, p.stock_quantity" +
                     ") " +
                     "SELECT id, name, stock_quantity, reserved, sold_30days, " +
                     "       (stock_quantity - reserved) as available, " +
                     "       CASE " +
                     "           WHEN stock_quantity = 0 THEN 'OUT_OF_STOCK' " +
                     "           WHEN (stock_quantity - reserved) <= (sold_30days /
30 * 7) THEN 'LOW_STOCK' " +
                     "           ELSE 'IN_STOCK' " +
                     "       END as stock_status " +
                     "FROM inventory_summary " +
                     "ORDER BY " +
                     "    CASE stock_status " +
                     "        WHEN 'OUT_OF_STOCK' THEN 1 " +
                     "        WHEN 'LOW_STOCK' THEN 2 " +
                     "        ELSE 3 " +
```

```
                    "    END, name";

        return jdbcTemplate.query(sql, (rs, rowNum) -> {
            ProductInventoryDto dto = new ProductInventoryDto();
            dto.setId(rs.getLong("id"));
            dto.setName(rs.getString("name"));
            dto.setStockQuantity(rs.getInt("stock_quantity"));
            dto.setReserved(rs.getInt("reserved"));
            dto.setSold30Days(rs.getInt("sold_30days"));
            dto.setAvailable(rs.getInt("available"));
            dto.setStockStatus(StockStatus.valueOf(rs.getString("stock_status")));
            return dto;
        });
    }
}
```

Advanced Projections and DTOs

Optimize data transfer with projections and DTOs:

```
// Closed projection (interface-based)
public interface ProductSummary {
    Long getId();
    String getName();
    BigDecimal getPrice();

    @Value("#{target.price.multiply(new java.math.BigDecimal(1.2))}")
    BigDecimal getPriceWithTax();
}

// Open projection with custom logic
public interface ProductDetail {
    Long getId();
    String getName();
    String getDescription();
    BigDecimal getPrice();

    @Value("#{target.stockQuantity > 0 ? 'In Stock' : 'Out of Stock'}")
    String getStockStatus();

    @Value("#{@categoryService.getCategoryName(target.category.id)}")
    String getCategoryName();
}

// Class-based projection with constructor
public class ProductListItem {
    private final Long id;
    private final String name;
    private final BigDecimal price;
    private final Integer stockQuantity;
```

```
    // Constructor used by Spring Data JPA with constructor expression
    public ProductListItem(Long id, String name, BigDecimal price, Integer
stockQuantity) {
        this.id = id;
        this.name = name;
        this.price = price;
        this.stockQuantity = stockQuantity;
    }

    // Getters
}

// Repository with projection support
public interface ProductRepository extends JpaRepository<Product, Long> {

    // Interface-based projection
    List<ProductSummary> findByCategory_Id(Long categoryId);

    // Class-based projection with constructor expression
    @Query("SELECT new com.example.dto.ProductListItem(p.id, p.name, p.price,
p.stockQuantity) " +
           "FROM Product p WHERE p.category.id = :categoryId")
    List<ProductListItem> findProductListItemsByCategoryId(@Param("categoryId")
Long categoryId);

    // Dynamic projection
    <T> List<T> findByPriceGreaterThan(BigDecimal price, Class<T> type);
}

@Service
public class ProductService {

    private final ProductRepository productRepository;

    public <T> List<T> findProductsByPriceRange(
            BigDecimal minPrice, BigDecimal maxPrice, Class<T> projectionType) {

        if (minPrice != null && maxPrice != null) {
            return productRepository.findByPriceBetween(minPrice, maxPrice,
projectionType);
        } else if (minPrice != null) {
            return productRepository.findByPriceGreaterThanEqual(minPrice,
projectionType);
        } else if (maxPrice != null) {
            return productRepository.findByPriceLessThanEqual(maxPrice,
projectionType);
        } else {
            return productRepository.findBy(projectionType);
        }
    }

    // Custom mapping with MapStruct
    @Autowired
    private ProductMapper productMapper;
```

```
    public List<ProductDto> findProductsByCriteria(ProductSearchCriteria criteria)
{
        Specification<Product> spec = buildSpecification(criteria);
        List<Product> products = productRepository.findAll(spec);

        return productMapper.toDtoList(products);
    }
}
```

Dynamic Sorting and Filtering

Allow flexible sorting and filtering options:

```
@Controller
@RequestMapping("/api/products")
public class ProductController {

    private final ProductService productService;

    public ProductController(ProductService productService) {
        this.productService = productService;
    }

    @GetMapping
    public ResponseEntity<Page<ProductDto>> getProducts(
            @RequestParam(required = false) String name,
            @RequestParam(required = false) Long categoryId,
            @RequestParam(required = false) BigDecimal minPrice,
            @RequestParam(required = false) BigDecimal maxPrice,
            @RequestParam(required = false, defaultValue = "false") boolean
inStockOnly,
            @RequestParam(required = false, defaultValue = "0") int page,
            @RequestParam(required = false, defaultValue = "10") int size,
            @RequestParam(required = false, defaultValue = "id") String sort,
            @RequestParam(required = false, defaultValue = "asc") String
direction) {

        // Parse sort parameters
        List<SortCriteria> sortCriteria = parseSortParameter(sort, direction);

        // Create pageable with sort
        Pageable pageable = createPageable(page, size, sortCriteria);

        // Apply filters and sort
        Page<ProductDto> products = productService.findProducts(
                name, categoryId, minPrice, maxPrice, inStockOnly, pageable);

        return ResponseEntity.ok(products);
    }
```

```
    private List<SortCriteria> parseSortParameter(String sort, String direction) {
        List<SortCriteria> result = new ArrayList<>();

        // Handle multiple sort fields (comma-separated)
        String[] sortFields = sort.split(",");
        String[] directions = direction.split(",");

        for (int i = 0; i < sortFields.length; i++) {
            String field = sortFields[i].trim();

            // Default to ASC if no direction specified for this field
            Sort.Direction dir = Sort.Direction.ASC;
            if (i < directions.length) {
                dir = "desc".equalsIgnoreCase(directions[i].trim()) ?
                        Sort.Direction.DESC : Sort.Direction.ASC;
            }

            result.add(new SortCriteria(field, dir));
        }

        return result;
    }

    private Pageable createPageable(int page, int size, List<SortCriteria>
sortCriteria) {
        if (sortCriteria.isEmpty()) {
            return PageRequest.of(page, size);
        }

        List<Sort.Order> orders = sortCriteria.stream()
                .map(sc -> new Sort.Order(sc.getDirection(), sc.getField()))
                .collect(Collectors.toList());

        return PageRequest.of(page, size, Sort.by(orders));
    }

    private static class SortCriteria {
        private final String field;
        private final Sort.Direction direction;

        public SortCriteria(String field, Sort.Direction direction) {
            this.field = field;
            this.direction = direction;
        }

        public String getField() {
            return field;
        }

        public Sort.Direction getDirection() {
            return direction;
        }
    }
}
```

```java
@Service
public class ProductService {

    private final ProductRepository productRepository;
    private final ProductMapper productMapper;

    // Constructor injection

    public Page<ProductDto> findProducts(
            String name,
            Long categoryId,
            BigDecimal minPrice,
            BigDecimal maxPrice,
            boolean inStockOnly,
            Pageable pageable) {

        // Security check: validate and sanitize sort fields
        validateSortFields(pageable);

        // Build specification
        Specification<Product> spec = Specification.where(null);

        if (name != null && !name.isEmpty()) {
            spec = spec.and(ProductSpecifications.hasName(name));
        }

        if (categoryId != null) {
            spec = spec.and(ProductSpecifications.hasCategory(categoryId));
        }

        spec = spec.and(ProductSpecifications.priceBetween(minPrice, maxPrice));

        if (inStockOnly) {
            spec = spec.and(ProductSpecifications.isInStock());
        }

        // Query with specification and pageable
        Page<Product> productPage = productRepository.findAll(spec, pageable);

        // Map to DTOs
        return productPage.map(productMapper::toDto);
    }

    private void validateSortFields(Pageable pageable) {
        // Set of allowed sort fields
        Set<String> allowedFields = Set.of(
                "id", "name", "price", "stockQuantity", "createdAt");

        if (pageable.getSort() != null) {
            for (Sort.Order order : pageable.getSort()) {
                String property = order.getProperty();
                if (!allowedFields.contains(property)) {
                    throw new InvalidSortFieldException("Invalid sort field: " +
```

```
property);
                }
            }
        }
    }
}
```

Bulk Operations and Batch Processing

Optimize performance for bulk operations:

```
@Service
@Transactional
public class BulkOperationService {

    private final EntityManager entityManager;
    private final JdbcTemplate jdbcTemplate;
    private final ProductRepository productRepository;

    // Constructor injection

    // Batch inserts with JPA
    public void batchInsertProducts(List<Product> products) {
        int batchSize = 50;

        for (int i = 0; i < products.size(); i++) {
            entityManager.persist(products.get(i));

            // Flush and clear the persistence context periodically
            if (i % batchSize == 0 || i == products.size() - 1) {
                entityManager.flush();
                entityManager.clear();
            }
        }
    }

    // Batch updates with JPA
    public void batchUpdateProducts(List<Product> products) {
        int batchSize = 50;

        for (int i = 0; i < products.size(); i++) {
            Product product = products.get(i);

            if (product.getId() != null) {
                entityManager.merge(product);
            }

            if (i % batchSize == 0 || i == products.size() - 1) {
                entityManager.flush();
                entityManager.clear();
            }
```

```
        }
    }

    // Batch deletes with JPA
    public void batchDeleteProducts(List<Long> productIds) {
        productRepository.deleteAllByIdInBatch(productIds);
    }

    // Bulk update with JPQL
    @Modifying
    @Query("UPDATE Product p SET p.price = p.price * :factor WHERE p.category.id =
:categoryId")
    public void increasePricesForCategory(Long categoryId, BigDecimal factor) {
        productRepository.increasePricesForCategory(categoryId, factor);
    }

    // Bulk update with native SQL
    public void updateProductStockQuantities(Map<Long, Integer> stockUpdates) {
        String sql = "UPDATE products SET stock_quantity = ? WHERE id = ?";

        jdbcTemplate.batchUpdate(sql, new BatchPreparedStatementSetter() {
            @Override
            public void setValues(PreparedStatement ps, int i) throws SQLException
{
                Map.Entry<Long, Integer> entry =

stockUpdates.entrySet().stream().skip(i).findFirst().get();
                ps.setInt(1, entry.getValue());
                ps.setLong(2, entry.getKey());
            }

            @Override
            public int getBatchSize() {
                return stockUpdates.size();
            }
        });
    }

    // Mass insert with JDBC
    public void bulkInsertOrderItems(List<OrderItem> items) {
        String sql = "INSERT INTO order_items (order_id, product_id, quantity,
price) " +
                     "VALUES (?, ?, ?, ?)";

        jdbcTemplate.batchUpdate(sql, new BatchPreparedStatementSetter() {
            @Override
            public void setValues(PreparedStatement ps, int i) throws SQLException
{
                OrderItem item = items.get(i);
                ps.setLong(1, item.getOrder().getId());
                ps.setLong(2, item.getProduct().getId());
                ps.setInt(3, item.getQuantity());
                ps.setBigDecimal(4, item.getPrice());
            }
```

```
            @Override
            public int getBatchSize() {
                return items.size();
            }
        });
    }
}
```

Caching Strategies

Caching improves application performance by storing frequently accessed data in memory.

Spring Cache Abstraction

Spring provides a caching abstraction that supports various cache providers:

```
@Configuration
@EnableCaching
public class CachingConfig {

    @Bean
    public CacheManager cacheManager() {
        // Simple in-memory cache for development
        SimpleCacheManager cacheManager = new SimpleCacheManager();

        cacheManager.setCaches(Arrays.asList(
                new ConcurrentMapCache("products"),
                new ConcurrentMapCache("categories"),
                new ConcurrentMapCache("productsByCategory")
        ));

        return cacheManager;
    }
}

@Service
public class ProductService {

    private final ProductRepository productRepository;

    // Basic caching
    @Cacheable("products")
    public Product findById(Long id) {
        // This will be executed only if the product is not in the cache
        return productRepository.findById(id)
                .orElseThrow(() -> new ResourceNotFoundException("Product not
found"));
    }

    // Conditional caching
```

```
    @Cacheable(value = "products", condition = "#id > 0", unless = "#result ==
null")
    public Product findByIdWithCondition(Long id) {
        return productRepository.findById(id).orElse(null);
    }

    // Cache with key
    @Cacheable(value = "productsByCategory", key = "#categoryId")
    public List<Product> findByCategoryId(Long categoryId) {
        return productRepository.findByCategoryId(categoryId);
    }

    // Cache eviction
    @CacheEvict(value = "products", key = "#id")
    public void deleteProduct(Long id) {
        productRepository.deleteById(id);
    }

    // Update cache
    @CachePut(value = "products", key = "#product.id")
    public Product updateProduct(Product product) {
        return productRepository.save(product);
    }

    // Multiple cache operations
    @Caching(evict = {
        @CacheEvict(value = "products", key = "#product.id"),
        @CacheEvict(value = "productsByCategory", key = "#product.category.id")
    })
    public void refreshProduct(Product product) {
        // Some logic to refresh product data
    }

    // Clear all entries in a cache
    @CacheEvict(value = "products", allEntries = true)
    public void clearAllProductCache() {
        // Method can be empty, the annotation does the work
    }
}
```

Redis for Distributed Caching

Redis is a popular choice for distributed caching:

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

```java
@Configuration
@EnableCaching
public class RedisCacheConfig {

    @Bean
    public RedisConnectionFactory redisConnectionFactory() {
        LettuceConnectionFactory connectionFactory = new
LettuceConnectionFactory();
        // Configure connection properties if needed
        // connectionFactory.setHostName("localhost");
        // connectionFactory.setPort(6379);
        return connectionFactory;
    }

    @Bean
    public RedisTemplate<String, Object> redisTemplate() {
        RedisTemplate<String, Object> template = new RedisTemplate<>();
        template.setConnectionFactory(redisConnectionFactory());

        // Use Jackson2JsonRedisSerializer to serialize/deserialize objects
        Jackson2JsonRedisSerializer<Object> serializer =
                new Jackson2JsonRedisSerializer<>(Object.class);

        // Configure ObjectMapper for better type information
        ObjectMapper mapper = new ObjectMapper();
        mapper.activateDefaultTyping(
                LaissezFaireSubTypeValidator.instance,
                ObjectMapper.DefaultTyping.NON_FINAL,
                JsonTypeInfo.As.PROPERTY
        );
        serializer.setObjectMapper(mapper);

        template.setValueSerializer(serializer);
        template.setHashValueSerializer(serializer);

        // Use StringRedisSerializer for keys
        template.setKeySerializer(new StringRedisSerializer());
        template.setHashKeySerializer(new StringRedisSerializer());

        template.afterPropertiesSet();
        return template;
    }

    @Bean
    public CacheManager cacheManager(RedisConnectionFactory
redisConnectionFactory) {
        // Set up Redis cache manager
        RedisCacheConfiguration defaultCacheConfig = RedisCacheConfiguration
                .defaultCacheConfig()
                .disableCachingNullValues()
                .entryTtl(Duration.ofMinutes(30)); // Default TTL

        // Configure different TTLs for specific caches
```

```
        Map<String, RedisCacheConfiguration> cacheConfigurations = new HashMap<>
();

        cacheConfigurations.put("products", RedisCacheConfiguration
                .defaultCacheConfig()
                .entryTtl(Duration.ofHours(1)));

        cacheConfigurations.put("categories", RedisCacheConfiguration
                .defaultCacheConfig()
                .entryTtl(Duration.ofHours(24)));

        cacheConfigurations.put("userSession", RedisCacheConfiguration
                .defaultCacheConfig()
                .entryTtl(Duration.ofMinutes(30)));

        return RedisCacheManager.builder(redisConnectionFactory)
                .cacheDefaults(defaultCacheConfig)
                .withInitialCacheConfigurations(cacheConfigurations)
                .transactionAware()
                .build();
    }
}
```

Caffeine for In-Memory Caching

Caffeine is a high-performance in-memory cache:

```
<dependency>
    <groupId>com.github.ben-manes.caffeine</groupId>
    <artifactId>caffeine</artifactId>
</dependency>
```

```
@Configuration
@EnableCaching
public class CaffeineCacheConfig {

    @Bean
    public CacheManager cacheManager() {
        CaffeineCacheManager cacheManager = new CaffeineCacheManager();

        // Default cache specification
        cacheManager.setCacheSpecification("maximumSize=500,expireAfterWrite=5m");

        // Configure specific caches with different specifications
        Map<String, Caffeine<Object, Object>> cacheBuilders = new HashMap<>();

        cacheBuilders.put("products", Caffeine.newBuilder()
                .maximumSize(1000)
                .expireAfterWrite(Duration.ofMinutes(15))
```

```
                .recordStats());

        cacheBuilders.put("categories", Caffeine.newBuilder()
                .maximumSize(100)
                .expireAfterWrite(Duration.ofHours(1))
                .recordStats());

        cacheManager.setCacheBuildersByName(cacheBuilders);

        return cacheManager;
    }

    @Bean
    public CacheMetricsCollector cacheMetricsCollector(CacheManager cacheManager)
{
        CacheMetricsCollector collector = new CacheMetricsCollector();
        collector.bind(cacheManager);
        return collector;
    }
}

@Service
public class ProductService {

    private final ProductRepository productRepository;
    private final Cache<Long, Product> loadingCache;

    public ProductService(ProductRepository productRepository) {
        this.productRepository = productRepository;

        // Create a Caffeine loading cache
        this.loadingCache = Caffeine.newBuilder()
                .maximumSize(10_000)
                .expireAfterWrite(Duration.ofMinutes(5))
                .refreshAfterWrite(Duration.ofMinutes(1))
                .build(this::loadProduct);
    }

    private Product loadProduct(Long id) {
        return productRepository.findById(id)
                .orElseThrow(() -> new ResourceNotFoundException("Product not
found"));
    }

    // Use the loading cache directly
    public Product getProduct(Long id) {
        return loadingCache.get(id);
    }

    public void invalidateProductCache(Long id) {
        loadingCache.invalidate(id);
    }

    public void updateProductInCache(Product product) {
```

```
            loadingCache.put(product.getId(), product);
        }
    }
```

Multi-level Caching

Combine different cache providers for optimal performance:

```
@Configuration
@EnableCaching
public class MultiLevelCacheConfig {

    @Bean
    public CacheManager cacheManager() {
        // First level: Caffeine (in-memory, fast)
        CaffeineCacheManager caffeineCacheManager = new CaffeineCacheManager();

caffeineCacheManager.setCacheSpecification("maximumSize=1000,expireAfterWrite=5m")
;

        // Second level: Redis (distributed, persistent)
        RedisCacheManager redisCacheManager =
RedisCacheManager.builder(redisConnectionFactory())
                .cacheDefaults(RedisCacheConfiguration
                        .defaultCacheConfig()
                        .entryTtl(Duration.ofHours(1)))
                .build();

        // Combine them with ChainedCacheManager
        return new ChainedCacheManager(caffeineCacheManager, redisCacheManager);
    }

    @Bean
    public RedisConnectionFactory redisConnectionFactory() {
        return new LettuceConnectionFactory();
    }
}

// Custom chained cache manager
public class ChainedCacheManager implements CacheManager {

    private final List<CacheManager> cacheManagers;

    public ChainedCacheManager(CacheManager... cacheManagers) {
        this.cacheManagers = Arrays.asList(cacheManagers);
    }

    @Override
    public Cache getCache(String name) {
        // Try to get the cache from each manager in order
        for (CacheManager cacheManager : cacheManagers) {
```

```
            Cache cache = cacheManager.getCache(name);
            if (cache != null) {
                return new ChainedCache(name, cacheManagers);
            }
        }
        return null;
    }

    @Override
    public Collection<String> getCacheNames() {
        Set<String> names = new HashSet<>();
        for (CacheManager cacheManager : cacheManagers) {
            names.addAll(cacheManager.getCacheNames());
        }
        return Collections.unmodifiableSet(names);
    }

    // Custom cache implementation that chains multiple caches
    private static class ChainedCache implements Cache {
        private final String name;
        private final List<CacheManager> cacheManagers;

        public ChainedCache(String name, List<CacheManager> cacheManagers) {
            this.name = name;
            this.cacheManagers = cacheManagers;
        }

        @Override
        public String getName() {
            return name;
        }

        @Override
        public Object getNativeCache() {
            return cacheManagers.stream()
                    .map(cm -> cm.getCache(name))
                    .filter(Objects::nonNull)
                    .map(Cache::getNativeCache)
                    .collect(Collectors.toList());
        }

        @Override
        public ValueWrapper get(Object key) {
            // Try to get from each cache in order
            for (CacheManager cacheManager : cacheManagers) {
                Cache cache = cacheManager.getCache(name);
                if (cache != null) {
                    ValueWrapper wrapper = cache.get(key);
                    if (wrapper != null) {
                        // Found in this cache, populate higher-level caches
                        populateHigherLevelCaches(key, wrapper.get());
                        return wrapper;
                    }
                }
```

```
        }
        return null;
    }

    @Override
    public <T> T get(Object key, Class<T> type) {
        // Try to get from each cache in order
        for (CacheManager cacheManager : cacheManagers) {
            Cache cache = cacheManager.getCache(name);
            if (cache != null) {
                T value = cache.get(key, type);
                if (value != null) {
                    // Found in this cache, populate higher-level caches
                    populateHigherLevelCaches(key, value);
                    return value;
                }
            }
        }
        return null;
    }

    @Override
    public <T> T get(Object key, Callable<T> valueLoader) {
        // Try to get from each cache in order
        T value = null;
        boolean found = false;

        for (CacheManager cacheManager : cacheManagers) {
            Cache cache = cacheManager.getCache(name);
            if (cache != null) {
                if (!found) {
                    ValueWrapper wrapper = cache.get(key);
                    if (wrapper != null) {
                        value = (T) wrapper.get();
                        found = true;
                    }
                }
            }
        }

        if (!found) {
            try {
                value = valueLoader.call();
                put(key, value);
            } catch (Exception e) {
                throw new ValueRetrievalException(key, valueLoader, e);
            }
        }

        return value;
    }

    @Override
    public void put(Object key, Object value) {
```

```
            // Put in all caches
            for (CacheManager cacheManager : cacheManagers) {
                Cache cache = cacheManager.getCache(name);
                if (cache != null) {
                    cache.put(key, value);
                }
            }
        }

        @Override
        public void evict(Object key) {
            // Evict from all caches
            for (CacheManager cacheManager : cacheManagers) {
                Cache cache = cacheManager.getCache(name);
                if (cache != null) {
                    cache.evict(key);
                }
            }
        }

        @Override
        public void clear() {
            // Clear all caches
            for (CacheManager cacheManager : cacheManagers) {
                Cache cache = cacheManager.getCache(name);
                if (cache != null) {
                    cache.clear();
                }
            }
        }

        private void populateHigherLevelCaches(Object key, Object value) {
            boolean found = false;

            // Populate higher level caches (after finding in a lower level)
            for (CacheManager cacheManager : cacheManagers) {
                Cache cache = cacheManager.getCache(name);
                if (cache != null) {
                    if (found) {
                        cache.put(key, value);
                    } else {
                        ValueWrapper wrapper = cache.get(key);
                        if (wrapper != null) {
                            found = true;
                        }
                    }
                }
            }
        }
    }
}
```

Cache Synchronization Strategies

Keep caches in sync with the database:

```
@Service
public class CacheSynchronizationService {

    private final CacheManager cacheManager;
    private final RedisTemplate<String, Object> redisTemplate;

    // Create a topic for cache synchronization
    @Bean
    public RedisMessageListenerContainer redisMessageListenerContainer(
            RedisConnectionFactory connectionFactory) {

        RedisMessageListenerContainer container = new
RedisMessageListenerContainer();
        container.setConnectionFactory(connectionFactory);

        // Listen for cache invalidation messages
        container.addMessageListener(
                new CacheInvalidationListener(cacheManager),
                new ChannelTopic("cache:invalidation"));

        return container;
    }

    // Listen for database changes and invalidate cache
    @TransactionalEventListener(phase = TransactionPhase.AFTER_COMMIT)
    public void handleProductChangedEvent(ProductChangedEvent event) {
        // Invalidate local cache
        Cache productsCache = cacheManager.getCache("products");
        if (productsCache != null) {
            productsCache.evict(event.getProductId());
        }

        // Publish message to other instances
        CacheInvalidationMessage message = new CacheInvalidationMessage(
                "products",
                event.getProductId().toString());

        redisTemplate.convertAndSend("cache:invalidation", message);
    }

    // Cache invalidation message listener
    public static class CacheInvalidationListener implements MessageListener {

        private final CacheManager cacheManager;

        public CacheInvalidationListener(CacheManager cacheManager) {
            this.cacheManager = cacheManager;
        }
```

```
        @Override
        public void onMessage(Message message, byte[] pattern) {
            try {
                String json = new String(message.getBody());
                ObjectMapper mapper = new ObjectMapper();
                CacheInvalidationMessage invalidationMessage =
                        mapper.readValue(json, CacheInvalidationMessage.class);

                // Invalidate the cache
                Cache cache =
cacheManager.getCache(invalidationMessage.getCacheName());
                if (cache != null) {
                    cache.evict(invalidationMessage.getKey());
                }
            } catch (Exception e) {
                // Log exception
            }
        }
    }

    // Message structure
    @Data
    @AllArgsConstructor
    @NoArgsConstructor
    public static class CacheInvalidationMessage {
        private String cacheName;
        private String key;
    }
}

// Entity event listeners
@Component
public class ProductEntityListener {

    private static ApplicationEventPublisher eventPublisher;

    @Autowired
    public void setEventPublisher(ApplicationEventPublisher eventPublisher) {
        ProductEntityListener.eventPublisher = eventPublisher;
    }

    @PostPersist
    @PostUpdate
    @PostRemove
    public void afterChange(Product product) {
        if (eventPublisher != null) {
            eventPublisher.publishEvent(new ProductChangedEvent(product.getId()));
        }
    }
}

// Event class
@Data
```

```
@AllArgsConstructor
public class ProductChangedEvent {
    private final Long productId;
}
```

Cache Aside Pattern

Implement the cache-aside pattern for more control:

```
@Service
public class ProductCacheService {

    private final ProductRepository productRepository;
    private final RedisTemplate<String, Product> redisTemplate;

    private static final String CACHE_KEY_PREFIX = "product:";
    private static final Duration CACHE_TTL = Duration.ofMinutes(30);

    public Product findById(Long id) {
        String cacheKey = CACHE_KEY_PREFIX + id;

        // Try to get from cache first
        Product product = redisTemplate.opsForValue().get(cacheKey);

        if (product == null) {
            // Cache miss - get from database
            product = productRepository.findById(id)
                    .orElseThrow(() -> new ResourceNotFoundException("Product not
found"));

            // Store in cache
            redisTemplate.opsForValue().set(cacheKey, product, CACHE_TTL);
        }

        return product;
    }

    public List<Product> findByCategoryId(Long categoryId) {
        String cacheKey = "category:" + categoryId + ":products";

        // Try to get list from cache
        List<Product> products = (List<Product>)
redisTemplate.opsForValue().get(cacheKey);

        if (products == null) {
            // Cache miss - get from database
            products = productRepository.findByCategoryId(categoryId);

            // Store in cache
            redisTemplate.opsForValue().set(cacheKey, products, CACHE_TTL);
        }
```

```
        return products;
    }

    public Product save(Product product) {
        // Save to database first
        Product savedProduct = productRepository.save(product);

        // Update cache
        String cacheKey = CACHE_KEY_PREFIX + savedProduct.getId();
        redisTemplate.opsForValue().set(cacheKey, savedProduct, CACHE_TTL);

        // Invalidate category cache if needed
        if (product.getCategory() != null) {
            redisTemplate.delete("category:" + product.getCategory().getId() +
":products");
        }

        return savedProduct;
    }

    public void delete(Long id) {
        // Get product to determine category before deletion
        Product product = productRepository.findById(id).orElse(null);

        // Delete from database
        productRepository.deleteById(id);

        // Remove from cache
        redisTemplate.delete(CACHE_KEY_PREFIX + id);

        // Invalidate category cache if needed
        if (product != null && product.getCategory() != null) {
            redisTemplate.delete("category:" + product.getCategory().getId() +
":products");
        }
    }

    // Cache preloading
    @Scheduled(cron = "0 0 * * * *") // Every hour
    public void preloadPopularProducts() {
        // Get most viewed products from analytics
        List<Long> popularProductIds = getPopularProductIds();

        for (Long id : popularProductIds) {
            String cacheKey = CACHE_KEY_PREFIX + id;

            // Only preload if not already in cache
            if (Boolean.FALSE.equals(redisTemplate.hasKey(cacheKey))) {
                productRepository.findById(id).ifPresent(product ->
                        redisTemplate.opsForValue().set(cacheKey, product,
CACHE_TTL));
            }
        }
```

```
    }

    private List<Long> getPopularProductIds() {
        // Implementation to get popular product IDs from analytics
        // This could be another service call or database query
        return Collections.emptyList();
    }
}
```

Best Practices for Caching

1. **Cache the Right Data**: Cache read-heavy, relatively static data.

```
@Service
public class CatalogService {

    // Good candidates for caching:

    @Cacheable("categories")
    public List<Category> getAllCategories() {
        // Categories rarely change
        return categoryRepository.findAll();
    }

    @Cacheable("productDetails")
    public ProductDetailDto getProductDetail(Long id) {
        // Product details are frequently accessed but don't change often
        Product product = productRepository.findById(id)
                .orElseThrow(() -> new ResourceNotFoundException("Product not
found"));

        return productMapper.toDetailDto(product);
    }

    // Bad candidates for caching:

    // Don't cache data that changes frequently
    public int getAvailableStock(Long productId) {
        // Stock levels change frequently
        return inventoryService.getCurrentStockLevel(productId);
    }

    // Don't cache user-specific or sensitive data in shared caches
    public UserProfileDto getUserProfile(Long userId) {
        // User data should be secured
        return userService.getUserProfile(userId);
    }
}
```

2. **Set Appropriate Time-to-Live (TTL)**: Balance freshness and performance.

```
@Configuration
@EnableCaching
public class CacheTtlConfig {

    @Bean
    public CacheManager cacheManager(RedisConnectionFactory
redisConnectionFactory) {
        Map<String, RedisCacheConfiguration> cacheConfigurations = new HashMap<>
();

        // Very static data - long TTL
        cacheConfigurations.put("countries", RedisCacheConfiguration
                .defaultCacheConfig()
                .entryTtl(Duration.ofDays(7))); // Countries rarely change

        // Semi-static data - medium TTL
        cacheConfigurations.put("categories", RedisCacheConfiguration
                .defaultCacheConfig()
                .entryTtl(Duration.ofHours(24))); // Categories change
occasionally

        // More dynamic data - short TTL
        cacheConfigurations.put("products", RedisCacheConfiguration
                .defaultCacheConfig()
                .entryTtl(Duration.ofMinutes(30))); // Products change more
frequently

        // Very dynamic data - very short TTL
        cacheConfigurations.put("productStock", RedisCacheConfiguration
                .defaultCacheConfig()
                .entryTtl(Duration.ofMinutes(1))); // Stock changes rapidly

        return RedisCacheManager.builder(redisConnectionFactory)
                .withInitialCacheConfigurations(cacheConfigurations)
                .build();
    }
}
```

3. **Use Cache Keys Wisely**: Design keys for efficient lookup and management.

```
@Service
public class CacheKeyService {

    // Use simple keys for single objects
    @Cacheable(value = "products", key = "#id")
    public Product findById(Long id) {
        return productRepository.findById(id)
                .orElseThrow(() -> new ResourceNotFoundException("Product not
found"));
    }
```

```
    // Use composite keys for queries with multiple parameters
    @Cacheable(value = "products", key = "'category_' + #categoryId + '_page_' +
#pageable.pageNumber + '_size_' + #pageable.pageSize")
    public Page<Product> findByCategoryId(Long categoryId, Pageable pageable) {
        return productRepository.findByCategoryId(categoryId, pageable);
    }

    // Use SpEL expressions for complex keys
    @Cacheable(value = "productSearch",
               key = "T(com.example.util.CacheKeyGenerator).generateKey('search',
#criteria)")
    public List<Product> searchProducts(ProductSearchCriteria criteria) {
        return productRepository.findByCriteria(criteria);
    }
}

// Helper class for generating cache keys
public class CacheKeyGenerator {

    public static String generateKey(String prefix, ProductSearchCriteria
criteria) {
        StringBuilder key = new StringBuilder(prefix);

        if (criteria.getName() != null) {
            key.append("_name_").append(criteria.getName());
        }

        if (criteria.getCategoryId() != null) {
            key.append("_cat_").append(criteria.getCategoryId());
        }

        if (criteria.getMinPrice() != null) {
            key.append("_minp_").append(criteria.getMinPrice());
        }

        if (criteria.getMaxPrice() != null) {
            key.append("_maxp_").append(criteria.getMaxPrice());
        }

        return key.toString();
    }
}
```

4. **Implement Cache Eviction Strategy**: Keep cache contents fresh.

```
@Service
public class CacheEvictionService {

    private final CacheManager cacheManager;

    // Evict specific entry
```

```
    @CacheEvict(value = "products", key = "#id")
    public void evictProduct(Long id) {
        // Method can be empty, the annotation does the work
    }

    // Evict multiple entries
    @Caching(evict = {
        @CacheEvict(value = "products", key = "#product.id"),
        @CacheEvict(value = "productsByCategory", key = "#product.category.id")
    })
    public void evictRelatedCaches(Product product) {
        // Method can be empty
    }

    // Evict all entries in a cache
    @CacheEvict(value = "products", allEntries = true)
    public void evictAllProducts() {
        // Method can be empty
    }

    // Scheduled cache eviction
    @Scheduled(cron = "0 0 0 * * *") // Midnight every day
    @CacheEvict(value = "products", allEntries = true)
    public void evictCachesAtMidnight() {
        // Method can be empty
    }

    // Programmatic cache eviction
    public void evictCachesForCategory(Long categoryId) {
        // Evict the category
        Cache categoriesCache = cacheManager.getCache("categories");
        if (categoriesCache != null) {
            categoriesCache.evict(categoryId);
        }

        // Evict the category's products list
        Cache productsByCategoryCache =
cacheManager.getCache("productsByCategory");
        if (productsByCategoryCache != null) {
            productsByCategoryCache.evict(categoryId);
        }

        // Evict individual products in this category
        // This requires additional logic to find the products in this category
    }
}
```

5. **Monitor Cache Performance**: Track hit rates and optimize accordingly.

```
@Configuration
public class CacheMonitoringConfig {
```

```
    @Bean
    public CacheMetricsCollector cacheMetricsCollector(CacheManager cacheManager)
{
        CacheMetricsCollector collector = new CacheMetricsCollector();

        // Register caches to collect metrics
        if (cacheManager instanceof ConcurrentMapCacheManager) {
            ConcurrentMapCacheManager cmCacheManager = (ConcurrentMapCacheManager)
cacheManager;
            for (String cacheName : cmCacheManager.getCacheNames()) {
                ConcurrentMapCache cache = (ConcurrentMapCache)
cmCacheManager.getCache(cacheName);
                if (cache != null) {
                    collector.addCache(cacheName, cache.getNativeCache());
                }
            }
        } else if (cacheManager instanceof CaffeineCacheManager) {
            CaffeineCacheManager caffeineCacheManager = (CaffeineCacheManager)
cacheManager;
            for (String cacheName : caffeineCacheManager.getCacheNames()) {
                Cache cache = caffeineCacheManager.getCache(cacheName);
                if (cache != null) {
                    collector.addCache(cacheName, cache.getNativeCache());
                }
            }
        }

        return collector;
    }
}

@Service
@Slf4j
public class CachePerformanceService {

    private final CacheManager cacheManager;
    private final CacheMetrics cacheMetrics;

    // Constructor injection

    // Scheduled task to log cache performance
    @Scheduled(fixedRate = 300000) // Every 5 minutes
    public void logCachePerformance() {
        for (String cacheName : cacheManager.getCacheNames()) {
            CacheStats stats = cacheMetrics.getStats(cacheName);

            if (stats != null) {
                double hitRate = stats.hitRate();
                long hitCount = stats.hitCount();
                long missCount = stats.missCount();
                long evictionCount = stats.evictionCount();

                log.info("Cache '{}' performance - Hit Rate: {}, Hits: {}, Misses:
{}, Evictions: {}",
```

```
                        cacheName, String.format("%.2f", hitRate), hitCount,
missCount, evictionCount);

                // Alert on low hit rates
                if (hitRate < 0.5 && (hitCount + missCount) > 1000) {
                    log.warn("Cache '{}' has a low hit rate ({}). Consider
optimizing.",
                            cacheName, String.format("%.2f", hitRate));
                }

                // Alert on high eviction rates
                if (evictionCount > 1000) {
                    log.warn("Cache '{}' has a high eviction count ({}). Consider
increasing size.",
                            cacheName, evictionCount);
                }
            }
        }
    }

    // Collect and return current cache statistics
    public Map<String, CacheStatistics> getCacheStatistics() {
        Map<String, CacheStatistics> statistics = new HashMap<>();

        for (String cacheName : cacheManager.getCacheNames()) {
            CacheStats stats = cacheMetrics.getStats(cacheName);

            if (stats != null) {
                CacheStatistics cacheStats = new CacheStatistics(
                        stats.hitRate(),
                        stats.hitCount(),
                        stats.missCount(),
                        stats.evictionCount(),
                        stats.loadSuccessCount(),
                        stats.loadFailureCount(),
                        stats.totalLoadTime(),
                        stats.averageLoadPenalty()
                );

                statistics.put(cacheName, cacheStats);
            }
        }

        return statistics;
    }

    // DTO for cache statistics
    @Data
    @AllArgsConstructor
    public static class CacheStatistics {
        private double hitRate;
        private long hitCount;
        private long missCount;
        private long evictionCount;
```

```
        private long loadSuccessCount;
        private long loadFailureCount;
        private long totalLoadTime;
        private double averageLoadPenalty;
    }
}
```

Multiple Data Sources Configuration

In real-world applications, you may need to connect to multiple databases or data sources.

Basic Multiple Data Source Configuration

1. **Configure Two Separate Data Sources**:

```
@Configuration
@EnableTransactionManagement
@EnableJpaRepositories(
    basePackages = "com.example.repository.primary",
    entityManagerFactoryRef = "primaryEntityManagerFactory",
    transactionManagerRef = "primaryTransactionManager"
)
public class PrimaryDataSourceConfig {

    @Primary
    @Bean
    @ConfigurationProperties("spring.datasource.primary")
    public DataSourceProperties primaryDataSourceProperties() {
        return new DataSourceProperties();
    }

    @Primary
    @Bean
    @ConfigurationProperties("spring.datasource.primary.hikari")
    public DataSource primaryDataSource(
            @Qualifier("primaryDataSourceProperties") DataSourceProperties
properties) {
        return properties.initializeDataSourceBuilder()
                .type(HikariDataSource.class)
                .build();
    }

    @Primary
    @Bean
    public LocalContainerEntityManagerFactoryBean primaryEntityManagerFactory(
            EntityManagerFactoryBuilder builder,
            @Qualifier("primaryDataSource") DataSource dataSource) {

        return builder
                .dataSource(dataSource)
                .packages("com.example.entity.primary")
```

```
                .persistenceUnit("primary")
                .properties(jpaProperties())
                .build();
    }

    @Primary
    @Bean
    public PlatformTransactionManager primaryTransactionManager(
            @Qualifier("primaryEntityManagerFactory") EntityManagerFactory
entityManagerFactory) {
        return new JpaTransactionManager(entityManagerFactory);
    }

    private Map<String, Object> jpaProperties() {
        Map<String, Object> props = new HashMap<>();
        props.put("hibernate.hbm2ddl.auto", "update");
        props.put("hibernate.dialect", "org.hibernate.dialect.MySQL8Dialect");
        return props;
    }
}

@Configuration
@EnableTransactionManagement
@EnableJpaRepositories(
    basePackages = "com.example.repository.secondary",
    entityManagerFactoryRef = "secondaryEntityManagerFactory",
    transactionManagerRef = "secondaryTransactionManager"
)
public class SecondaryDataSourceConfig {

    @Bean
    @ConfigurationProperties("spring.datasource.secondary")
    public DataSourceProperties secondaryDataSourceProperties() {
        return new DataSourceProperties();
    }

    @Bean
    @ConfigurationProperties("spring.datasource.secondary.hikari")
    public DataSource secondaryDataSource(
            @Qualifier("secondaryDataSourceProperties") DataSourceProperties
properties) {
        return properties.initializeDataSourceBuilder()
                .type(HikariDataSource.class)
                .build();
    }

    @Bean
    public LocalContainerEntityManagerFactoryBean secondaryEntityManagerFactory(
            EntityManagerFactoryBuilder builder,
            @Qualifier("secondaryDataSource") DataSource dataSource) {

        return builder
                .dataSource(dataSource)
                .packages("com.example.entity.secondary")
```

```
                .persistenceUnit("secondary")
                .properties(jpaProperties())
                .build();
    }

    @Bean
    public PlatformTransactionManager secondaryTransactionManager(
            @Qualifier("secondaryEntityManagerFactory") EntityManagerFactory
entityManagerFactory) {
        return new JpaTransactionManager(entityManagerFactory);
    }

    private Map<String, Object> jpaProperties() {
        Map<String, Object> props = new HashMap<>();
        props.put("hibernate.hbm2ddl.auto", "update");
        props.put("hibernate.dialect", "org.hibernate.dialect.PostgreSQLDialect");
        return props;
    }
}
```

2. **Configure Application Properties**:

```
# Primary Database (MySQL)
spring.datasource.primary.url=jdbc:mysql://localhost:3306/primary_db
spring.datasource.primary.username=root
spring.datasource.primary.password=password
spring.datasource.primary.driver-class-name=com.mysql.cj.jdbc.Driver
spring.datasource.primary.hikari.maximum-pool-size=10
spring.datasource.primary.hikari.minimum-idle=5

# Secondary Database (PostgreSQL)
spring.datasource.secondary.url=jdbc:postgresql://localhost:5432/secondary_db
spring.datasource.secondary.username=postgres
spring.datasource.secondary.password=password
spring.datasource.secondary.driver-class-name=org.postgresql.Driver
spring.datasource.secondary.hikari.maximum-pool-size=5
spring.datasource.secondary.hikari.minimum-idle=2
```

3. **Define Entities for Each Data Source**:

```
// Primary database entity
@Entity
@Table(name = "customers")
public class Customer {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    private String name;
```

```
    private String email;

    // Getters and setters
}

// Secondary database entity
@Entity
@Table(name = "analytics_events")
public class AnalyticsEvent {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    private String eventType;
    private LocalDateTime eventTime;
    private String eventData;

    // Getters and setters
}
```

4. **Create Repositories for Each Data Source**:

```
// Primary database repository
@Repository
public interface CustomerRepository extends JpaRepository<Customer, Long> {
    List<Customer> findByEmail(String email);
}

// Secondary database repository
@Repository
public interface AnalyticsEventRepository extends JpaRepository<AnalyticsEvent,
Long> {
    List<AnalyticsEvent> findByEventType(String eventType);
    List<AnalyticsEvent> findByEventTimeBetween(LocalDateTime start, LocalDateTime
end);
}
```

Routing Data Source for Dynamic Switching

Sometimes you need to switch between data sources dynamically:

```
public class DataSourceContextHolder {

    private static final ThreadLocal<String> contextHolder = new ThreadLocal<>();

    public static void setDataSourceType(String dataSourceType) {
        contextHolder.set(dataSourceType);
    }
```

```
    public static String getDataSourceType() {
        return contextHolder.get();
    }

    public static void clear() {
        contextHolder.remove();
    }
}

public class RoutingDataSource extends AbstractRoutingDataSource {

    @Override
    protected Object determineCurrentLookupKey() {
        return DataSourceContextHolder.getDataSourceType();
    }
}

@Configuration
@EnableTransactionManagement
@EnableJpaRepositories(basePackages = "com.example.repository")
public class RoutingDataSourceConfig {

    @Bean
    @ConfigurationProperties("spring.datasource.master")
    public DataSourceProperties masterDataSourceProperties() {
        return new DataSourceProperties();
    }

    @Bean
    @ConfigurationProperties("spring.datasource.slave")
    public DataSourceProperties slaveDataSourceProperties() {
        return new DataSourceProperties();
    }

    @Bean
    public DataSource masterDataSource(
            @Qualifier("masterDataSourceProperties") DataSourceProperties
properties) {
        return properties.initializeDataSourceBuilder().build();
    }

    @Bean
    public DataSource slaveDataSource(
            @Qualifier("slaveDataSourceProperties") DataSourceProperties
properties) {
        return properties.initializeDataSourceBuilder().build();
    }

    @Primary
    @Bean
    public DataSource routingDataSource(
            @Qualifier("masterDataSource") DataSource masterDataSource,
            @Qualifier("slaveDataSource") DataSource slaveDataSource) {
```

```
        RoutingDataSource routingDataSource = new RoutingDataSource();

        Map<Object, Object> dataSources = new HashMap<>();
        dataSources.put("MASTER", masterDataSource);
        dataSources.put("SLAVE", slaveDataSource);

        routingDataSource.setTargetDataSources(dataSources);
        routingDataSource.setDefaultTargetDataSource(masterDataSource);

        return routingDataSource;
    }

    @Primary
    @Bean
    public LocalContainerEntityManagerFactoryBean entityManagerFactory(
            EntityManagerFactoryBuilder builder,
            @Qualifier("routingDataSource") DataSource dataSource) {

        return builder
                .dataSource(dataSource)
                .packages("com.example.entity")
                .persistenceUnit("default")
                .build();
    }

    @Primary
    @Bean
    public PlatformTransactionManager transactionManager(
            @Qualifier("entityManagerFactory") EntityManagerFactory
entityManagerFactory) {
        return new JpaTransactionManager(entityManagerFactory);
    }
}

@Aspect
@Component
public class ReadOnlyRouteInterceptor {

    @Pointcut("@annotation(com.example.annotation.ReadOnly)")
    public void readOnlyOperation() {}

    @Before("readOnlyOperation()")
    public void setSlaveDataSource() {
        DataSourceContextHolder.setDataSourceType("SLAVE");
    }

    @After("readOnlyOperation()")
    public void clearDataSource() {
        DataSourceContextHolder.clear();
    }
}

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

```
public @interface ReadOnly {
}

@Service
public class CustomerService {

    private final CustomerRepository customerRepository;

    // Constructor injection

    // This uses the master datasource (write operations)
    @Transactional
    public Customer createCustomer(Customer customer) {
        return customerRepository.save(customer);
    }

    // This uses the slave datasource (read operations)
    @ReadOnly
    @Transactional(readOnly = true)
    public List<Customer> getAllCustomers() {
        return customerRepository.findAll();
    }
}
```

Read-Write Splitting

Separate read and write operations to different databases:

```
@Configuration
public class ReadWriteSplittingConfig {

    @Bean
    @ConfigurationProperties("spring.datasource.write")
    public DataSourceProperties writeDataSourceProperties() {
        return new DataSourceProperties();
    }

    @Bean
    @ConfigurationProperties("spring.datasource.read")
    public DataSourceProperties readDataSourceProperties() {
        return new DataSourceProperties();
    }

    @Bean
    public DataSource writeDataSource(
            @Qualifier("writeDataSourceProperties") DataSourceProperties
properties) {
        return properties.initializeDataSourceBuilder().build();
    }

    @Bean
```

```
    public DataSource readDataSource(
            @Qualifier("readDataSourceProperties") DataSourceProperties
properties) {
        return properties.initializeDataSourceBuilder().build();
    }

    @Bean
    public LazyConnectionDataSourceProxy lazyWriteDataSource(
            @Qualifier("writeDataSource") DataSource writeDataSource) {
        return new LazyConnectionDataSourceProxy(writeDataSource);
    }

    @Bean
    public TransactionAwareDataSourceProxy transactionAwareWriteDataSource(
            @Qualifier("lazyWriteDataSource") LazyConnectionDataSourceProxy
lazyWriteDataSource) {
        return new TransactionAwareDataSourceProxy(lazyWriteDataSource);
    }

    @Bean
    public DataSource readWriteDataSource(
            @Qualifier("transactionAwareWriteDataSource") DataSource
writeDataSource,
            @Qualifier("readDataSource") DataSource readDataSource) {

        return new ReadWriteSplittingDataSource(writeDataSource, readDataSource);
    }

    @Primary
    @Bean
    public LocalContainerEntityManagerFactoryBean entityManagerFactory(
            EntityManagerFactoryBuilder builder,
            @Qualifier("readWriteDataSource") DataSource dataSource) {

        return builder
                .dataSource(dataSource)
                .packages("com.example.entity")
                .persistenceUnit("default")
                .build();
    }

    @Primary
    @Bean
    public PlatformTransactionManager transactionManager(
            @Qualifier("entityManagerFactory") EntityManagerFactory
entityManagerFactory) {
        return new JpaTransactionManager(entityManagerFactory);
    }
}

public class ReadWriteSplittingDataSource extends AbstractRoutingDataSource {

    private final DataSource writeDataSource;
    private final DataSource readDataSource;
```

```
    public ReadWriteSplittingDataSource(DataSource writeDataSource, DataSource
readDataSource) {
        this.writeDataSource = writeDataSource;
        this.readDataSource = readDataSource;

        Map<Object, Object> dataSources = new HashMap<>();
        dataSources.put("WRITE", writeDataSource);
        dataSources.put("READ", readDataSource);

        setTargetDataSources(dataSources);
        setDefaultTargetDataSource(writeDataSource);
    }

    @Override
    protected Object determineCurrentLookupKey() {
        // Check if there's an active transaction
        boolean isReadOnly =
TransactionSynchronizationManager.isCurrentTransactionReadOnly();

        return isReadOnly ? "READ" : "WRITE";
    }
}

@Service
public class CustomerService {

    private final CustomerRepository customerRepository;

    // Constructor injection

    // This uses the write datasource (not read-only)
    @Transactional
    public Customer createCustomer(Customer customer) {
        return customerRepository.save(customer);
    }

    // This uses the read datasource (read-only)
    @Transactional(readOnly = true)
    public List<Customer> getAllCustomers() {
        return customerRepository.findAll();
    }
}
```

Sharding with Multiple Data Sources

For high-volume applications, you might need database sharding:

```
@Configuration
@EnableTransactionManagement
@EnableJpaRepositories(basePackages = "com.example.repository")
```

```
public class ShardingDataSourceConfig {

    @Bean
    @ConfigurationProperties("spring.datasource.shard0")
    public DataSourceProperties shard0DataSourceProperties() {
        return new DataSourceProperties();
    }

    @Bean
    @ConfigurationProperties("spring.datasource.shard1")
    public DataSourceProperties shard1DataSourceProperties() {
        return new DataSourceProperties();
    }

    @Bean
    @ConfigurationProperties("spring.datasource.shard2")
    public DataSourceProperties shard2DataSourceProperties() {
        return new DataSourceProperties();
    }

    @Bean
    public DataSource shard0DataSource(
            @Qualifier("shard0DataSourceProperties") DataSourceProperties
properties) {
        return properties.initializeDataSourceBuilder().build();
    }

    @Bean
    public DataSource shard1DataSource(
            @Qualifier("shard1DataSourceProperties") DataSourceProperties
properties) {
        return properties.initializeDataSourceBuilder().build();
    }

    @Bean
    public DataSource shard2DataSource(
            @Qualifier("shard2DataSourceProperties") DataSourceProperties
properties) {
        return properties.initializeDataSourceBuilder().build();
    }

    @Primary
    @Bean
    public ShardingDataSource shardingDataSource(
            @Qualifier("shard0DataSource") DataSource shard0DataSource,
            @Qualifier("shard1DataSource") DataSource shard1DataSource,
            @Qualifier("shard2DataSource") DataSource shard2DataSource) {

        return new ShardingDataSource(
                Map.of(
                        0, shard0DataSource,
                        1, shard1DataSource,
                        2, shard2DataSource
                )
```

```
        );
    }

    @Primary
    @Bean
    public EntityManagerFactory entityManagerFactory(
            @Qualifier("shardingDataSource") DataSource dataSource) {

        LocalContainerEntityManagerFactoryBean emf = new
LocalContainerEntityManagerFactoryBean();
        emf.setDataSource(dataSource);
        emf.setPackagesToScan("com.example.entity");

        JpaVendorAdapter vendorAdapter = new HibernateJpaVendorAdapter();
        emf.setJpaVendorAdapter(vendorAdapter);

        Properties jpaProperties = new Properties();
        jpaProperties.put("hibernate.hbm2ddl.auto", "update");
        jpaProperties.put("hibernate.dialect",
"org.hibernate.dialect.MySQL8Dialect");
        emf.setJpaProperties(jpaProperties);

        emf.afterPropertiesSet();
        return emf.getObject();
    }

    @Primary
    @Bean
    public PlatformTransactionManager transactionManager(
            @Qualifier("entityManagerFactory") EntityManagerFactory
entityManagerFactory) {
        return new JpaTransactionManager(entityManagerFactory);
    }
}

public class ShardingDataSource implements DataSource {

    private final Map<Integer, DataSource> shardDataSources;
    private final ThreadLocal<Integer> currentShard = new ThreadLocal<>();

    public ShardingDataSource(Map<Integer, DataSource> shardDataSources) {
        this.shardDataSources = shardDataSources;
    }

    public void setCurrentShard(Integer shardId) {
        currentShard.set(shardId);
    }

    private DataSource getCurrentDataSource() {
        Integer shardId = currentShard.get();

        if (shardId == null) {
            // Default to shard 0 if not specified
            shardId = 0;
```

```
        }

        DataSource dataSource = shardDataSources.get(shardId);

        if (dataSource == null) {
            throw new IllegalStateException("No data source found for shard ID: "
+ shardId);
        }

        return dataSource;
    }

    @Override
    public Connection getConnection() throws SQLException {
        return getCurrentDataSource().getConnection();
    }

    @Override
    public Connection getConnection(String username, String password) throws
SQLException {
        return getCurrentDataSource().getConnection(username, password);
    }

    // Implement other DataSource methods by delegating to the current data source
}

@Component
public class CustomerShardingStrategy {

    private final ShardingDataSource shardingDataSource;

    public CustomerShardingStrategy(ShardingDataSource shardingDataSource) {
        this.shardingDataSource = shardingDataSource;
    }

    public int calculateShardId(String customerId) {
        // Simple hash-based sharding
        return Math.abs(customerId.hashCode() % 3);
    }

    public void setShardForCustomer(String customerId) {
        int shardId = calculateShardId(customerId);
        shardingDataSource.setCurrentShard(shardId);
    }
}

@Service
public class ShardedCustomerService {

    private final CustomerRepository customerRepository;
    private final CustomerShardingStrategy shardingStrategy;

    // Constructor injection
```

```
    @Transactional
    public Customer findById(String customerId) {
        // Set the current shard based on customer ID
        shardingStrategy.setShardForCustomer(customerId);

        // Query the repository (which will use the current shard)
        return customerRepository.findById(customerId)
                .orElseThrow(() -> new ResourceNotFoundException("Customer not
found"));
    }

    @Transactional
    public Customer createCustomer(Customer customer) {
        // Generate customer ID if not set
        if (customer.getId() == null) {
            customer.setId(UUID.randomUUID().toString());
        }

        // Set the current shard based on customer ID
        shardingStrategy.setShardForCustomer(customer.getId());

        // Save to the current shard
        return customerRepository.save(customer);
    }
}
```

Best Practices for Multiple Data Sources

1. **Use Separate Entity Packages**: Keep entities for different data sources in separate packages.

```
// Primary database entities
package com.example.entity.primary;

@Entity
@Table(name = "customers")
public class Customer {
    // Entity definition
}

// Secondary database entities
package com.example.entity.secondary;

@Entity
@Table(name = "analytics_events")
public class AnalyticsEvent {
    // Entity definition
}
```

2. **Configure Connection Pools Appropriately**: Different data sources may need different pool settings.

```
// Primary database (high-traffic OLTP)
spring.datasource.primary.hikari.maximum-pool-size=20
spring.datasource.primary.hikari.minimum-idle=10
spring.datasource.primary.hikari.connection-timeout=30000
spring.datasource.primary.hikari.idle-timeout=600000

// Secondary database (analytics, reporting)
spring.datasource.secondary.hikari.maximum-pool-size=5
spring.datasource.secondary.hikari.minimum-idle=2
spring.datasource.secondary.hikari.connection-timeout=60000
spring.datasource.secondary.hikari.idle-timeout=1800000
```

3. **Be Careful with Transactions**: Distributed transactions across multiple data sources are complex.

```
@Service
public class OrderService {

    private final OrderRepository orderRepository;
    private final PaymentRepository paymentRepository;
    private final InventoryClient inventoryClient;

    // Instead of using a distributed transaction, use the Saga pattern
    @Transactional
    public Order createOrder(OrderDto orderDto) {
        // Step 1: Create order in the order database
        Order order = new Order();
        // Set order properties
        Order savedOrder = orderRepository.save(order);

        try {
            // Step 2: Create payment in the payment database
            Payment payment = new Payment();
            payment.setOrderId(savedOrder.getId());
            payment.setAmount(orderDto.getAmount());
            paymentRepository.save(payment);

            try {
                // Step 3: Update inventory in the inventory service
                InventoryUpdateRequest request = new InventoryUpdateRequest();
                // Set request properties
                inventoryClient.updateInventory(request);

                // All steps succeeded
                return savedOrder;
            } catch (Exception e) {
                // Compensating transaction for payment
                compensatePayment(payment.getId());
                throw e;
            }
        } catch (Exception e) {
            // Compensating transaction for order
```

```
            compensateOrder(savedOrder.getId());
            throw e;
        }
    }

    private void compensateOrder(Long orderId) {
        // Cancel or mark as failed
        Order order = orderRepository.findById(orderId).orElse(null);
        if (order != null) {
            order.setStatus(OrderStatus.FAILED);
            orderRepository.save(order);
        }
    }

    private void compensatePayment(Long paymentId) {
        // Reverse or cancel payment
        Payment payment = paymentRepository.findById(paymentId).orElse(null);
        if (payment != null) {
            payment.setStatus(PaymentStatus.CANCELLED);
            paymentRepository.save(payment);
        }
    }
}
```

4. **Sync Data When Necessary**: Use synchronization mechanisms for cross-database consistency.

```
@Service
@Slf4j
public class DataSyncService {

    private final CustomerRepository primaryCustomerRepository;
    private final CustomerAnalyticsRepository analyticsCustomerRepository;

    @Scheduled(fixedRate = 300000) // Every 5 minutes
    @Transactional(readOnly = true)
    public void syncCustomers() {
        LocalDateTime lastSyncTime = getLastSyncTime();
        LocalDateTime currentTime = LocalDateTime.now();

        // Get recently updated customers from primary database
        List<Customer> updatedCustomers = primaryCustomerRepository
                .findByUpdatedAtBetween(lastSyncTime, currentTime);

        if (!updatedCustomers.isEmpty()) {
            log.info("Syncing {} customers to analytics database",
updatedCustomers.size());

            List<CustomerAnalytics> analyticsCustomers = updatedCustomers.stream()
                    .map(this::convertToAnalyticsCustomer)
                    .collect(Collectors.toList());

            // Save to analytics database
```

```
            analyticsCustomerRepository.saveAll(analyticsCustomers);

            // Update last sync time
            updateLastSyncTime(currentTime);
        }
    }

    private CustomerAnalytics convertToAnalyticsCustomer(Customer customer) {
        CustomerAnalytics analyticsCustomer = new CustomerAnalytics();
        analyticsCustomer.setCustomerId(customer.getId());
        analyticsCustomer.setName(customer.getName());
        analyticsCustomer.setEmail(customer.getEmail());
        analyticsCustomer.setSegment(determineCustomerSegment(customer));
        analyticsCustomer.setLifetimeValue(calculateLifetimeValue(customer));
        return analyticsCustomer;
    }

    private LocalDateTime getLastSyncTime() {
        // Implementation to get last sync time from a configuration store
        return LocalDateTime.now().minusMinutes(5);
    }

    private void updateLastSyncTime(LocalDateTime time) {
        // Implementation to update last sync time in a configuration store
    }

    private String determineCustomerSegment(Customer customer) {
        // Logic to determine customer segment
        return "REGULAR";
    }

    private BigDecimal calculateLifetimeValue(Customer customer) {
        // Logic to calculate customer lifetime value
        return BigDecimal.ZERO;
    }
}
```

5. **Use Database-Specific Features Wisely**: Leverage the strengths of each database type.

```
@Repository
public interface CustomerRepository extends JpaRepository<Customer, Long> {

    // Uses MySQL's full-text search capabilities
    @Query(value = "SELECT * FROM customers WHERE MATCH(name, email, address)
AGAINST(?1 IN BOOLEAN MODE)",
           nativeQuery = true)
    List<Customer> fullTextSearch(String searchTerm);
}

@Repository
public interface ProductAnalyticsRepository extends
JpaRepository<ProductAnalytics, Long> {
```

```
    // Uses PostgreSQL's analytical functions
    @Query(value = "SELECT p.*, " +
                   "       RANK() OVER (PARTITION BY p.category_id ORDER BY p.sales
DESC) as rank, " +
                   "       PERCENT_RANK() OVER (PARTITION BY p.category_id ORDER BY
p.sales) as percentile " +
                   "FROM product_analytics p " +
                   "WHERE p.category_id = ?1",
           nativeQuery = true)
    List<Object[]> getProductRankingsByCategory(Long categoryId);
}
```

Database Migrations

Database migrations help manage database schema changes in a controlled, version-tracked manner.

Flyway Migration

Flyway is a popular database migration tool that integrates well with Spring Boot:

1. **Add Flyway Dependency**:

```
<dependency>
    <groupId>org.flywaydb</groupId>
    <artifactId>flyway-core</artifactId>
</dependency>

<!-- For MySQL support in newer versions -->
<dependency>
    <groupId>org.flywaydb</groupId>
    <artifactId>flyway-mysql</artifactId>
</dependency>
```

2. **Configure Flyway**:

```
# application.properties

# Enable Flyway
spring.flyway.enabled=true

# Migration scripts location
spring.flyway.locations=classpath:db/migration

# Schema history table name
spring.flyway.table=flyway_schema_history

# Baseline on migrate (for existing databases)
spring.flyway.baseline-on-migrate=true
```

```
spring.flyway.baseline-version=0

# Validate migration checksums
spring.flyway.validate-on-migrate=true

# Clean schema before migration (be careful, only for development)
spring.flyway.clean-disabled=true
```

3. **Create Migration Scripts**:

Migration scripts should be placed in `src/main/resources/db/migration` and follow a specific naming convention:

- `V1__Create_customers_table.sql`
- `V2__Add_address_to_customers.sql`
- `V3__Create_orders_table.sql`

```
-- V1__Create_customers_table.sql
CREATE TABLE customers (
    id BIGINT AUTO_INCREMENT PRIMARY KEY,
    name VARCHAR(100) NOT NULL,
    email VARCHAR(100) NOT NULL UNIQUE,
    created_at TIMESTAMP NOT NULL DEFAULT CURRENT_TIMESTAMP,
    updated_at TIMESTAMP NOT NULL DEFAULT CURRENT_TIMESTAMP ON UPDATE
CURRENT_TIMESTAMP
);

-- V2__Add_address_to_customers.sql
ALTER TABLE customers
ADD COLUMN address VARCHAR(255),
ADD COLUMN city VARCHAR(100),
ADD COLUMN postal_code VARCHAR(20),
ADD COLUMN country VARCHAR(50);

-- V3__Create_orders_table.sql
CREATE TABLE orders (
    id BIGINT AUTO_INCREMENT PRIMARY KEY,
    customer_id BIGINT NOT NULL,
    order_date TIMESTAMP NOT NULL DEFAULT CURRENT_TIMESTAMP,
    total_amount DECIMAL(10, 2) NOT NULL,
    status VARCHAR(20) NOT NULL,
    FOREIGN KEY (customer_id) REFERENCES customers(id)
);

CREATE TABLE order_items (
    id BIGINT AUTO_INCREMENT PRIMARY KEY,
    order_id BIGINT NOT NULL,
    product_id BIGINT NOT NULL,
    quantity INT NOT NULL,
    price DECIMAL(10, 2) NOT NULL,
```

```
    FOREIGN KEY (order_id) REFERENCES orders(id)
);
```

4. **Java-based Migrations**:

For complex migrations, you can use Java-based migration scripts:

```
package db.migration;

import org.flywaydb.core.api.migration.BaseJavaMigration;
import org.flywaydb.core.api.migration.Context;
import org.springframework.jdbc.core.JdbcTemplate;
import org.springframework.jdbc.datasource.SingleConnectionDataSource;

public class V4__Migrate_customer_addresses extends BaseJavaMigration {

    @Override
    public void migrate(Context context) throws Exception {
        JdbcTemplate jdbcTemplate = new JdbcTemplate(
                new SingleConnectionDataSource(context.getConnection(), true));

        // Check if address fields have any data
        Integer count = jdbcTemplate.queryForObject(
                "SELECT COUNT(*) FROM customers WHERE address IS NOT NULL",
Integer.class);

        if (count > 0) {
            // Create new addresses table
            jdbcTemplate.execute("CREATE TABLE addresses (" +
                    "id BIGINT AUTO_INCREMENT PRIMARY KEY, " +
                    "customer_id BIGINT NOT NULL, " +
                    "address_type VARCHAR(20) NOT NULL, " +
                    "address VARCHAR(255), " +
                    "city VARCHAR(100), " +
                    "postal_code VARCHAR(20), " +
                    "country VARCHAR(50), " +
                    "is_default BOOLEAN NOT NULL DEFAULT FALSE, " +
                    "FOREIGN KEY (customer_id) REFERENCES customers(id))");

            // Migrate existing address data
            jdbcTemplate.execute(
                    "INSERT INTO addresses (customer_id, address_type, address,
city, postal_code, country, is_default) " +
                    "SELECT id, 'SHIPPING', address, city, postal_code, country,
TRUE " +
                    "FROM customers WHERE address IS NOT NULL");
        } else {
            // Just create the new table
            jdbcTemplate.execute("CREATE TABLE addresses (" +
                    "id BIGINT AUTO_INCREMENT PRIMARY KEY, " +
                    "customer_id BIGINT NOT NULL, " +
                    "address_type VARCHAR(20) NOT NULL, " +
```

```
                    "address VARCHAR(255), " +
                    "city VARCHAR(100), " +
                    "postal_code VARCHAR(20), " +
                    "country VARCHAR(50), " +
                    "is_default BOOLEAN NOT NULL DEFAULT FALSE, " +
                    "FOREIGN KEY (customer_id) REFERENCES customers(id))");
        }
    }
}
```

5. **Callback Hooks**:

```
@Component
public class FlywayMigrationCallback implements FlywayCallback {

    private static final Logger log =
LoggerFactory.getLogger(FlywayMigrationCallback.class);

    @Override
    public void beforeMigrate(Connection connection) {
        log.info("Starting database migration");
    }

    @Override
    public void afterMigrate(Connection connection) {
        log.info("Database migration completed successfully");

        try (Statement stmt = connection.createStatement()) {
            // Run some post-migration validation
            ResultSet rs = stmt.executeQuery("SELECT COUNT(*) FROM
flyway_schema_history");
            if (rs.next()) {
                int migrationCount = rs.getInt(1);
                log.info("Total migrations executed: {}", migrationCount);
            }
        } catch (SQLException e) {
            log.error("Error during post-migration check", e);
        }
    }

    @Override
    public void beforeEachMigrate(Connection connection, MigrationInfo info) {
        log.info("Executing migration: {} - {}", info.getVersion(),
info.getDescription());
    }

    @Override
    public void afterEachMigrate(Connection connection, MigrationInfo info) {
        log.info("Completed migration: {} - {}", info.getVersion(),
info.getDescription());
    }
```

```
    // Implement other callback methods
}
```

6. **Programmatic Flyway Control**:

```
@Service
public class DatabaseMigrationService {

    private final Flyway flyway;

    public DatabaseMigrationService(Flyway flyway) {
        this.flyway = flyway;
    }

    public void migrateDatabase() {
        // Execute pending migrations
        MigrateResult result = flyway.migrate();

        // Get migration info
        MigrationInfo[] appliedMigrations = result.migrations;
        int migrationsExecuted = result.migrationsExecuted;

        // Print migration summary
        for (MigrationInfo migration : appliedMigrations) {
            System.out.println("Migration: " + migration.getVersion() + " - " +
migration.getDescription());
        }

        System.out.println("Total migrations executed: " + migrationsExecuted);
    }

    public void repairMetadata() {
        // Repair the metadata table
        flyway.repair();
    }

    public void validateMigrations() {
        // Validate applied migrations against available ones
        flyway.validate();
    }

    public MigrationInfo[] getMigrationInfo() {
        // Get information about all migrations
        return flyway.info().all();
    }

    public boolean isClean() {
        return flyway.info().current() == null;
    }

    public void cleanDatabase() {
        // WARNING: This will drop all objects in the schema
```

```
            flyway.clean();
        }
    }
```

Liquibase Migration

Liquibase is another powerful database migration tool:

1. **Add Liquibase Dependency**:

```
<dependency>
    <groupId>org.liquibase</groupId>
    <artifactId>liquibase-core</artifactId>
</dependency>
```

2. **Configure Liquibase**:

```
# application.properties

# Enable Liquibase
spring.liquibase.enabled=true

# Change log configuration
spring.liquibase.change-log=classpath:db/changelog/db.changelog-master.xml

# Set default schema
spring.liquibase.default-schema=public

# Set contexts to use
spring.liquibase.contexts=development

# Set labels to use
spring.liquibase.labels=

# Whether to first drop the database schema
spring.liquibase.drop-first=false
```

3. **Create Changelog Master File**:

`src/main/resources/db/changelog/db.changelog-master.xml`:

```
<?xml version="1.0" encoding="UTF-8"?>
<databaseChangeLog
    xmlns="http://www.liquibase.org/xml/ns/dbchangelog"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://www.liquibase.org/xml/ns/dbchangelog
                        http://www.liquibase.org/xml/ns/dbchangelog/dbchangelog-
```

```
4.5.xsd">

    <!-- Include all changelog files -->
    <include file="db/changelog/changes/01-create-customers-table.xml"/>
    <include file="db/changelog/changes/02-add-address-to-customers.xml"/>
    <include file="db/changelog/changes/03-create-orders-table.xml"/>
    <include file="db/changelog/changes/04-migrate-customer-addresses.xml"/>
</databaseChangeLog>
```

4. **Create Individual Changelog Files**:

src/main/resources/db/changelog/changes/01-create-customers-table.xml:

```
<?xml version="1.0" encoding="UTF-8"?>
<databaseChangeLog
    xmlns="http://www.liquibase.org/xml/ns/dbchangelog"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://www.liquibase.org/xml/ns/dbchangelog
                        http://www.liquibase.org/xml/ns/dbchangelog/dbchangelog-
4.5.xsd">

    <changeSet id="01" author="developer">
        <createTable tableName="customers">
            <column name="id" type="BIGINT" autoIncrement="true">
                <constraints primaryKey="true" nullable="false"/>
            </column>
            <column name="name" type="VARCHAR(100)">
                <constraints nullable="false"/>
            </column>
            <column name="email" type="VARCHAR(100)">
                <constraints nullable="false" unique="true"/>
            </column>
            <column name="created_at" type="TIMESTAMP"
defaultValueComputed="CURRENT_TIMESTAMP">
                <constraints nullable="false"/>
            </column>
            <column name="updated_at" type="TIMESTAMP"
defaultValueComputed="CURRENT_TIMESTAMP">
                <constraints nullable="false"/>
            </column>
        </createTable>

        <createIndex indexName="idx_customers_email" tableName="customers">
            <column name="email"/>
        </createIndex>
    </changeSet>
</databaseChangeLog>
```

src/main/resources/db/changelog/changes/02-add-address-to-customers.xml:

```xml
<?xml version="1.0" encoding="UTF-8"?>
<databaseChangeLog
    xmlns="http://www.liquibase.org/xml/ns/dbchangelog"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://www.liquibase.org/xml/ns/dbchangelog
                        http://www.liquibase.org/xml/ns/dbchangelog/dbchangelog-
4.5.xsd">

    <changeSet id="02" author="developer">
        <addColumn tableName="customers">
            <column name="address" type="VARCHAR(255)"/>
            <column name="city" type="VARCHAR(100)"/>
            <column name="postal_code" type="VARCHAR(20)"/>
            <column name="country" type="VARCHAR(50)"/>
        </addColumn>
    </changeSet>
</databaseChangeLog>
```

5. **XML, YAML, JSON, or SQL Formats**:

Liquibase supports multiple formats. Here's a YAML example:

```yaml
databaseChangeLog:
  - changeSet:
      id: 03
      author: developer
      changes:
        - createTable:
            tableName: orders
            columns:
              - column:
                  name: id
                  type: BIGINT
                  autoIncrement: true
                  constraints:
                    primaryKey: true
                    nullable: false
              - column:
                  name: customer_id
                  type: BIGINT
                  constraints:
                    nullable: false
                    foreignKeyName: fk_orders_customer
                    references: customers(id)
              - column:
                  name: order_date
                  type: TIMESTAMP
                  defaultValueComputed: CURRENT_TIMESTAMP
                  constraints:
                    nullable: false
              - column:
```

```
                name: total_amount
                type: DECIMAL(10, 2)
                constraints:
                  nullable: false
            - column:
                name: status
                type: VARCHAR(20)
                constraints:
                  nullable: false
        - createTable:
            tableName: order_items
            columns:
              - column:
                  name: id
                  type: BIGINT
                  autoIncrement: true
                  constraints:
                    primaryKey: true
                    nullable: false
              - column:
                  name: order_id
                  type: BIGINT
                  constraints:
                    nullable: false
                    foreignKeyName: fk_order_items_order
                    references: orders(id)
              - column:
                  name: product_id
                  type: BIGINT
                  constraints:
                    nullable: false
              - column:
                  name: quantity
                  type: INT
                  constraints:
                    nullable: false
              - column:
                  name: price
                  type: DECIMAL(10, 2)
                  constraints:
                    nullable: false
```

6. **Custom SQL Scripts**:

```
<changeSet id="04" author="developer">
    <sql>
        -- Create addresses table
        CREATE TABLE addresses (
            id BIGINT AUTO_INCREMENT PRIMARY KEY,
            customer_id BIGINT NOT NULL,
            address_type VARCHAR(20) NOT NULL,
            address VARCHAR(255),
```

```
            city VARCHAR(100),
            postal_code VARCHAR(20),
            country VARCHAR(50),
            is_default BOOLEAN NOT NULL DEFAULT FALSE,
            FOREIGN KEY (customer_id) REFERENCES customers(id)
        );

        -- Migrate existing address data
        INSERT INTO addresses (customer_id, address_type, address, city,
postal_code, country, is_default)
        SELECT id, 'SHIPPING', address, city, postal_code, country, TRUE
        FROM customers
        WHERE address IS NOT NULL;
    </sql>
    <rollback>
        DROP TABLE addresses;
    </rollback>
</changeSet>
```

7. **Using Preconditions**:

```
<changeSet id="05" author="developer">
    <preConditions onFail="MARK_RAN">
        <tableExists tableName="customers"/>
        <columnExists tableName="customers" columnName="address"/>
        <sqlCheck expectedResult="1">SELECT COUNT(*) > 0 FROM customers WHERE
address IS NOT NULL</sqlCheck>
    </preConditions>

    <!-- Migration steps here -->
</changeSet>
```

8. **Adding Initial Data**:

```
<changeSet id="06" author="developer">
    <loadData
        file="db/data/countries.csv"
        tableName="countries">
        <column name="code" type="STRING"/>
        <column name="name" type="STRING"/>
    </loadData>

    <insert tableName="roles">
        <column name="name" value="ROLE_ADMIN"/>
        <column name="description" value="Administrator role"/>
    </insert>

    <insert tableName="roles">
        <column name="name" value="ROLE_USER"/>
```

```
        <column name="description" value="Regular user role"/>
    </insert>
</changeSet>
```

Best Practices for Database Migrations

1. **Make Migrations Repeatable and Idempotent**:

```
-- V1__Create_index_if_not_exists.sql
CREATE INDEX IF NOT EXISTS idx_customers_email ON customers (email);
```

```
<changeSet id="01" author="developer">
    <preConditions onFail="MARK_RAN">
        <not>
            <indexExists indexName="idx_customers_email"/>
        </not>
    </preConditions>

    <createIndex indexName="idx_customers_email" tableName="customers">
        <column name="email"/>
    </createIndex>
</changeSet>
```

2. **Include Rollback Scripts**:

```
-- V1__Create_customers_table.sql
CREATE TABLE customers (
    id BIGINT AUTO_INCREMENT PRIMARY KEY,
    name VARCHAR(100) NOT NULL,
    email VARCHAR(100) NOT NULL UNIQUE
);

-- R__1__Create_customers_table.sql (rollback)
DROP TABLE IF EXISTS customers;
```

```
<changeSet id="01" author="developer">
    <createTable tableName="customers">
        <!-- Table definition -->
    </createTable>

    <rollback>
        <dropTable tableName="customers"/>
    </rollback>
</changeSet>
```

3. **Use Context-Based Migrations**:

```
# Development profile
spring.flyway.contexts=dev,common

# Production profile
spring.flyway.contexts=prod,common
```

```
-- V1__Create_test_data.sql
-- @context dev
INSERT INTO customers (name, email) VALUES ('Test User', 'test@example.com');
```

```
<changeSet id="01" author="developer" context="dev">
    <insert tableName="customers">
        <column name="name" value="Test User"/>
        <column name="email" value="test@example.com"/>
    </insert>
</changeSet>

<changeSet id="02" author="developer" context="prod">
    <insert tableName="configuration">
        <column name="key" value="maintenance_mode"/>
        <column name="value" value="false"/>
    </insert>
</changeSet>

<changeSet id="03" author="developer" context="common">
    <!-- Common changes for all environments -->
</changeSet>
```

4. **Break Down Complex Migrations**:

```
<!-- Instead of one large changeset -->
<changeSet id="01" author="developer">
    <!-- Hundreds of schema changes here -->
</changeSet>

<!-- Better approach: split into smaller, focused changesets -->
<changeSet id="01-1" author="developer">
    <!-- Create tables -->
</changeSet>

<changeSet id="01-2" author="developer">
    <!-- Create foreign keys -->
</changeSet>
```

```
<changeSet id="01-3" author="developer">
    <!-- Create indexes -->
</changeSet>
```

5. **Use Tags for Release Management**:

```
@Service
public class DatabaseReleaseService {

    private final Flyway flyway;

    // Constructor injection

    @Transactional
    public void tagCurrentVersion(String tag) {
        // Tag the current database state
        flyway.getConfiguration()
              .getFluentConfiguration()
              .baselineVersion("1.0")
              .baselineDescription("Base version")
              .validateMigrationNaming(true)
              .load()
              .tag(tag);
    }

    @Transactional
    public void migrateToTag(String tag) {
        // Migrate up to a specific tag
        flyway.getConfiguration()
              .getFluentConfiguration()
              .target(tag)
              .load()
              .migrate();
    }
}
```

6. **Test Migrations Before Deployment**:

```
@SpringBootTest
@TestPropertySource(properties = {
    "spring.flyway.enabled=false"  // Disable auto-migration
})
public class DatabaseMigrationTest {

    @Autowired
    private DataSource dataSource;

    @Test
```

```
    public void testDatabaseMigration() {
        // Create a new Flyway instance with a clean database
        Flyway flyway = Flyway.configure()
                .dataSource(dataSource)
                .cleanDisabled(false)  // Enable clean for testing
                .locations("classpath:db/migration")
                .load();

        // Clean the database
        flyway.clean();

        // Execute migrations
        MigrateResult result = flyway.migrate();

        // Verify migrations were applied
        assertEquals(5, result.migrationsExecuted);

        // Verify specific database objects exist
        JdbcTemplate jdbcTemplate = new JdbcTemplate(dataSource);

        Integer customerTableExists = jdbcTemplate.queryForObject(
                "SELECT COUNT(*) FROM information_schema.tables WHERE table_name =
'customers'",
                Integer.class);
        assertEquals(1, customerTableExists);

        Integer orderTableExists = jdbcTemplate.queryForObject(
                "SELECT COUNT(*) FROM information_schema.tables WHERE table_name =
'orders'",
                Integer.class);
        assertEquals(1, orderTableExists);
    }
}
```

7. **Document Migrations**:

```
-- V1__Add_customer_status.sql
-- Description: Add customer status column with default 'ACTIVE' value
-- Author: John Doe
-- Date: 2023-05-15
-- Jira Ticket: CRM-123

-- Add status column with default value
ALTER TABLE customers ADD COLUMN status VARCHAR(20) NOT NULL DEFAULT 'ACTIVE';

-- Create index on status column
CREATE INDEX idx_customers_status ON customers (status);

-- Update existing inactive customers
UPDATE customers SET status = 'INACTIVE' WHERE last_activity_date < NOW() -
INTERVAL 1 YEAR;
```

D. Security Implementation

Spring Security Architecture

Spring Security is a powerful and highly customizable authentication and access-control framework.

Core Components

Spring Security is built around several key components that work together to secure your application:

1. **SecurityContextHolder**: Stores details of the authenticated user, making security information accessible to the application.
2. **Authentication**: Represents the current user in the system, including credentials, authorities, and authentication status.
3. **AuthenticationManager**: Handles the authentication process.
4. **AuthenticationProvider**: Validates specific types of authentication.
5. **UserDetailsService**: Loads user-specific data during authentication.
6. **PasswordEncoder**: Handles password hashing and validation.
7. **AccessDecisionManager**: Makes authorization decisions for secured resources.
8. **FilterChain**: A series of security filters that process the request.

Basic Configuration

A simple security configuration for a Spring Boot application:

```
@Configuration
@EnableWebSecurity
public class SecurityConfig {

    @Bean
    public SecurityFilterChain securityFilterChain(HttpSecurity http) throws
Exception {
        http
            .authorizeHttpRequests(authorize -> authorize
                .requestMatchers("/", "/home", "/public/**").permitAll()
                .requestMatchers("/api/admin/**").hasRole("ADMIN")
                .requestMatchers("/api/user/**").hasRole("USER")
                .anyRequest().authenticated()
            )
            .formLogin(form -> form
                .loginPage("/login")
                .defaultSuccessUrl("/dashboard")
                .failureUrl("/login?error=true")
                .permitAll()
            )
            .logout(logout -> logout
                .logoutUrl("/logout")
                .logoutSuccessUrl("/login?logout=true")
                .invalidateHttpSession(true)
```

```
                .deleteCookies("JSESSIONID")
                .permitAll()
            )
            .rememberMe(remember -> remember
                .key("uniqueAndSecretKey")
                .tokenValiditySeconds(86400) // 1 day
            );

        return http.build();
    }

    @Bean
    public PasswordEncoder passwordEncoder() {
        return new BCryptPasswordEncoder();
    }
}
```

User Management

Configuring user details for authentication:

```
@Configuration
public class UserManagementConfig {

    @Bean
    public UserDetailsService userDetailsService(PasswordEncoder passwordEncoder)
{
        // In-memory user details for demonstration
        InMemoryUserDetailsManager manager = new InMemoryUserDetailsManager();

        UserDetails user = User.withUsername("user")
                .password(passwordEncoder.encode("password"))
                .roles("USER")
                .build();

        UserDetails admin = User.withUsername("admin")
                .password(passwordEncoder.encode("admin"))
                .roles("ADMIN", "USER")  // Admin also has USER role
                .build();

        manager.createUser(user);
        manager.createUser(admin);

        return manager;
    }
}
```

Real-world applications typically use a database for user management:

```java
@Entity
@Table(name = "users")
public class User {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    @Column(nullable = false, unique = true)
    private String username;

    @Column(nullable = false)
    private String password;

    @Column(nullable = false)
    private boolean enabled = true;

    @ManyToMany(fetch = FetchType.EAGER)
    @JoinTable(
        name = "user_roles",
        joinColumns = @JoinColumn(name = "user_id"),
        inverseJoinColumns = @JoinColumn(name = "role_id")
    )
    private Set<Role> roles = new HashSet<>();

    // Getters and setters
}

@Entity
@Table(name = "roles")
public class Role {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    @Column(nullable = false, unique = true)
    private String name;

    // Getters and setters
}

public interface UserRepository extends JpaRepository<User, Long> {
    Optional<User> findByUsername(String username);
}

public interface RoleRepository extends JpaRepository<Role, Long> {
    Optional<Role> findByName(String name);
}

@Service
public class UserService implements UserDetailsService {
```

```
    private final UserRepository userRepository;
    private final PasswordEncoder passwordEncoder;

    // Constructor injection

    @Override
    public UserDetails loadUserByUsername(String username) throws
UsernameNotFoundException {
        User user = userRepository.findByUsername(username)
                .orElseThrow(() -> new UsernameNotFoundException("User not found:
" + username));

        return org.springframework.security.core.userdetails.User
                .withUsername(user.getUsername())
                .password(user.getPassword())
                .authorities(getAuthorities(user))
                .accountExpired(!user.isEnabled())
                .accountLocked(!user.isEnabled())
                .credentialsExpired(!user.isEnabled())
                .disabled(!user.isEnabled())
                .build();
    }

    private Collection<? extends GrantedAuthority> getAuthorities(User user) {
        return user.getRoles().stream()
                .map(role -> new SimpleGrantedAuthority(role.getName()))
                .collect(Collectors.toList());
    }

    @Transactional
    public User createUser(String username, String password, Set<String>
roleNames) {
        // Check if username already exists
        if (userRepository.findByUsername(username).isPresent()) {
            throw new UsernameAlreadyExistsException("Username already exists: " +
username);
        }

        // Create new user
        User user = new User();
        user.setUsername(username);
        user.setPassword(passwordEncoder.encode(password));

        // Assign roles
        Set<Role> roles = roleNames.stream()
                .map(roleName -> roleRepository.findByName(roleName)
                        .orElseThrow(() -> new RoleNotFoundException("Role not
found: " + roleName)))
                .collect(Collectors.toSet());

        user.setRoles(roles);

        return userRepository.save(user);
```

```
    }
}
```

Custom Authentication Provider

For custom authentication mechanisms:

```
@Component
public class CustomAuthenticationProvider implements AuthenticationProvider {

    private final UserDetailsService userDetailsService;
    private final PasswordEncoder passwordEncoder;

    // Constructor injection

    @Override
    public Authentication authenticate(Authentication authentication) throws
AuthenticationException {
        String username = authentication.getName();
        String password = authentication.getCredentials().toString();

        UserDetails userDetails = userDetailsService.loadUserByUsername(username);

        if (passwordEncoder.matches(password, userDetails.getPassword())) {
            // Authentication successful
            List<GrantedAuthority> authorities = new ArrayList<>
(userDetails.getAuthorities());

            // Add any additional runtime authorities if needed

            return new UsernamePasswordAuthenticationToken(
                    userDetails,
                    null,  // Credentials are erased after authentication
                    authorities
            );
        } else {
            throw new BadCredentialsException("Invalid password");
        }
    }

    @Override
    public boolean supports(Class<?> authentication) {
        return
UsernamePasswordAuthenticationToken.class.isAssignableFrom(authentication);
    }
}

@Configuration
public class AuthenticationConfig {

    private final CustomAuthenticationProvider authenticationProvider;
```

```
    // Constructor injection

    @Bean
    public AuthenticationManager authenticationManager(AuthenticationConfiguration
config) throws Exception {
        return config.getAuthenticationManager();
    }

    @Autowired
    public void configureGlobal(AuthenticationManagerBuilder auth) {
        auth.authenticationProvider(authenticationProvider);
    }
}
```

Filter Chain

The security filter chain processes security-related operations in a specific order:

```
@Configuration
@EnableWebSecurity
public class SecurityConfig {

    @Bean
    public SecurityFilterChain securityFilterChain(HttpSecurity http) throws
Exception {
        http
            // Add custom filter before standard filter
            .addFilterBefore(
                new CustomRequestHeaderAuthenticationFilter(),
                UsernamePasswordAuthenticationFilter.class
            )
            // Add custom filter after standard filter
            .addFilterAfter(
                new AuditLogFilter(),
                LogoutFilter.class
            )
            // Add custom filter at a specific position
            .addFilterAt(
                new ConcurrentSessionFilter(),
                ConcurrentSessionFilter.class
            );

        return http.build();
    }
}

// Custom authentication filter
public class CustomRequestHeaderAuthenticationFilter extends OncePerRequestFilter
{
```

```
    @Override
    protected void doFilterInternal(
            HttpServletRequest request,
            HttpServletResponse response,
            FilterChain filterChain) throws ServletException, IOException {

        String apiKey = request.getHeader("X-API-KEY");

        if (apiKey != null) {
            // Validate API key
            try {
                ApiKeyAuthenticationToken auth = new
ApiKeyAuthenticationToken(apiKey);
                SecurityContextHolder.getContext().setAuthentication(auth);
            } catch (AuthenticationException e) {
                // Log authentication failure
            }
        }

        filterChain.doFilter(request, response);
    }
}

// Custom audit log filter
public class AuditLogFilter extends OncePerRequestFilter {

    @Override
    protected void doFilterInternal(
            HttpServletRequest request,
            HttpServletResponse response,
            FilterChain filterChain) throws ServletException, IOException {

        long startTime = System.currentTimeMillis();

        // Continue with the filter chain
        filterChain.doFilter(request, response);

        // Log after the request is processed
        long executionTime = System.currentTimeMillis() - startTime;

        Authentication auth =
SecurityContextHolder.getContext().getAuthentication();
        String username = auth != null ? auth.getName() : "anonymous";

        log.info("Request: {} {} - User: {} - Status: {} - Execution Time: {} ms",
                request.getMethod(),
                request.getRequestURI(),
                username,
                response.getStatus(),
                executionTime);
    }
}
```

Authentication Event Handling

React to authentication-related events:

```
@Component
public class AuthenticationEventListener {

    private final Logger log =
LoggerFactory.getLogger(AuthenticationEventListener.class);
    private final LoginAttemptService loginAttemptService;
    private final UserService userService;

    // Constructor injection

    @EventListener
    public void onSuccess(AuthenticationSuccessEvent event) {
        // Get authentication details
        Authentication auth = event.getAuthentication();

        if (auth instanceof UsernamePasswordAuthenticationToken) {
            String username = auth.getName();
            log.info("Login successful for user: {}", username);

            // Reset failed attempt counter
            loginAttemptService.resetFailedAttempts(username);

            // Update last login timestamp
            userService.updateLastLoginTime(username);
        }
    }

    @EventListener
    public void onFailure(AuthenticationFailureBadCredentialsEvent event) {
        String username = event.getAuthentication().getName();
        log.warn("Failed login attempt for user: {}", username);

        // Record failed login attempt
        loginAttemptService.recordFailedAttempt(username);

        // Check if account should be locked
        if (loginAttemptService.getFailedAttempts(username) >= 5) {
            log.warn("Locking account due to too many failed attempts: {}",
username);
            userService.lockAccount(username);
        }
    }

    @EventListener
    public void onLogout(LogoutSuccessEvent event) {
        String username = event.getAuthentication().getName();
        log.info("User logged out: {}", username);
```

```
        // Perform any logout-related cleanup
    }
}
```

Exception Handling

Custom security exception handling:

```
@Configuration
public class SecurityExceptionConfig {

    @Bean
    public AccessDeniedHandler accessDeniedHandler() {
        return (request, response, accessDeniedException) -> {
            // Log the access denied exception
            log.warn("Access denied for user: {}",

Optional.ofNullable(SecurityContextHolder.getContext().getAuthentication())
                            .map(Authentication::getName)
                            .orElse("unknown"));

            // For API requests, return JSON response
            if (request.getHeader("Accept").contains("application/json")) {
                response.setStatus(HttpStatus.FORBIDDEN.value());
                response.setContentType(MediaType.APPLICATION_JSON_VALUE);

                ObjectMapper mapper = new ObjectMapper();
                mapper.writeValue(response.getWriter(), Map.of(
                        "error", "Access denied",
                        "message", "You don't have permission to access this
resource",
                        "status", 403,
                        "path", request.getRequestURI()
                ));
            } else {
                // For web requests, redirect to access denied page
                response.sendRedirect("/access-denied");
            }
        };
    }

    @Bean
    public AuthenticationEntryPoint authenticationEntryPoint() {
        return (request, response, authException) -> {
            // Log the authentication exception
            log.warn("Authentication failed: {}", authException.getMessage());

            // For API requests, return JSON response
            if (request.getHeader("Accept").contains("application/json")) {
                response.setStatus(HttpStatus.UNAUTHORIZED.value());
                response.setContentType(MediaType.APPLICATION_JSON_VALUE);
```

```
                ObjectMapper mapper = new ObjectMapper();
                mapper.writeValue(response.getWriter(), Map.of(
                        "error", "Unauthorized",
                        "message", "Authentication required to access this
resource",
                        "status", 401,
                        "path", request.getRequestURI()
                ));
            } else {
                // For web requests, redirect to login page
                response.sendRedirect("/login?unauthorized=true");
            }
        };
    }

    @Bean
    public SecurityFilterChain securityFilterChain(HttpSecurity http) throws
Exception {
        http
            // Configure exception handling
            .exceptionHandling(exceptions -> exceptions
                .accessDeniedHandler(accessDeniedHandler())
                .authenticationEntryPoint(authenticationEntryPoint())
            );

        return http.build();
    }
}
```

Spring Boot Security Auto-configuration

Spring Boot automatically configures basic security:

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

When Spring Boot detects Spring Security in the classpath, it automatically:

1. Creates a default `UserDetailsService` with a default user
2. Enables protection against CSRF attacks
3. Creates login and logout endpoints
4. Secures all endpoints that don't have explicit permissions

Override these defaults with custom configuration:

```
# application.properties

# Default user for basic authentication
spring.security.user.name=admin
spring.security.user.password=adminpass
spring.security.user.roles=ADMIN

# Disable auto-configuration (use custom configuration instead)
spring.security.auto-configure.disable=true
```

Security Architecture Best Practices

1. **Design Defense in Depth**:

```
@Configuration
@EnableWebSecurity
public class SecurityConfig {

    @Bean
    public SecurityFilterChain securityFilterChain(HttpSecurity http) throws
Exception {
        http
            // Protection against common web vulnerabilities
            .headers(headers -> headers
                .contentSecurityPolicy(csp -> csp
                    .policyDirectives("default-src 'self'; script-src 'self'
https://trusted.cdn.com")
                )
                .frameOptions().deny()
                .xssProtection().block(true)
            )
            .csrf()
            // Use same-site cookies
            .csrfTokenRepository(CookieCsrfTokenRepository.withHttpOnlyFalse())
            .and()
            // Use HTTPS for everything
            .requiresChannel().anyRequest().requiresSecure()
            .and()
            // Prevent session fixation
            .sessionManagement(session -> session
                .sessionFixation().migrateSession()
                .maximumSessions(1)
                .maxSessionsPreventsLogin(true)
            );

        return http.build();
    }
}
```

2. **Isolate Security Code**:

```
// Keep security code in dedicated modules/packages
package com.example.security;

@Configuration
public class WebSecurityConfig {
    // Security configuration
}

@Configuration
public class AuthenticationConfig {
    // Authentication configuration
}

@Configuration
public class AuthorizationConfig {
    // Authorization configuration
}
```

3. **Create Security Services for Reusable Logic**:

```
@Service
public class SecurityService {

    private final PasswordEncoder passwordEncoder;
    private final TokenProvider tokenProvider;

    // Constructor injection

    public String hashPassword(String plainPassword) {
        return passwordEncoder.encode(plainPassword);
    }

    public boolean validatePassword(String plainPassword, String hashedPassword) {
        return passwordEncoder.matches(plainPassword, hashedPassword);
    }

    public String generateToken(Authentication authentication) {
        return tokenProvider.createToken(authentication);
    }

    public boolean validateToken(String token) {
        return tokenProvider.validateToken(token);
    }

    public Authentication getAuthentication(String token) {
        return tokenProvider.getAuthentication(token);
    }
}
```

4. **Implement Proper Logging**:

```
@Component
public class SecurityLogger {

    private final Logger log = LoggerFactory.getLogger(SecurityLogger.class);

    public void logAuthenticationSuccess(String username, String source) {
        log.info("Authentication success - User: {} - Source: {} - Time: {}",
                username, source, LocalDateTime.now());
    }

    public void logAuthenticationFailure(String username, String source, String
reason) {
        log.warn("Authentication failure - User: {} - Source: {} - Reason: {} -
Time: {}",
                username, source, reason, LocalDateTime.now());
    }

    public void logAccessDenied(String username, String resource) {
        log.warn("Access denied - User: {} - Resource: {} - Time: {}",
                username, resource, LocalDateTime.now());
    }

    public void logSecurityEvent(String eventType, String description) {
        log.info("Security event - Type: {} - Description: {} - Time: {}",
                eventType, description, LocalDateTime.now());
    }
}
```

5. **Use Role-Based Authorization with Granular Permissions**:

```
@Entity
@Table(name = "authorities")
public class Authority {

    @Id
    private String name;

    // Getters and setters
}

@Entity
@Table(name = "roles")
public class Role {

    @Id
    private String name;
```

```
    @ManyToMany(fetch = FetchType.EAGER)
    @JoinTable(
        name = "role_authorities",
        joinColumns = @JoinColumn(name = "role_name"),
        inverseJoinColumns = @JoinColumn(name = "authority_name")
    )
    private Set<Authority> authorities = new HashSet<>();

    // Getters and setters
}

@Configuration
@EnableMethodSecurity
public class MethodSecurityConfig {

    @Bean
    public MethodSecurityExpressionHandler methodSecurityExpressionHandler(
            PermissionEvaluator permissionEvaluator) {

        DefaultMethodSecurityExpressionHandler expressionHandler =
                new DefaultMethodSecurityExpressionHandler();
        expressionHandler.setPermissionEvaluator(permissionEvaluator);
        return expressionHandler;
    }

    @Bean
    public PermissionEvaluator permissionEvaluator() {
        return new DomainPermissionEvaluator();
    }
}

@Component
public class DomainPermissionEvaluator implements PermissionEvaluator {

    @Override
    public boolean hasPermission(Authentication auth, Object targetDomainObject,
Object permission) {
        if (auth == null || targetDomainObject == null || permission == null) {
            return false;
        }

        // Check if user has required permission on the domain object
        String targetType =
targetDomainObject.getClass().getSimpleName().toUpperCase();

        return hasPermission(auth, extractIdFromDomainObject(targetDomainObject),
                targetType, permission);
    }

    @Override
    public boolean hasPermission(Authentication auth, Serializable targetId,
                                 String targetType, Object permission) {
        if (auth == null || targetType == null || permission == null) {
            return false;
```

```
        }

        // Check user's authorities against required permission
        String requiredPermission = targetType + "_" +
permission.toString().toUpperCase();

        return auth.getAuthorities().stream()
                .map(GrantedAuthority::getAuthority)
                .anyMatch(authority -> authority.equals(requiredPermission));
    }

    private Serializable extractIdFromDomainObject(Object domainObject) {
        // Extract ID from domain object using reflection
        try {
            Method getIdMethod = domainObject.getClass().getMethod("getId");
            return (Serializable) getIdMethod.invoke(domainObject);
        } catch (Exception e) {
            throw new IllegalArgumentException("Domain object must have getId()
method");
        }
    }
}
```

Authentication Mechanisms

Spring Security supports various authentication mechanisms for different use cases.

Form-Based Authentication

Traditional form-based login:

```
@Configuration
@EnableWebSecurity
public class FormAuthenticationConfig {

    @Bean
    public SecurityFilterChain securityFilterChain(HttpSecurity http) throws
Exception {
        http
            .authorizeHttpRequests(authorize -> authorize
                .requestMatchers("/login", "/register", "/css/**",
"/js/**").permitAll()
                .anyRequest().authenticated()
            )
            .formLogin(form -> form
                .loginPage("/login")                          // Custom login page
                .loginProcessingUrl("/authenticate")          // Form submission
URL
                .usernameParameter("username")                // Username parameter
name
                .passwordParameter("password")                // Password parameter
```

```
name
                .defaultSuccessUrl("/dashboard", true)       // Redirect on
success
                .failureUrl("/login?error=true")             // Redirect on
failure
                .successHandler(authenticationSuccessHandler()) // Custom success
handler
                .failureHandler(authenticationFailureHandler()) // Custom failure
handler
            )
            .rememberMe(remember -> remember
                .key("unique-and-secret")                    // Secret key for
token
                .tokenValiditySeconds(86400)                 // Token validity (1
day)
                .rememberMeParameter("remember-me")          // Checkbox parameter
name
                .rememberMeCookieName("remember-me-cookie")  // Cookie name
                .tokenRepository(persistentTokenRepository()) // Store tokens in
database
            )
            .logout(logout -> logout
                .logoutUrl("/logout")                        // Logout URL
                .logoutSuccessUrl("/login?logout=true")      // Redirect after
logout
                .invalidateHttpSession(true)                 // Invalidate session
                .deleteCookies("JSESSIONID", "remember-me-cookie") // Clean up
cookies
            );

        return http.build();
    }

    @Bean
    public PersistentTokenRepository persistentTokenRepository() {
        JdbcTokenRepositoryImpl tokenRepository = new JdbcTokenRepositoryImpl();
        tokenRepository.setDataSource(dataSource);
        // tokenRepository.setCreateTableOnStartup(true); // Create table if not
exists
        return tokenRepository;
    }

    @Bean
    public AuthenticationSuccessHandler authenticationSuccessHandler() {
        return (request, response, authentication) -> {
            // Get user details
            UserDetails userDetails = (UserDetails) authentication.getPrincipal();

            // Log successful login
            log.info("User {} logged in successfully", userDetails.getUsername());

            // Update last login time in database
            userService.updateLastLoginTime(userDetails.getUsername());
```

```
            // Determine where to redirect based on user role
            if (userDetails.getAuthorities().stream()
                    .anyMatch(a -> a.getAuthority().equals("ROLE_ADMIN"))) {
                response.sendRedirect("/admin/dashboard");
            } else {
                response.sendRedirect("/dashboard");
            }
        };
    }

    @Bean
    public AuthenticationFailureHandler authenticationFailureHandler() {
        return (request, response, exception) -> {
            // Get attempted username
            String username = request.getParameter("username");

            // Log failed login
            log.warn("Failed login attempt for user: {}", username);

            // Record failed attempt in database
            loginAttemptService.recordFailedAttempt(username);

            // Add error message based on exception type
            String errorMessage;
            if (exception instanceof BadCredentialsException) {
                errorMessage = "Invalid username or password";
            } else if (exception instanceof LockedException) {
                errorMessage = "Account is locked";
            } else if (exception instanceof DisabledException) {
                errorMessage = "Account is disabled";
            } else if (exception instanceof AccountExpiredException) {
                errorMessage = "Account has expired";
            } else if (exception instanceof CredentialsExpiredException) {
                errorMessage = "Credentials have expired";
            } else {
                errorMessage = "Authentication failed";
            }

            // Redirect to login page with error message
            response.sendRedirect("/login?error=true&message=" +
URLEncoder.encode(errorMessage, "UTF-8"));
        };
    }
}
```

Basic Authentication

HTTP Basic authentication for API access:

```
@Configuration
@EnableWebSecurity
```

```
public class BasicAuthConfig {

    @Bean
    public SecurityFilterChain apiSecurityFilterChain(HttpSecurity http) throws
Exception {
        http
            .securityMatcher("/api/**")  // Apply only to API endpoints
            .authorizeHttpRequests(authorize -> authorize
                .requestMatchers("/api/public/**").permitAll()
                .anyRequest().authenticated()
            )
            .httpBasic(Customizer.withDefaults())  // Enable Basic Authentication
            .sessionManagement(session -> session
                .sessionCreationPolicy(SessionCreationPolicy.STATELESS)  // No
sessions for API
            );

        return http.build();
    }
}
```

JWT (JSON Web Token) Authentication

Using JWT for stateless authentication:

```
@Configuration
@EnableWebSecurity
public class JwtSecurityConfig {

    private final JwtTokenProvider tokenProvider;
    private final JwtAuthenticationEntryPoint authenticationEntryPoint;

    // Constructor injection

    @Bean
    public SecurityFilterChain jwtSecurityFilterChain(HttpSecurity http) throws
Exception {
        http
            .csrf().disable()  // CSRF not needed for JWT as it's stateless
            .authorizeHttpRequests(authorize -> authorize
                .requestMatchers("/api/auth/**").permitAll()
                .anyRequest().authenticated()
            )
            .exceptionHandling(exceptions -> exceptions
                .authenticationEntryPoint(authenticationEntryPoint)
            )
            .sessionManagement(session -> session
                .sessionCreationPolicy(SessionCreationPolicy.STATELESS)
            )
            .addFilterBefore(
                new JwtAuthenticationFilter(tokenProvider),
```

```
                UsernamePasswordAuthenticationFilter.class
            );

        return http.build();
    }
}

@Component
public class JwtTokenProvider {

    private final UserDetailsService userDetailsService;

    private final String secretKey;
    private final long tokenValidityInMilliseconds;

    public JwtTokenProvider(
            UserDetailsService userDetailsService,
            @Value("${jwt.secret-key}") String secretKey,
            @Value("${jwt.token-validity-in-seconds}") long
tokenValidityInSeconds) {
        this.userDetailsService = userDetailsService;
        this.secretKey = Base64.getEncoder().encodeToString(secretKey.getBytes());
        this.tokenValidityInMilliseconds = tokenValidityInSeconds * 1000;
    }

    public String createToken(Authentication authentication) {
        // Set token claims
        String username = authentication.getName();
        Collection<? extends GrantedAuthority> authorities =
authentication.getAuthorities();
        Claims claims = Jwts.claims().setSubject(username);
        claims.put("auth", authorities.stream()
                .map(GrantedAuthority::getAuthority)
                .collect(Collectors.joining(",")));

        // Set token expiration
        Date now = new Date();
        Date validity = new Date(now.getTime() + tokenValidityInMilliseconds);

        // Create token
        return Jwts.builder()
                .setClaims(claims)
                .setIssuedAt(now)
                .setExpiration(validity)
                .signWith(SignatureAlgorithm.HS512, secretKey)
                .compact();
    }

    public Authentication getAuthentication(String token) {
        // Parse token claims
        Claims claims = Jwts.parser()
                .setSigningKey(secretKey)
                .parseClaimsJws(token)
                .getBody();
```

```
        // Extract username and authorities
        String username = claims.getSubject();
        List<SimpleGrantedAuthority> authorities = Arrays.stream(
                claims.get("auth").toString().split(","))
                .map(SimpleGrantedAuthority::new)
                .collect(Collectors.toList());

        // Load user details
        UserDetails userDetails = userDetailsService.loadUserByUsername(username);

        return new UsernamePasswordAuthenticationToken(userDetails, "",
authorities);
    }

    public boolean validateToken(String token) {
        try {
            Jwts.parser().setSigningKey(secretKey).parseClaimsJws(token);
            return true;
        } catch (JwtException | IllegalArgumentException e) {
            // Invalid or expired token
            return false;
        }
    }

    public String resolveToken(HttpServletRequest request) {
        String bearerToken = request.getHeader("Authorization");
        if (bearerToken != null && bearerToken.startsWith("Bearer ")) {
            return bearerToken.substring(7);
        }
        return null;
    }
}

@Component
public class JwtAuthenticationFilter extends OncePerRequestFilter {

    private final JwtTokenProvider tokenProvider;

    // Constructor injection

    @Override
    protected void doFilterInternal(
            HttpServletRequest request,
            HttpServletResponse response,
            FilterChain filterChain) throws ServletException, IOException {

        try {
            // Extract token from request
            String token = tokenProvider.resolveToken(request);

            if (token != null && tokenProvider.validateToken(token)) {
                // Get authentication from token
                Authentication authentication =
```

```
tokenProvider.getAuthentication(token);

                // Set authentication in context

SecurityContextHolder.getContext().setAuthentication(authentication);
            }
        } catch (Exception e) {
            // Log token validation failure
            log.error("Could not set user authentication in security context", e);
        }

        filterChain.doFilter(request, response);
    }
}

@Component
public class JwtAuthenticationEntryPoint implements AuthenticationEntryPoint {

    @Override
    public void commence(
            HttpServletRequest request,
            HttpServletResponse response,
            AuthenticationException authException) throws IOException {

        // Set response status and content type
        response.setStatus(HttpStatus.UNAUTHORIZED.value());
        response.setContentType(MediaType.APPLICATION_JSON_VALUE);

        // Create error response
        Map<String, Object> errorDetails = new HashMap<>();
        errorDetails.put("timestamp", new Date());
        errorDetails.put("status", HttpStatus.UNAUTHORIZED.value());
        errorDetails.put("error", "Unauthorized");
        errorDetails.put("message", "Invalid or missing JWT token");
        errorDetails.put("path", request.getRequestURI());

        // Write error response
        ObjectMapper mapper = new ObjectMapper();
        mapper.writeValue(response.getWriter(), errorDetails);
    }
}

@RestController
@RequestMapping("/api/auth")
public class AuthController {

    private final AuthenticationManager authenticationManager;
    private final JwtTokenProvider tokenProvider;
    private final UserService userService;

    // Constructor injection

    @PostMapping("/login")
    public ResponseEntity<?> login(@Valid @RequestBody LoginRequest loginRequest)
```

```
{
        try {
            // Authenticate user
            Authentication authentication = authenticationManager.authenticate(
                    new UsernamePasswordAuthenticationToken(
                            loginRequest.getUsername(),
                            loginRequest.getPassword()
                    )
            );

            // Set authentication in context
            SecurityContextHolder.getContext().setAuthentication(authentication);

            // Generate token
            String token = tokenProvider.createToken(authentication);

            // Create response
            Map<String, Object> response = new HashMap<>();
            response.put("token", token);
            response.put("username", authentication.getName());
            response.put("roles", authentication.getAuthorities().stream()
                    .map(GrantedAuthority::getAuthority)
                    .collect(Collectors.toList()));

            return ResponseEntity.ok(response);
        } catch (AuthenticationException e) {
            return ResponseEntity.status(HttpStatus.UNAUTHORIZED)
                    .body(Map.of("error", "Invalid username or password"));
        }
    }

    @PostMapping("/register")
    public ResponseEntity<?> register(@Valid @RequestBody RegisterRequest
registerRequest) {
        // Check if username already exists
        if (userService.existsByUsername(registerRequest.getUsername())) {
            return ResponseEntity.badRequest()
                    .body(Map.of("error", "Username already exists"));
        }

        // Check if email already exists
        if (userService.existsByEmail(registerRequest.getEmail())) {
            return ResponseEntity.badRequest()
                    .body(Map.of("error", "Email already exists"));
        }

        // Create new user
        User user = userService.createUser(
                registerRequest.getUsername(),
                registerRequest.getEmail(),
                registerRequest.getPassword(),
                Set.of("ROLE_USER")
        );
```

```
        return ResponseEntity.status(HttpStatus.CREATED)
                .body(Map.of(
                        "id", user.getId(),
                        "username", user.getUsername(),
                        "email", user.getEmail(),
                        "message", "User registered successfully"
                ));
    }
}
```

OAuth2 Authentication

Integration with OAuth2 providers:

```
@Configuration
@EnableWebSecurity
public class OAuth2SecurityConfig {

    @Bean
    public SecurityFilterChain oauth2SecurityFilterChain(HttpSecurity http) throws
Exception {
        http
            .authorizeHttpRequests(authorize -> authorize
                .requestMatchers("/", "/login/**", "/oauth2/**").permitAll()
                .anyRequest().authenticated()
            )
            .oauth2Login(oauth2 -> oauth2
                .loginPage("/login")
                .defaultSuccessUrl("/dashboard", true)
                .failureUrl("/login?error=true")
                .userInfoEndpoint(userInfo -> userInfo
                    .userService(oAuth2UserService())
                    .oidcUserService(oidcUserService())
                )
                .authorizationEndpoint(authorization -> authorization
                    .baseUri("/oauth2/authorize")

.authorizationRequestRepository(cookieAuthorizationRequestRepository())
                )
                .redirectionEndpoint(redirection -> redirection
                    .baseUri("/oauth2/callback/*")
                )
                .successHandler(oAuth2AuthenticationSuccessHandler())
                .failureHandler(oAuth2AuthenticationFailureHandler())
            );

        return http.build();
    }

    @Bean
    public OAuth2UserService<OAuth2UserRequest, OAuth2User> oAuth2UserService() {
```

```
        return new CustomOAuth2UserService();
    }

    @Bean
    public OAuth2UserService<OidcUserRequest, OidcUser> oidcUserService() {
        return new CustomOidcUserService();
    }

    @Bean
    public AuthorizationRequestRepository<OAuth2AuthorizationRequest>
cookieAuthorizationRequestRepository() {
        return new HttpCookieOAuth2AuthorizationRequestRepository();
    }

    @Bean
    public AuthenticationSuccessHandler oAuth2AuthenticationSuccessHandler() {
        return new OAuth2AuthenticationSuccessHandler();
    }

    @Bean
    public AuthenticationFailureHandler oAuth2AuthenticationFailureHandler() {
        return new OAuth2AuthenticationFailureHandler();
    }
}
```

@Service public class CustomOAuth2UserService extends DefaultOAuth2UserService {

```
private final UserRepository userRepository;

// Constructor injection

@Override
public OAuth2User loadUser(OAuth2UserRequest userRequest) throws
OAuth2AuthenticationException {
    OAuth2User oauth2User = super.loadUser(userRequest);

    try {
        return processOAuth2User(userRequest, oauth2User);
    } catch (Exception ex) {
        throw new OAuth2AuthenticationException(ex.getMessage());
    }
}

private OAuth2User processOAuth2User(OAuth2UserRequest userRequest, OAuth2User
oauth2User) {
    // Extract provider details
    String provider = userRequest.getClientRegistration().getRegistrationId();

    // Extract user information (provider-specific)
```

```
    OAuth2UserInfo userInfo = OAuth2UserInfoFactory.getOAuth2UserInfo(provider,
oauth2User.getAttributes());

    // Check if user exists
    Optional<User> userOptional = userRepository.findByEmail(userInfo.getEmail());
    User user;

    if (userOptional.isPresent()) {
        // Update existing user
        user = userOptional.get();

        // Check if user is linked to this OAuth2 provider
        if
(!user.getProvider().equals(AuthProvider.valueOf(provider.toUpperCase()))) {
            throw new OAuth2AuthenticationException(
                    "You're signed up with " + user.getProvider() + " account. " +
                    "Please use your " + user.getProvider() + " account to log
in.");
        }

        // Update user details
        user = updateExistingUser(user, userInfo);
    } else {
        // Create new user
        user = registerNewUser(userRequest, userInfo);
    }

    // Return user with authorities
    return UserPrincipal.create(user, oauth2User.getAttributes());
}

private User registerNewUser(OAuth2UserRequest userRequest, OAuth2UserInfo
userInfo) {
    // Create new user entity
    User user = new User();


user.setProvider(AuthProvider.valueOf(userRequest.getClientRegistration().getRegis
trationId().toUpperCase()));
    user.setProviderId(userInfo.getId());
    user.setName(userInfo.getName());
    user.setEmail(userInfo.getEmail());
    user.setImageUrl(userInfo.getImageUrl());
    user.setEmailVerified(true);

    // Set default role for OAuth users
    Role userRole = roleRepository.findByName("ROLE_USER")
            .orElseThrow(() -> new ResourceNotFoundException("Default role not
found"));
    user.setRoles(Set.of(userRole));
```

```
        return userRepository.save(user);
    }

    private User updateExistingUser(User user, OAuth2UserInfo userInfo) {
        // Update user information
        user.setName(userInfo.getName());
        user.setImageUrl(userInfo.getImageUrl());

        return userRepository.save(user);
    }
```

}

// Factory for different OAuth2 providers public class OAuth2UserInfoFactory {

```
    public static OAuth2UserInfo getOAuth2UserInfo(String registrationId, Map<String,
    Object> attributes) {
        if (registrationId.equalsIgnoreCase(AuthProvider.GOOGLE.toString())) {
            return new GoogleOAuth2UserInfo(attributes);
        } else if (registrationId.equalsIgnoreCase(AuthProvider.FACEBOOK.toString()))
    {
            return new FacebookOAuth2UserInfo(attributes);
        } else if (registrationId.equalsIgnoreCase(AuthProvider.GITHUB.toString())) {
            return new GithubOAuth2UserInfo(attributes);
        } else {
            throw new OAuth2AuthenticationException("Sorry! Login with " +
    registrationId + " is not supported yet.");
        }
    }
```

}

// Abstract class for OAuth2 user info public abstract class OAuth2UserInfo {

```
    protected Map<String, Object> attributes;

    public OAuth2UserInfo(Map<String, Object> attributes) {
        this.attributes = attributes;
    }

    public Map<String, Object> getAttributes() {
        return attributes;
    }

    public abstract String getId();

    public abstract String getName();
```

```
    public abstract String getEmail();

    public abstract String getImageUrl();

}

// Google implementation public class GoogleOAuth2UserInfo extends OAuth2UserInfo {

    public GoogleOAuth2UserInfo(Map<String, Object> attributes) {
        super(attributes);
    }

    @Override
    public String getId() {
        return (String) attributes.get("sub");
    }

    @Override
    public String getName() {
        return (String) attributes.get("name");
    }

    @Override
    public String getEmail() {
        return (String) attributes.get("email");
    }

    @Override
    public String getImageUrl() {
        return (String) attributes.get("picture");
    }

}

// Configuration for OAuth2 providers @Configuration public class OAuth2ClientConfig {

    @Bean
    public ClientRegistrationRepository clientRegistrationRepository() {
        return new InMemoryClientRegistrationRepository(
                googleClientRegistration(),
                facebookClientRegistration(),
                githubClientRegistration()
        );
    }

    private ClientRegistration googleClientRegistration() {
        return ClientRegistration.withRegistrationId("google")
                .clientId("google-client-id")
```

```
            .clientSecret("google-client-secret")

.clientAuthenticationMethod(ClientAuthenticationMethod.CLIENT_SECRET_BASIC)
            .authorizationGrantType(AuthorizationGrantType.AUTHORIZATION_CODE)
            .redirectUri("{baseUrl}/oauth2/callback/{registrationId}")
            .scope("email", "profile")
            .authorizationUri("https://accounts.google.com/o/oauth2/v2/auth")
            .tokenUri("https://www.googleapis.com/oauth2/v4/token")
            .userInfoUri("https://www.googleapis.com/oauth2/v3/userinfo")
            .userNameAttributeName(IdTokenClaimNames.SUB)
            .jwkSetUri("https://www.googleapis.com/oauth2/v3/certs")
            .clientName("Google")
            .build();
}

private ClientRegistration facebookClientRegistration() {
    return ClientRegistration.withRegistrationId("facebook")
            .clientId("facebook-client-id")
            .clientSecret("facebook-client-secret")

.clientAuthenticationMethod(ClientAuthenticationMethod.CLIENT_SECRET_POST)
            .authorizationGrantType(AuthorizationGrantType.AUTHORIZATION_CODE)
            .redirectUri("{baseUrl}/oauth2/callback/{registrationId}")
            .scope("email", "public_profile")
            .authorizationUri("https://www.facebook.com/v12.0/dialog/oauth")
            .tokenUri("https://graph.facebook.com/v12.0/oauth/access_token")
            .userInfoUri("https://graph.facebook.com/v12.0/me?
fields=id,name,email,picture")
            .userNameAttributeName("id")
            .clientName("Facebook")
            .build();
}

private ClientRegistration githubClientRegistration() {
    return ClientRegistration.withRegistrationId("github")
            .clientId("github-client-id")
            .clientSecret("github-client-secret")

.clientAuthenticationMethod(ClientAuthenticationMethod.CLIENT_SECRET_BASIC)
            .authorizationGrantType(AuthorizationGrantType.AUTHORIZATION_CODE)
            .redirectUri("{baseUrl}/oauth2/callback/{registrationId}")
            .scope("user:email", "read:user")
            .authorizationUri("https://github.com/login/oauth/authorize")
            .tokenUri("https://github.com/login/oauth/access_token")
            .userInfoUri("https://api.github.com/user")
            .userNameAttributeName("id")
            .clientName("GitHub")
            .build();
}
```

}

// Properties for OAuth2 configuration @ConfigurationProperties(prefix = "app.oauth2") public class OAuth2Properties {

```
private final Auth auth = new Auth();
private final OAuth2 oauth2 = new OAuth2();

public static class Auth {
    private String tokenSecret;
    private long tokenExpirationMs;

    // Getters and setters
}

public static class OAuth2 {
    private List<String> authorizedRedirectUris = new ArrayList<>();

    // Getters and setters
}

// Getters
public Auth getAuth() {
    return auth;
}

public OAuth2 getOauth2() {
    return oauth2;
}
```

}

```
#### LDAP Authentication

Integration with LDAP directory services:

```java
@Configuration
@EnableWebSecurity
public class LdapSecurityConfig {

    @Bean
    public SecurityFilterChain ldapSecurityFilterChain(HttpSecurity http) throws
Exception {
        http
            .authorizeHttpRequests(authorize -> authorize
                .requestMatchers("/login").permitAll()
                .anyRequest().authenticated()
```
```

```
            )
            .formLogin(Customizer.withDefaults());

        return http.build();
    }

    @Bean
    public AuthenticationManager
ldapAuthenticationManager(BaseLdapPathContextSource contextSource) {
        LdapBindAuthenticationManagerFactory factory = new
LdapBindAuthenticationManagerFactory(contextSource);

        // Configure LDAP authentication settings
        factory.setUserDnPatterns("uid={0},ou=people");
        factory.setUserSearchBase("ou=people");
        factory.setUserSearchFilter("(uid={0})");

        return factory.createAuthenticationManager();
    }

    @Bean
    public BaseLdapPathContextSource contextSource() {
        LdapConnectionDetails connectionDetails = ldapConnectionDetails();

        DefaultSpringSecurityContextSource contextSource = new
DefaultSpringSecurityContextSource(

connectionDetails.getUrls().stream().findFirst().orElse("ldap://localhost:389"));

        contextSource.setUserDn(connectionDetails.getManagerDn());
        contextSource.setPassword(connectionDetails.getManagerPassword());
        contextSource.setBase("dc=example,dc=com");

        return contextSource;
    }

    @Bean
    public LdapConnectionDetails ldapConnectionDetails() {
        return new LdapConnectionDetails() {
            @Override
            public List<String> getUrls() {
                return List.of("ldap://ldap.example.com:389");
            }

            @Override
            public String getManagerDn() {
                return "cn=admin,dc=example,dc=com";
            }

            @Override
            public String getManagerPassword() {
                return "admin_password";
            }
        };
```

```
    }
}
```

Multiple Authentication Providers

Combining multiple authentication methods:

```
@Configuration
@EnableWebSecurity
public class MultipleAuthProvidersConfig {

    private final UserDetailsService userDetailsService;
    private final PasswordEncoder passwordEncoder;
    private final JwtTokenProvider jwtTokenProvider;
    private final CustomOAuth2UserService oAuth2UserService;

    // Constructor injection

    @Bean
    public SecurityFilterChain securityFilterChain(HttpSecurity http) throws
Exception {
        http
            .authorizeHttpRequests(authorize -> authorize
                .requestMatchers("/api/auth/**", "/oauth2/**").permitAll()
                .requestMatchers("/api/admin/**").hasRole("ADMIN")
                .anyRequest().authenticated()
            )
            .formLogin(form -> form
                .loginPage("/login")
                .permitAll()
            )
            .oauth2Login(oauth2 -> oauth2
                .userInfoEndpoint(userInfo -> userInfo
                    .userService(oAuth2UserService)
                )
            )
            .addFilterBefore(
                new JwtAuthenticationFilter(jwtTokenProvider),
                UsernamePasswordAuthenticationFilter.class
            );

        return http.build();
    }

    @Bean
    public AuthenticationManager authenticationManager(AuthenticationConfiguration
authConfig) throws Exception {
        return authConfig.getAuthenticationManager();
    }

    @Bean
```

```
    public DaoAuthenticationProvider daoAuthenticationProvider() {
        DaoAuthenticationProvider provider = new DaoAuthenticationProvider();
        provider.setUserDetailsService(userDetailsService);
        provider.setPasswordEncoder(passwordEncoder);
        return provider;
    }

    @Bean
    public JwtAuthenticationProvider jwtAuthenticationProvider() {
        return new JwtAuthenticationProvider(jwtTokenProvider);
    }

    @Bean
    public LdapAuthenticationProvider ldapAuthenticationProvider() {
        // Configure LDAP authentication provider
        BindAuthenticator authenticator = new BindAuthenticator(contextSource());
        authenticator.setUserDnPatterns(new String[] { "uid=
{0},ou=people,dc=example,dc=com" });

        LdapAuthenticationProvider provider = new
LdapAuthenticationProvider(authenticator, authoritiesPopulator());
        return provider;
    }

    @Bean
    public DefaultLdapAuthoritiesPopulator authoritiesPopulator() {
        DefaultLdapAuthoritiesPopulator populator = new
DefaultLdapAuthoritiesPopulator(
                contextSource(), "ou=groups,dc=example,dc=com");
        populator.setGroupSearchFilter("(uniqueMember={0})");
        populator.setRolePrefix("ROLE_");
        populator.setSearchSubtree(true);
        return populator;
    }

    @Bean
    public BaseLdapPathContextSource contextSource() {
        // Configure LDAP context source
        DefaultSpringSecurityContextSource contextSource = new
DefaultSpringSecurityContextSource(
                "ldap://ldap.example.com:389/dc=example,dc=com");
        contextSource.setUserDn("cn=admin,dc=example,dc=com");
        contextSource.setPassword("admin_password");
        return contextSource;
    }

    @Autowired
    public void configureGlobal(AuthenticationManagerBuilder auth) throws
Exception {
        // Register authentication providers
        auth.authenticationProvider(daoAuthenticationProvider())
            .authenticationProvider(jwtAuthenticationProvider())
            .authenticationProvider(ldapAuthenticationProvider());
```

```
    }
}
```

Two-Factor Authentication (2FA)

Implementing additional security with two-factor authentication:

```
@Configuration
@EnableWebSecurity
public class TwoFactorSecurityConfig {

    private final UserDetailsService userDetailsService;
    private final PasswordEncoder passwordEncoder;
    private final TwoFactorAuthenticationService twoFactorService;

    // Constructor injection

    @Bean
    public SecurityFilterChain securityFilterChain(HttpSecurity http) throws
Exception {
        http
            .authorizeHttpRequests(authorize -> authorize
                .requestMatchers("/login", "/login/2fa").permitAll()
                .anyRequest().authenticated()
            )
            .formLogin(form -> form
                .loginPage("/login")
                .successHandler(twoFactorAuthenticationSuccessHandler())
            )
            .addFilterBefore(
                new CustomTwoFactorAuthenticationFilter(authenticationManager()),
                UsernamePasswordAuthenticationFilter.class
            );

        return http.build();
    }

    @Bean
    public AuthenticationSuccessHandler twoFactorAuthenticationSuccessHandler() {
        return (request, response, authentication) -> {
            // Check if user requires 2FA
            UserDetails userDetails = (UserDetails) authentication.getPrincipal();
            User user = userRepository.findByUsername(userDetails.getUsername())
                    .orElseThrow(() -> new UsernameNotFoundException("User not
found"));

            if (user.isTwoFactorEnabled()) {
                // Generate and send 2FA code
                String code = twoFactorService.generateAndSendCode(user);

                // Save partial authentication in session
```

```
                HttpSession session = request.getSession();
                session.setAttribute("SPRING_SECURITY_CONTEXT",
SecurityContextHolder.getContext());
                session.setAttribute("REQUIRES_2FA", true);

                // Redirect to 2FA verification page
                response.sendRedirect("/login/2fa");
            } else {
                // No 2FA required, redirect to home page
                response.sendRedirect("/");
            }
        };
    }

    @Bean
    public AuthenticationManager authenticationManager(AuthenticationConfiguration
authConfig) throws Exception {
        return authConfig.getAuthenticationManager();
    }

    @Bean
    public DaoAuthenticationProvider daoAuthenticationProvider() {
        DaoAuthenticationProvider provider = new DaoAuthenticationProvider();
        provider.setUserDetailsService(userDetailsService);
        provider.setPasswordEncoder(passwordEncoder);
        return provider;
    }

    @Autowired
    public void configureGlobal(AuthenticationManagerBuilder auth) throws
Exception {
        auth.authenticationProvider(daoAuthenticationProvider());
    }
}

@Service
public class TwoFactorAuthenticationService {

    private final UserRepository userRepository;
    private final EmailService emailService;
    private final SecureRandom secureRandom;

    // Constructor injection

    public String generateAndSendCode(User user) {
        // Generate a 6-digit code
        String code = generateRandomCode();

        // Store code in user entity
        user.setTwoFactorCode(code);
        user.setTwoFactorExpiry(LocalDateTime.now().plusMinutes(5)); // Valid for
5 minutes
        userRepository.save(user);
```

```
        // Send code via email
        emailService.sendTwoFactorCode(user.getEmail(), code);

        return code;
    }

    public boolean verifyCode(String username, String code) {
        User user = userRepository.findByUsername(username)
                .orElseThrow(() -> new UsernameNotFoundException("User not
found"));

        // Check if code matches and has not expired
        if (user.getTwoFactorCode().equals(code) &&
                LocalDateTime.now().isBefore(user.getTwoFactorExpiry())) {

            // Clear code after successful verification
            user.setTwoFactorCode(null);
            user.setTwoFactorExpiry(null);
            userRepository.save(user);

            return true;
        }

        return false;
    }

    private String generateRandomCode() {
        // Generate random 6-digit code
        int code = secureRandom.nextInt(900000) + 100000;
        return String.valueOf(code);
    }
}

@Component
public class CustomTwoFactorAuthenticationFilter extends OncePerRequestFilter {

    private final AuthenticationManager authenticationManager;

    // Constructor injection

    @Override
    protected void doFilterInternal(
            HttpServletRequest request,
            HttpServletResponse response,
            FilterChain filterChain) throws ServletException, IOException {

        HttpSession session = request.getSession(false);

        if (session != null &&
Boolean.TRUE.equals(session.getAttribute("REQUIRES_2FA")) &&
                "/login/2fa".equals(request.getRequestURI()) &&
"POST".equals(request.getMethod())) {

            // Get security context from session
```

```
            SecurityContext securityContext = (SecurityContext)
session.getAttribute("SPRING_SECURITY_CONTEXT");

            if (securityContext != null) {
                // Get existing authentication
                Authentication existingAuth = securityContext.getAuthentication();
                String username = existingAuth.getName();

                // Get 2FA code from request
                String code = request.getParameter("code");

                try {
                    // Verify 2FA code
                    if (twoFactorService.verifyCode(username, code)) {
                        // Code is valid, set authentication in security context

SecurityContextHolder.getContext().setAuthentication(existingAuth);

                        // Clear 2FA flag
                        session.removeAttribute("REQUIRES_2FA");

                        // Redirect to home page
                        response.sendRedirect("/");
                        return;
                    } else {
                        // Code is invalid, redirect back to 2FA page with error
                        response.sendRedirect("/login/2fa?error=true");
                        return;
                    }
                } catch (Exception e) {
                    // Handle verification error
                    response.sendRedirect("/login/2fa?error=true");
                    return;
                }
            }
        }

        // Continue with filter chain for non-2FA requests
        filterChain.doFilter(request, response);
    }

    @Override
    protected boolean shouldNotFilter(HttpServletRequest request) {
        // Only apply this filter to 2FA verification endpoint
        return !"/login/2fa".equals(request.getRequestURI()) ||
!"POST".equals(request.getMethod());
    }
}

@Controller
public class TwoFactorAuthenticationController {

    @GetMapping("/login/2fa")
    public String twoFactorPage(HttpSession session, Model model) {
```

```
        // Check if user requires 2FA
        if (session == null ||
!Boolean.TRUE.equals(session.getAttribute("REQUIRES_2FA"))) {
            return "redirect:/login";
        }

        return "2fa";
    }

    @PostMapping("/login/2fa")
    public void processTwoFactor() {
        // This method does nothing
        // The CustomTwoFactorAuthenticationFilter handles the actual verification
    }
}
```

Authorization and Access Control

Spring Security provides comprehensive authorization mechanisms to control access to resources.

Role-Based Access Control (RBAC)

Implementing role-based security:

```
@Configuration
@EnableWebSecurity
public class RbacSecurityConfig {

    @Bean
    public SecurityFilterChain securityFilterChain(HttpSecurity http) throws
Exception {
        http
            .authorizeHttpRequests(authorize -> authorize
                // Public access
                .requestMatchers("/", "/home", "/register", "/css/**",
"/js/**").permitAll()

                // User role access
                .requestMatchers("/dashboard", "/profile/**").hasRole("USER")

                // Manager role access
                .requestMatchers("/reports/**").hasRole("MANAGER")

                // Admin role access
                .requestMatchers("/admin/**", "/actuator/**").hasRole("ADMIN")

                // Super admin role access
                .requestMatchers("/system/**").hasRole("SUPER_ADMIN")

                // Any authenticated user
                .anyRequest().authenticated()
```

```
            );

        return http.build();
    }
}

@Entity
@Table(name = "roles")
public class Role {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    @Column(nullable = false, unique = true)
    private String name;

    @ManyToMany(mappedBy = "roles")
    private Set<User> users = new HashSet<>();

    // Getters and setters
}

@Entity
@Table(name = "users")
public class User {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    @Column(nullable = false, unique = true)
    private String username;

    @Column(nullable = false)
    private String password;

    @ManyToMany(fetch = FetchType.EAGER)
    @JoinTable(
        name = "user_roles",
        joinColumns = @JoinColumn(name = "user_id"),
        inverseJoinColumns = @JoinColumn(name = "role_id")
    )
    private Set<Role> roles = new HashSet<>();

    // Getters and setters
}

@Service
public class UserDetailsServiceImpl implements UserDetailsService {

    private final UserRepository userRepository;

    // Constructor injection
```

```
    @Override
    public UserDetails loadUserByUsername(String username) throws
UsernameNotFoundException {
        User user = userRepository.findByUsername(username)
                .orElseThrow(() -> new UsernameNotFoundException("User not found:
" + username));

        // Convert roles to Spring Security authorities
        Set<GrantedAuthority> authorities = user.getRoles().stream()
                .map(role -> new SimpleGrantedAuthority("ROLE_" +
role.getName().toUpperCase()))
                .collect(Collectors.toSet());

        return new org.springframework.security.core.userdetails.User(
                user.getUsername(),
                user.getPassword(),
                authorities
        );
    }
}
```

Permission-Based Access Control

Implementing more granular permission-based security:

```
@Entity
@Table(name = "permissions")
public class Permission {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    @Column(nullable = false, unique = true)
    private String name;

    // Getters and setters
}

@Entity
@Table(name = "roles")
public class Role {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    @Column(nullable = false, unique = true)
    private String name;
```

```
    @ManyToMany(fetch = FetchType.EAGER)
    @JoinTable(
        name = "role_permissions",
        joinColumns = @JoinColumn(name = "role_id"),
        inverseJoinColumns = @JoinColumn(name = "permission_id")
    )
    private Set<Permission> permissions = new HashSet<>();

    @ManyToMany(mappedBy = "roles")
    private Set<User> users = new HashSet<>();

    // Getters and setters
}

@Configuration
@EnableWebSecurity
public class PermissionBasedSecurityConfig {

    @Bean
    public SecurityFilterChain securityFilterChain(HttpSecurity http) throws
Exception {
        http
            .authorizeHttpRequests(authorize -> authorize
                // Public access
                .requestMatchers("/", "/home").permitAll()

                // Permission-based access
                .requestMatchers("/users/**").hasAuthority("USER_READ")
                .requestMatchers("/users/create").hasAuthority("USER_CREATE")
                .requestMatchers("/users/update").hasAuthority("USER_UPDATE")
                .requestMatchers("/users/delete").hasAuthority("USER_DELETE")

                .requestMatchers("/products/**").hasAuthority("PRODUCT_READ")

.requestMatchers("/products/create").hasAuthority("PRODUCT_CREATE")

.requestMatchers("/products/update").hasAuthority("PRODUCT_UPDATE")

.requestMatchers("/products/delete").hasAuthority("PRODUCT_DELETE")

                // Any authenticated user
                .anyRequest().authenticated()
            );

        return http.build();
    }
}

@Service
public class CustomUserDetailsService implements UserDetailsService {

    private final UserRepository userRepository;

    // Constructor injection
```

```
    @Override
    public UserDetails loadUserByUsername(String username) throws
UsernameNotFoundException {
        User user = userRepository.findByUsername(username)
                .orElseThrow(() -> new UsernameNotFoundException("User not found:
" + username));

        // Get user roles
        Set<GrantedAuthority> authorities = new HashSet<>();

        // Add role-based authorities
        for (Role role : user.getRoles()) {
            authorities.add(new SimpleGrantedAuthority("ROLE_" +
role.getName().toUpperCase()));

            // Add permission-based authorities
            for (Permission permission : role.getPermissions()) {
                authorities.add(new SimpleGrantedAuthority(permission.getName()));
            }
        }

        return new org.springframework.security.core.userdetails.User(
                user.getUsername(),
                user.getPassword(),
                authorities
        );
    }
}
```

Expression-Based Access Control

Using Spring Expression Language (SpEL) for complex authorization rules:

```
@Configuration
@EnableWebSecurity
public class ExpressionBasedSecurityConfig {

    @Bean
    public SecurityFilterChain securityFilterChain(HttpSecurity http) throws
Exception {
        http
            .authorizeHttpRequests(authorize -> authorize
                // Using hasRole expressions
                .requestMatchers("/admin/**").hasRole("ADMIN")

                // Using hasAuthority expressions
                .requestMatchers("/api/users/**").hasAuthority("USER_ADMIN")

                // Using hasAnyRole expressions
                .requestMatchers("/reports/**").hasAnyRole("ADMIN", "MANAGER")
```

```
                // Using hasAnyAuthority expressions
                .requestMatchers("/documents/**").hasAnyAuthority("DOC_READ",
"DOC_WRITE")

                // Using access expressions
                .requestMatchers("/premium/**").access(new
WebExpressionAuthorizationManager(
                        "hasRole('PREMIUM') and hasAuthority('PAYMENT_VERIFIED')")
                )

                // Custom access expression for business hours (9 AM to 5 PM)
                .requestMatchers("/business/**").access(new
WebExpressionAuthorizationManager(
                        "hasRole('EMPLOYEE') and
@businessHoursEvaluator.isBusinessHours()")
                )

                // IP address restriction
                .requestMatchers("/internal/**").access(new
WebExpressionAuthorizationManager(
                        "hasRole('ADMIN') and hasIpAddress('192.168.1.0/24')")
                )

                // Any authenticated user
                .anyRequest().authenticated()
            );

        return http.build();
    }

    @Bean
    public BusinessHoursEvaluator businessHoursEvaluator() {
        return new BusinessHoursEvaluator();
    }
}

@Component("businessHoursEvaluator")
public class BusinessHoursEvaluator {

    private static final LocalTime START_TIME = LocalTime.of(9, 0);
    private static final LocalTime END_TIME = LocalTime.of(17, 0);

    public boolean isBusinessHours() {
        LocalTime currentTime = LocalTime.now();
        LocalDate currentDate = LocalDate.now();

        // Check if weekend
        if (currentDate.getDayOfWeek() == DayOfWeek.SATURDAY ||
            currentDate.getDayOfWeek() == DayOfWeek.SUNDAY) {
            return false;
        }

        // Check if within business hours
```

```
        return !currentTime.isBefore(START_TIME) &&
!currentTime.isAfter(END_TIME);
    }
}
```

Custom Authorization Rules

Implementing custom authorization logic:

```
@Component
public class CustomAuthorizationManager<T> implements AuthorizationManager<T> {

    private final UserRepository userRepository;
    private final AuditService auditService;

    // Constructor injection

    @Override
    public AuthorizationDecision check(Supplier<Authentication> authentication, T
object) {
        // Cast to HTTP request if applicable
        HttpServletRequest request = null;
        if (object instanceof Supplier) {
            Supplier<?> supplier = (Supplier<?>) object;
            Object supplierObj = supplier.get();
            if (supplierObj instanceof HttpServletRequest) {
                request = (HttpServletRequest) supplierObj;
            }
        }

        // Get user details
        Authentication auth = authentication.get();
        if (auth == null || !auth.isAuthenticated()) {
            return new AuthorizationDecision(false);
        }

        String username = auth.getName();

        // Example: Allow access during business hours only for employees
        if (request != null && request.getRequestURI().startsWith("/business")) {
            boolean isEmployee = auth.getAuthorities().stream()
                    .anyMatch(a -> a.getAuthority().equals("ROLE_EMPLOYEE"));

            boolean isBusinessHours = businessHoursEvaluator.isBusinessHours();

            // Audit access attempt
            auditService.logAccessAttempt(username, request.getRequestURI(),
                    isEmployee && isBusinessHours);

            return new AuthorizationDecision(isEmployee && isBusinessHours);
        }
```

```
        // Example: Rate limiting for API endpoints
        if (request != null && request.getRequestURI().startsWith("/api")) {
            boolean underRateLimit = rateLimitService.checkRateLimit(username);

            if (!underRateLimit) {
                // Log rate limit exceeded
                auditService.logRateLimitExceeded(username,
request.getRequestURI());
            }

            return new AuthorizationDecision(underRateLimit);
        }

        // Default to allowing access
        return new AuthorizationDecision(true);
    }
}

@Configuration
@EnableWebSecurity
public class CustomAuthorizationConfig {

    private final CustomAuthorizationManager<RequestAuthorizationContext>
customAuthorizationManager;

    // Constructor injection

    @Bean
    public SecurityFilterChain securityFilterChain(HttpSecurity http) throws
Exception {
        http
            .authorizeHttpRequests(authorize -> authorize
                // Apply custom authorization to specific paths
                .requestMatchers("/business/**", "/api/**")
                .access(customAuthorizationManager)

                // Standard authorization for other paths
                .anyRequest().authenticated()
            );

        return http.build();
    }
}
```

Hierarchical Roles

Setting up role hierarchy for simplified access control:

```
@Configuration
@EnableWebSecurity
```

```
public class RoleHierarchyConfig {

    @Bean
    public RoleHierarchy roleHierarchy() {
        RoleHierarchyImpl roleHierarchy = new RoleHierarchyImpl();

        // Define role hierarchy (higher roles include permissions of lower roles)
        String hierarchy =
                "ROLE_SUPER_ADMIN > ROLE_ADMIN\n" +
                "ROLE_ADMIN > ROLE_MANAGER\n" +
                "ROLE_MANAGER > ROLE_USER";

        roleHierarchy.setHierarchy(hierarchy);
        return roleHierarchy;
    }

    @Bean
    public AuthorizationManager<RequestAuthorizationContext>
requestMatcherAuthorization() {
        RequestMatcherDelegatingAuthorizationManager.Builder builder =
RequestMatcherDelegatingAuthorizationManager.builder();

        builder.add(new AntPathRequestMatcher("/admin/**"), new
RoleHierarchyAuthorizationManager(roleHierarchy(), "ROLE_ADMIN"));
        builder.add(new AntPathRequestMatcher("/management/**"), new
RoleHierarchyAuthorizationManager(roleHierarchy(), "ROLE_MANAGER"));
        builder.add(new AntPathRequestMatcher("/user/**"), new
RoleHierarchyAuthorizationManager(roleHierarchy(), "ROLE_USER"));

        return builder.build();
    }

    @Bean
    public SecurityFilterChain securityFilterChain(HttpSecurity http) throws
Exception {
        http
            .authorizeHttpRequests(authorize -> authorize
                .requestMatchers("/admin/**", "/management/**", "/user/**")
                .access(requestMatcherAuthorization())
                .anyRequest().authenticated()
            );

        return http.build();
    }

    @Bean
    public MethodSecurityExpressionHandler methodSecurityExpressionHandler() {
        DefaultMethodSecurityExpressionHandler expressionHandler = new
DefaultMethodSecurityExpressionHandler();
        expressionHandler.setRoleHierarchy(roleHierarchy());
        return expressionHandler;
    }
}
```

```
// Custom AuthorizationManager with role hierarchy support
public class RoleHierarchyAuthorizationManager implements
AuthorizationManager<RequestAuthorizationContext> {

    private final RoleHierarchy roleHierarchy;
    private final String role;

    public RoleHierarchyAuthorizationManager(RoleHierarchy roleHierarchy, String
role) {
        this.roleHierarchy = roleHierarchy;
        this.role = role;
    }

    @Override
    public AuthorizationDecision check(Supplier<Authentication> authentication,
RequestAuthorizationContext context) {
        Authentication auth = authentication.get();
        if (auth == null) {
            return new AuthorizationDecision(false);
        }

        // Get all reachable authorities based on role hierarchy
        Collection<GrantedAuthority> authorities =
roleHierarchy.getReachableGrantedAuthorities(auth.getAuthorities());

        boolean hasRole = authorities.stream()
                .map(GrantedAuthority::getAuthority)
                .anyMatch(a -> a.equals(role));

        return new AuthorizationDecision(hasRole);
    }
}
```

Method-level Security

Spring Security allows securing methods with annotations.

@Secured Annotation

Basic role-based method security:

```
@Configuration
@EnableMethodSecurity(securedEnabled = true)
public class MethodSecurityConfig {
    // Configure method security
}

@Service
public class UserService {

    // Requires ADMIN role
```

```
    @Secured("ROLE_ADMIN")
    public List<User> getAllUsers() {
        return userRepository.findAll();
    }

    // Requires ADMIN or MANAGER role
    @Secured({"ROLE_ADMIN", "ROLE_MANAGER"})
    public User getUserById(Long id) {
        return userRepository.findById(id)
                .orElseThrow(() -> new ResourceNotFoundException("User not
found"));
    }

    // No annotation - accessible to any authenticated user
    public User getCurrentUser(Authentication authentication) {
        return userRepository.findByUsername(authentication.getName())
                .orElseThrow(() -> new UsernameNotFoundException("User not
found"));
    }
}
```

@PreAuthorize and @PostAuthorize

More powerful SpEL-based method security:

```
@Configuration
@EnableMethodSecurity(prePostEnabled = true)
public class PrePostMethodSecurityConfig {
    // Configure method security
}

@Service
public class DocumentService {

    // Check before method execution
    @PreAuthorize("hasRole('ADMIN') or hasAuthority('DOC_ADMIN')")
    public List<Document> getAllDocuments() {
        return documentRepository.findAll();
    }

    // More complex expressions with method parameters
    @PreAuthorize("hasAuthority('DOC_READ') or #id <= 10")
    public Document getDocumentById(Long id) {
        return documentRepository.findById(id)
                .orElseThrow(() -> new ResourceNotFoundException("Document not
found"));
    }

    // Check user is the owner of the document
    @PreAuthorize("#document.ownerId == authentication.principal.id or
hasRole('ADMIN')")
```

```
    public Document saveDocument(Document document) {
        return documentRepository.save(document);
    }

    // Check after method execution based on the returned value
    @PostAuthorize("returnObject.ownerId == authentication.principal.id or
hasRole('ADMIN')")
    public Document getDocumentWithSensitiveData(Long id) {
        Document document = documentRepository.findById(id)
                .orElseThrow(() -> new ResourceNotFoundException("Document not
found"));

        // Load sensitive data
        document.setSensitiveData(sensitiveDataRepository.findByDocumentId(id));

        return document;
    }

    // Combining Pre and Post authorization
    @PreAuthorize("hasAuthority('DOC_WRITE')")
    @PostAuthorize("returnObject.status == 'PUBLISHED' or returnObject.ownerId ==
authentication.principal.id")
    public Document publishDocument(Long id) {
        Document document = documentRepository.findById(id)
                .orElseThrow(() -> new ResourceNotFoundException("Document not
found"));

        document.setStatus("PUBLISHED");
        document.setPublishedDate(new Date());

        return documentRepository.save(document);
    }
}
```

@PreFilter and @PostFilter

Filter collections before or after method execution:

```
@Service
public class TaskService {

    // Filter input collection - only process tasks owned by the current user
    @PreFilter("filterObject.assigneeId == authentication.principal.id or
hasRole('MANAGER')")
    public List<Task> saveBulkTasks(List<Task> tasks) {
        return taskRepository.saveAll(tasks);
    }

    // Filter output collection - only return tasks owned by the current user
    @PostFilter("filterObject.assigneeId == authentication.principal.id or
hasRole('MANAGER')")
```

```
    public List<Task> getAllTasks() {
        return taskRepository.findAll();
    }

    // Combining PreFilter and PostFilter with other annotations
    @PreAuthorize("hasAuthority('TASK_ADMIN')")
    @PreFilter("filterObject.status != 'DELETED'")
    @PostFilter("filterObject.sensitive == false or hasRole('ADMIN')")
    public List<Task> processTasks(List<Task> tasks) {
        for (Task task : tasks) {
            task.setLastProcessed(new Date());
            taskRepository.save(task);
        }
        return tasks;
    }
}
```

Custom Security Expressions

Creating custom security expressions for reusable authorization logic:

```
@Component
public class CustomSecurityExpressions {

    private final ProjectRepository projectRepository;

    // Constructor injection

    public boolean isProjectOwner(Long projectId) {
        Authentication authentication =
SecurityContextHolder.getContext().getAuthentication();
        String currentUsername = authentication.getName();

        Project project = projectRepository.findById(projectId)
                .orElseThrow(() -> new ResourceNotFoundException("Project not
found"));

        return project.getOwnerUsername().equals(currentUsername);
    }

    public boolean isProjectMember(Long projectId) {
        Authentication authentication =
SecurityContextHolder.getContext().getAuthentication();
        String currentUsername = authentication.getName();

        return projectRepository.existsByIdAndMembersUsername(projectId,
currentUsername);
    }

    public boolean isWithinBusinessHours() {
        LocalTime currentTime = LocalTime.now();
```

```
        LocalDate currentDate = LocalDate.now();

        // Check if weekend
        if (currentDate.getDayOfWeek() == DayOfWeek.SATURDAY ||
            currentDate.getDayOfWeek() == DayOfWeek.SUNDAY) {
            return false;
        }

        // Check if within business hours (9 AM to 5 PM)
        return !currentTime.isBefore(LocalTime.of(9, 0)) &&
               !currentTime.isAfter(LocalTime.of(17, 0));
    }

    public boolean hasPermissionOnProject(Long projectId, String permission) {
        Authentication authentication =
SecurityContextHolder.getContext().getAuthentication();
        String currentUsername = authentication.getName();

        // Check if user has the specified permission on the project
        return projectRepository.hasPermission(projectId, currentUsername,
permission);
    }
}

@Configuration
@EnableMethodSecurity(prePostEnabled = true)
public class CustomMethodSecurityConfig {

    @Bean
    public MethodSecurityExpressionHandler methodSecurityExpressionHandler() {
        DefaultMethodSecurityExpressionHandler expressionHandler = new
DefaultMethodSecurityExpressionHandler();

        // Register custom security expression root
        expressionHandler.setExpressionParser(
                new SpelExpressionParser(new SpelParserConfiguration(true,
true)));
        expressionHandler.setPermissionEvaluator(new DomainPermissionEvaluator());

        return expressionHandler;
    }
}

@Service
public class ProjectService {

    // Use custom security expressions
    @PreAuthorize("@customSecurityExpressions.isProjectOwner(#projectId) or
hasRole('ADMIN')")
    public Project getProjectById(Long projectId) {
        return projectRepository.findById(projectId)
                .orElseThrow(() -> new ResourceNotFoundException("Project not
found"));
    }
```

```
    @PreAuthorize("@customSecurityExpressions.isProjectMember(#projectId) and " +
                  "@customSecurityExpressions.isWithinBusinessHours()")
    public List<Task> getProjectTasks(Long projectId) {
        return taskRepository.findByProjectId(projectId);
    }

    @PreAuthorize("@customSecurityExpressions.hasPermissionOnProject(#projectId,
'WRITE')")
    public Project updateProject(Long projectId, ProjectUpdateRequest
updateRequest) {
        Project project = projectRepository.findById(projectId)
                .orElseThrow(() -> new ResourceNotFoundException("Project not
found"));

        // Update project properties
        project.setName(updateRequest.getName());
        project.setDescription(updateRequest.getDescription());

        return projectRepository.save(project);
    }
}
```

Method Security with Annotations

Using custom annotations for cleaner method security:

```
// Custom permission annotation
@Target({ElementType.METHOD, ElementType.TYPE})
@Retention(RetentionPolicy.RUNTIME)
@PreAuthorize("hasPermission(#id, 'ENTITY', 'READ')")
public @interface CanRead {
}

@Target({ElementType.METHOD, ElementType.TYPE})
@Retention(RetentionPolicy.RUNTIME)
@PreAuthorize("hasPermission(#id, 'ENTITY', 'WRITE')")
public @interface CanWrite {
}

@Target({ElementType.METHOD, ElementType.TYPE})
@Retention(RetentionPolicy.RUNTIME)
@PreAuthorize("hasPermission(#id, 'ENTITY', 'DELETE')")
public @interface CanDelete {
}

// Project-specific annotations
@Target({ElementType.METHOD, ElementType.TYPE})
@Retention(RetentionPolicy.RUNTIME)
@PreAuthorize("@customSecurityExpressions.isProjectOwner(#projectId) or
hasRole('ADMIN')")
```

```
public @interface ProjectOwnerAccess {
}

@Target({ElementType.METHOD, ElementType.TYPE})
@Retention(RetentionPolicy.RUNTIME)
@PreAuthorize("@customSecurityExpressions.isProjectMember(#projectId)")
public @interface ProjectMemberAccess {
}

// Using custom annotations
@Service
public class AnnotatedProjectService {

    @ProjectOwnerAccess
    public Project getProjectById(Long projectId) {
        return projectRepository.findById(projectId)
                .orElseThrow(() -> new ResourceNotFoundException("Project not
found"));
    }

    @ProjectMemberAccess
    public List<Task> getProjectTasks(Long projectId) {
        return taskRepository.findByProjectId(projectId);
    }

    @CanWrite
    public Entity updateEntity(Long id, Entity entity) {
        return entityRepository.save(entity);
    }

    @CanDelete
    public void deleteEntity(Long id) {
        entityRepository.deleteById(id);
    }
}
```

CORS and CSRF Protection

Configure Cross-Origin Resource Sharing (CORS) and Cross-Site Request Forgery (CSRF) protection.

CORS Configuration

Allowing controlled cross-origin requests:

```
@Configuration
@EnableWebSecurity
public class CorsSecurityConfig {

    @Bean
    public SecurityFilterChain securityFilterChain(HttpSecurity http) throws
Exception {
```

```
        http
            .cors(cors -> cors.configurationSource(corsConfigurationSource()))
            .authorizeHttpRequests(authorize -> authorize
                .requestMatchers("/api/**").authenticated()
                .anyRequest().permitAll()
            );

        return http.build();
    }

    @Bean
    public CorsConfigurationSource corsConfigurationSource() {
        CorsConfiguration configuration = new CorsConfiguration();

        // Allow specific origins
        configuration.setAllowedOrigins(List.of(
                "https://example.com",
                "https://sub.example.com"
        ));

        // Alternatively, for development:
        // configuration.setAllowedOrigins(List.of("*"));

        // Allow specific HTTP methods
        configuration.setAllowedMethods(List.of(
                "GET", "POST", "PUT", "PATCH", "DELETE", "OPTIONS"
        ));

        // Allow specific headers
        configuration.setAllowedHeaders(List.of(
                "Authorization", "Content-Type", "X-Requested-With",
                "Accept", "Origin", "Access-Control-Request-Method",
                "Access-Control-Request-Headers"
        ));

        // Allow credentials (cookies)
        configuration.setAllowCredentials(true);

        // Expose headers to the client
        configuration.setExposedHeaders(List.of(
                "Access-Control-Allow-Origin",
                "Access-Control-Allow-Credentials",
                "Authorization"
        ));

        // Cache preflight response for 30 minutes
        configuration.setMaxAge(1800L);

        UrlBasedCorsConfigurationSource source = new
UrlBasedCorsConfigurationSource();
        source.registerCorsConfiguration("/api/**", configuration);

        return source;
    }
```

```
}

// Alternative @CrossOrigin annotation on controllers
@RestController
@RequestMapping("/api/products")
@CrossOrigin(origins = "https://example.com", maxAge = 3600)
public class ProductController {

    @GetMapping
    public List<Product> getAllProducts() {
        return productService.findAll();
    }

    @GetMapping("/{id}")
    public Product getProductById(@PathVariable Long id) {
        return productService.findById(id);
    }

    // Method-level CORS configuration overrides class-level
    @CrossOrigin(origins = "*")
    @PostMapping
    public Product createProduct(@RequestBody ProductDto productDto) {
        return productService.create(productDto);
    }
}

// Global CORS configuration using WebMvcConfigurer
@Configuration
public class WebConfig implements WebMvcConfigurer {

    @Override
    public void addCorsMappings(CorsRegistry registry) {
        registry.addMapping("/api/**")
                .allowedOrigins("https://example.com")
                .allowedMethods("GET", "POST", "PUT", "DELETE")
                .allowedHeaders("*")
                .allowCredentials(true)
                .maxAge(3600);

        registry.addMapping("/public/**")
                .allowedOrigins("*")
                .allowedMethods("GET")
                .maxAge(3600);
    }
}
```

CSRF Protection

Preventing Cross-Site Request Forgery attacks:

```
@Configuration
@EnableWebSecurity
public class CsrfSecurityConfig {

    @Bean
    public SecurityFilterChain securityFilterChain(HttpSecurity http) throws
Exception {
        http
            .csrf(csrf -> csrf
                // Custom CSRF token repository
                .csrfTokenRepository(csrfTokenRepository())

                // Custom CSRF request handler
                .csrfTokenRequestHandler(new CsrfTokenRequestAttributeHandler())

                // Ignore CSRF for specific paths (often API endpoints)
                .ignoringRequestMatchers(
                        "/api/auth/login",
                        "/api/webhook/**"
                )
            );

        return http.build();
    }

    @Bean
    public CsrfTokenRepository csrfTokenRepository() {
        // Use cookie-based CSRF tokens
        CookieCsrfTokenRepository repository =
CookieCsrfTokenRepository.withHttpOnlyFalse();
        repository.setCookieName("XSRF-TOKEN");        // Cookie name
        repository.setHeaderName("X-XSRF-TOKEN");      // Header name
        repository.setCookiePath("/");                 // Cookie path
        repository.setCookieMaxAge(Duration.ofHours(1)); // Cookie expiration
        return repository;
    }

    // Custom CSRF token repository with server-side storage
    public static class CustomCsrfTokenRepository implements CsrfTokenRepository {

        private final TokenRepository tokenRepository;

        public CustomCsrfTokenRepository(TokenRepository tokenRepository) {
            this.tokenRepository = tokenRepository;
        }

        @Override
        public CsrfToken generateToken(HttpServletRequest request) {
            String token = UUID.randomUUID().toString();
            return new DefaultCsrfToken("X-XSRF-TOKEN", "_csrf", token);
        }

        @Override
```

```
        public void saveToken(CsrfToken token, HttpServletRequest request,
HttpServletResponse response) {
            if (token == null) {
                // Delete existing token
                String sessionId = getSessionId(request);
                if (sessionId != null) {
                    tokenRepository.deleteToken(sessionId);
                }
                return;
            }

            // Save token in database linked to session ID
            String sessionId = getOrCreateSessionId(request, response);
            tokenRepository.saveToken(sessionId, token.getToken());

            // Also save token in cookie for JavaScript access
            Cookie cookie = new Cookie("XSRF-TOKEN", token.getToken());
            cookie.setPath("/");
            cookie.setHttpOnly(false); // Allow JavaScript access
            cookie.setMaxAge(3600);
            response.addCookie(cookie);
        }

        @Override
        public CsrfToken loadToken(HttpServletRequest request) {
            String sessionId = getSessionId(request);
            if (sessionId == null) {
                return null;
            }

            // Load token from database
            String token = tokenRepository.getToken(sessionId);
            if (token == null) {
                return null;
            }

            return new DefaultCsrfToken("X-XSRF-TOKEN", "_csrf", token);
        }

        private String getSessionId(HttpServletRequest request) {
            HttpSession session = request.getSession(false);
            return session != null ? session.getId() : null;
        }

        private String getOrCreateSessionId(HttpServletRequest request,
HttpServletResponse response) {
            HttpSession session = request.getSession(true);
            return session.getId();
        }
    }
}

// Custom CSRF protection for SPA (Single Page Applications)
@Component
```

```
public class SpaWebFilter extends OncePerRequestFilter {

    @Override
    protected void doFilterInternal(
            HttpServletRequest request,
            HttpServletResponse response,
            FilterChain filterChain) throws ServletException, IOException {

        CsrfToken csrfToken = (CsrfToken)
request.getAttribute(CsrfToken.class.getName());
        if (csrfToken != null) {
            // Add CSRF token to response headers for SPA clients
            response.setHeader(csrfToken.getHeaderName(), csrfToken.getToken());
        }

        filterChain.doFilter(request, response);
    }
}

// JavaScript example for using CSRF tokens in an SPA
/*
// Function to get CSRF token from cookies
function getCsrfToken() {
    const value = `; ${document.cookie}`;
    const parts = value.split(`; XSRF-TOKEN=`);
    if (parts.length === 2) return parts.pop().split(';').shift();
    return null;
}

// Add CSRF token to all AJAX requests
$.ajaxSetup({
    beforeSend: function(xhr) {
        const token = getCsrfToken();
        if (token) {
            xhr.setRequestHeader('X-XSRF-TOKEN', token);
        }
    }
});

// Example AJAX request with CSRF token
$.ajax({
    url: '/api/products',
    type: 'POST',
    data: JSON.stringify({ name: 'Product 1', price: 99.99 }),
    contentType: 'application/json',
    success: function(response) {
        console.log('Product created:', response);
    },
    error: function(xhr, status, error) {
        console.error('Error creating product:', error);
    }
});
*/
```

Complete Security Configuration

Combining CORS, CSRF, and other security features:

```
@Configuration
@EnableWebSecurity
public class WebSecurityConfig {

    private final CustomAuthenticationProvider authenticationProvider;
    private final CustomOAuth2UserService oAuth2UserService;
    private final JwtTokenProvider jwtTokenProvider;

    // Constructor injection

    @Bean
    public SecurityFilterChain securityFilterChain(HttpSecurity http) throws
Exception {
        http
            // CORS configuration
            .cors(cors -> cors.configurationSource(corsConfigurationSource()))

            // CSRF configuration
            .csrf(csrf -> csrf

.csrfTokenRepository(CookieCsrfTokenRepository.withHttpOnlyFalse())
                .ignoringRequestMatchers("/api/auth/**", "/api/webhook/**")
            )

            // HTTP security headers
            .headers(headers -> headers
                .contentSecurityPolicy(csp -> csp
                    .policyDirectives("default-src 'self'; script-src 'self'
https://trusted.cdn.com; img-src 'self' data:;")
                )
                .frameOptions(frame -> frame.deny())
                .xssProtection(xss -> xss.block(true))
                .contentTypeOptions(Customizer.withDefaults())
            )

            // Authorization configuration
            .authorizeHttpRequests(authorize -> authorize
                .requestMatchers("/", "/home", "/login", "/register", "/css/**",
"/js/**").permitAll()
                .requestMatchers("/api/public/**").permitAll()
                .requestMatchers("/api/auth/**").permitAll()
                .requestMatchers("/api/admin/**").hasRole("ADMIN")
                .requestMatchers("/api/user/**").hasRole("USER")
                .anyRequest().authenticated()
            )

            // Authentication methods
```

```
            .formLogin(form -> form
                .loginPage("/login")
                .defaultSuccessUrl("/dashboard")
                .failureUrl("/login?error=true")
                .permitAll()
            )

            .oauth2Login(oauth2 -> oauth2
                .loginPage("/login")
                .userInfoEndpoint(userInfo -> userInfo
                    .userService(oAuth2UserService)
                )
                .successHandler(oAuth2AuthenticationSuccessHandler())
                .failureHandler(oAuth2AuthenticationFailureHandler())
            )

            // JWT filter
            .addFilterBefore(
                new JwtAuthenticationFilter(jwtTokenProvider),
                UsernamePasswordAuthenticationFilter.class
            )

            // Session management
            .sessionManagement(session -> session
                .sessionCreationPolicy(SessionCreationPolicy.ALWAYS)
                .invalidSessionUrl("/login?expired=true")
                .maximumSessions(1)
                .maxSessionsPreventsLogin(false)
                .expiredUrl("/login?expired=true")
            )

            // Logout configuration
            .logout(logout -> logout
                .logoutUrl("/logout")
                .logoutSuccessUrl("/login?logout=true")
                .invalidateHttpSession(true)
                .deleteCookies("JSESSIONID", "remember-me")
                .permitAll()
            )

            // Remember-me functionality
            .rememberMe(remember -> remember
                .key("secure-remember-me-key")
                .tokenValiditySeconds(86400) // 1 day
            )

            // Exception handling
            .exceptionHandling(exceptions -> exceptions
                .accessDeniedPage("/access-denied")
                .authenticationEntryPoint(new CustomAuthenticationEntryPoint())
            );

        return http.build();
    }
```

```
    @Bean
    public AuthenticationManager authenticationManager(AuthenticationConfiguration
authConfig) throws Exception {
        return authConfig.getAuthenticationManager();
    }

    @Bean
    public CorsConfigurationSource corsConfigurationSource() {
        CorsConfiguration configuration = new CorsConfiguration();
        configuration.setAllowedOrigins(List.of("https://example.com"));
        configuration.setAllowedMethods(List.of("GET", "POST", "PUT", "DELETE"));
        configuration.setAllowedHeaders(List.of("*"));
        configuration.setAllowCredentials(true);

        UrlBasedCorsConfigurationSource source = new
UrlBasedCorsConfigurationSource();
        source.registerCorsConfiguration("/**", configuration);
        return source;
    }

    @Bean
    public AuthenticationSuccessHandler oAuth2AuthenticationSuccessHandler() {
        return new OAuth2AuthenticationSuccessHandler();
    }

    @Bean
    public AuthenticationFailureHandler oAuth2AuthenticationFailureHandler() {
        return new OAuth2AuthenticationFailureHandler();
    }

    @Bean
    public PasswordEncoder passwordEncoder() {
        return new BCryptPasswordEncoder();
    }
}
```

Modern Security Best Practices

Follow these best practices for secure Spring Boot applications.

Secure Password Handling

Properly encoding and verifying passwords:

```
@Configuration
public class PasswordConfig {

    @Bean
    public PasswordEncoder passwordEncoder() {
        // Use bcrypt with strength 12 (default is 10)
```

```
        return new BCryptPasswordEncoder(12);
    }
}

@Service
public class UserService {

    private final UserRepository userRepository;
    private final PasswordEncoder passwordEncoder;

    // Constructor injection

    @Transactional
    public User createUser(UserRegistrationDto registrationDto) {
        // Check if username already exists
        if (userRepository.existsByUsername(registrationDto.getUsername())) {
            throw new UsernameAlreadyExistsException("Username already exists");
        }

        // Check if email already exists
        if (userRepository.existsByEmail(registrationDto.getEmail())) {
            throw new EmailAlreadyExistsException("Email already exists");
        }

        // Create new user
        User user = new User();
        user.setUsername(registrationDto.getUsername());
        user.setEmail(registrationDto.getEmail());

        // Hash password before storing
        user.setPassword(passwordEncoder.encode(registrationDto.getPassword()));

        // Set default role
        Role userRole = roleRepository.findByName("ROLE_USER")
                .orElseThrow(() -> new RuntimeException("Default role not
found"));
        user.setRoles(Set.of(userRole));

        // Set account properties
        user.setEnabled(true);
        user.setCreatedAt(LocalDateTime.now());

        return userRepository.save(user);
    }

    @Transactional
    public void changePassword(Long userId, ChangePasswordRequest request) {
        User user = userRepository.findById(userId)
                .orElseThrow(() -> new ResourceNotFoundException("User not
found"));

        // Verify current password
        if (!passwordEncoder.matches(request.getCurrentPassword(),
user.getPassword())) {
```

```
            throw new InvalidPasswordException("Current password is incorrect");
        }

        // Hash and set new password
        user.setPassword(passwordEncoder.encode(request.getNewPassword()));
        user.setUpdatedAt(LocalDateTime.now());

        userRepository.save(user);
    }

    @Transactional
    public void resetPassword(String token, String newPassword) {
        // Find password reset token
        PasswordResetToken resetToken = resetTokenRepository.findByToken(token)
                .orElseThrow(() -> new InvalidTokenException("Invalid or expired
password reset token"));

        // Check if token is expired
        if (resetToken.getExpiryDate().isBefore(LocalDateTime.now())) {
            resetTokenRepository.delete(resetToken);
            throw new InvalidTokenException("Password reset token has expired");
        }

        // Update user's password
        User user = resetToken.getUser();
        user.setPassword(passwordEncoder.encode(newPassword));
        user.setUpdatedAt(LocalDateTime.now());

        userRepository.save(user);

        // Delete used token
        resetTokenRepository.delete(resetToken);
    }
}
```

Content Security Policy (CSP)

Protecting against XSS attacks with Content Security Policy:

```
@Configuration
@EnableWebSecurity
public class ContentSecurityPolicyConfig {

    @Bean
    public SecurityFilterChain securityFilterChain(HttpSecurity http) throws
Exception {
        http
            .headers(headers -> headers
                .contentSecurityPolicy(csp -> csp
                    .policyDirectives(
                        // Only allow resources from the same origin by default
```

```
                        "default-src 'self'; " +

                        // Allow scripts from same origin and specific trusted
CDNs
                        "script-src 'self' https://cdnjs.cloudflare.com
https://cdn.jsdelivr.net; " +

                        // Allow styles from same origin and specific trusted CDNs
                        "style-src 'self' https://fonts.googleapis.com
https://cdn.jsdelivr.net; " +

                        // Allow fonts from same origin and specific trusted
sources
                        "font-src 'self' https://fonts.gstatic.com; " +

                        // Allow images from same origin, data URLs, and specific
trusted sources
                        "img-src 'self' data: https://secure.example.com; " +

                        // Restrict form submissions to same origin
                        "form-action 'self'; " +

                        // Restrict frame embedding
                        "frame-ancestors 'self'; " +

                        // Block mixed content
                        "block-all-mixed-content; " +

                        // Require HTTPS for all subresources
                        "upgrade-insecure-requests;"
                    )
                )
            );

        return http.build();
    }
}

// Alternatively, use headers in the application properties
// application.properties:
// server.servlet.session.cookie.secure=true
// server.servlet.session.cookie.http-only=true
// server.servlet.session.cookie.same-site=strict
```

Rate Limiting

Protecting against brute force attacks with rate limiting:

```
@Component
public class RateLimitingFilter extends OncePerRequestFilter {
```

```
    private static final int MAX_REQUESTS_PER_MINUTE = 60;

    private final Map<String, List<LocalDateTime>> requestTimestamps = new
ConcurrentHashMap<>();

    @Override
    protected void doFilterInternal(
            HttpServletRequest request,
            HttpServletResponse response,
            FilterChain filterChain) throws ServletException, IOException {

        // Get client IP address
        String clientIp = getClientIp(request);

        // Rate limiting for specific endpoints
        if (request.getRequestURI().startsWith("/api/")) {
            if (isRateLimited(clientIp)) {
                // Too many requests
                response.setStatus(HttpStatus.TOO_MANY_REQUESTS.value());
                response.setContentType(MediaType.APPLICATION_JSON_VALUE);

                ObjectMapper mapper = new ObjectMapper();
                mapper.writeValue(response.getWriter(), Map.of(
                        "status", 429,
                        "error", "Too Many Requests",
                        "message", "Rate limit exceeded. Please try again later."
                ));
                return;
            }
        }

        // Continue with the filter chain
        filterChain.doFilter(request, response);
    }

    private boolean isRateLimited(String clientIp) {
        LocalDateTime now = LocalDateTime.now();

        // Clean up old timestamps
        requestTimestamps.computeIfPresent(clientIp, (ip, timestamps) -> {
            LocalDateTime oneMinuteAgo = now.minusMinutes(1);
            return timestamps.stream()
                    .filter(timestamp -> timestamp.isAfter(oneMinuteAgo))
                    .collect(Collectors.toList());
        });

        // Get current timestamps for client
        List<LocalDateTime> timestamps = requestTimestamps.computeIfAbsent(
                clientIp, k -> new ArrayList<>());

        // Check if too many requests within the time window
        if (timestamps.size() >= MAX_REQUESTS_PER_MINUTE) {
            return true; // Rate limited
        }
```

```
        // Add current timestamp
        timestamps.add(now);
        return false; // Not rate limited
    }

    private String getClientIp(HttpServletRequest request) {
        String xForwardedFor = request.getHeader("X-Forwarded-For");
        if (xForwardedFor != null && !xForwardedFor.isEmpty()) {
            // Get first IP in case of multiple proxies
            return xForwardedFor.split(",")[0].trim();
        }
        return request.getRemoteAddr();
    }
}

// More sophisticated rate limiting with Redis
@Service
public class RedisRateLimitService {

    private final StringRedisTemplate redisTemplate;

    // Constructor injection

    public boolean allowRequest(String key, int maxRequests, int
timeWindowSeconds) {
        String redisKey = "rate_limit:" + key;

        // Get current count
        Long currentCount = redisTemplate.opsForValue().increment(redisKey);

        // Set expiry on first request
        if (currentCount != null && currentCount == 1) {
            redisTemplate.expire(redisKey, timeWindowSeconds, TimeUnit.SECONDS);
        }

        // Check if limit exceeded
        return currentCount != null && currentCount <= maxRequests;
    }
}

@Component
public class AuthenticationFailureListener {

    private final RedisRateLimitService rateLimitService;
    private final UserService userService;

    // Constructor injection

    @EventListener
    public void onAuthenticationFailure(AuthenticationFailureBadCredentialsEvent
event) {
        String username = event.getAuthentication().getName();
        String clientIp = ((WebAuthenticationDetails)
```

```
event.getAuthentication().getDetails()).getRemoteAddress();

        // Rate limit by username (prevent username enumeration)
        boolean usernameAllowed = rateLimitService.allowRequest(
                "auth:username:" + username, 5, 300); // 5 attempts per 5 minutes

        // Rate limit by IP (prevent distributed attacks)
        boolean ipAllowed = rateLimitService.allowRequest(
                "auth:ip:" + clientIp, 10, 300); // 10 attempts per 5 minutes

        if (!usernameAllowed && userService.existsByUsername(username)) {
            // Lock account if too many failed attempts
            userService.lockAccount(username);
        }
    }
}
```

Secure HTTP Headers

Setting secure headers for all responses:

```
@Configuration
@EnableWebSecurity
public class SecureHeadersConfig {

    @Bean
    public SecurityFilterChain securityFilterChain(HttpSecurity http) throws
Exception {
        http
            .headers(headers -> headers
                // X-Content-Type-Options: nosniff
                // Prevents browsers from interpreting files as a different MIME
type
                .contentTypeOptions(Customizer.withDefaults())

                // X-Frame-Options: DENY
                // Prevents clickjacking attacks by denying framing
                .frameOptions(frame -> frame.deny())

                // X-XSS-Protection: 1; mode=block
                // Enables XSS filtering in browsers
                .xssProtection(xss -> xss.block(true))

                // Strict-Transport-Security: max-age=31536000; includeSubDomains
                // Forces HTTPS usage
                .httpStrictTransportSecurity(hsts -> hsts
                    .includeSubDomains(true)
                    .maxAgeInSeconds(31536000)
                )

                // Referrer-Policy: strict-origin-when-cross-origin
```

```
                // Controls how much referrer information is included with
requests
                .referrerPolicy(referrer -> referrer

.policy(ReferrerPolicyHeaderWriter.ReferrerPolicy.STRICT_ORIGIN_WHEN_CROSS_ORIGIN)
                )

                // Permissions-Policy: geolocation=(), camera=(), microphone=()
                // Controls browser features available to the site
                .permissionsPolicy(permissions -> permissions
                    .policy("geolocation=(), camera=(), microphone=()")
                )

                // Content-Security-Policy
                .contentSecurityPolicy(csp -> csp
                    .policyDirectives("default-src 'self'; script-src 'self'")
                )
            );

        return http.build();
    }
}
```

Secure Cookie Configuration

Setting secure cookies for session management:

```
@Configuration
public class CookieSecurityConfig {

    @Bean
    public CookieSerializer cookieSerializer() {
        DefaultCookieSerializer serializer = new DefaultCookieSerializer();
        serializer.setCookieName("SESSIONID");         // Custom session cookie
name
        serializer.setCookiePath("/");                  // Cookie path
        serializer.setDomainNamePattern("^.+?\\.(\\w+\\.[a-z]+)$"); // Restrict to
subdomains
        serializer.setCookieMaxAge(3600);              // Cookie lifespan in
seconds
        serializer.setUseHttpOnlyCookie(true);         // Prevent JavaScript
access
        serializer.setUseSecureCookie(true);           // Require HTTPS
        serializer.setSameSite("Strict");              // SameSite attribute
        return serializer;
    }

    @Bean
    public CookieHttpSessionIdResolver httpSessionIdResolver() {
        CookieHttpSessionIdResolver resolver = new CookieHttpSessionIdResolver();
        resolver.setCookieSerializer(cookieSerializer());
```

```
        return resolver;
    }
}

// Alternatively, use properties for configuration
// application.properties:
// server.servlet.session.cookie.name=SESSIONID
// server.servlet.session.cookie.path=/
// server.servlet.session.cookie.domain=example.com
// server.servlet.session.cookie.max-age=3600
// server.servlet.session.cookie.http-only=true
// server.servlet.session.cookie.secure=true
// server.servlet.session.cookie.same-site=strict
```

Security Hardening for Production

Additional security measures for production environments:

```
@Configuration
@Profile("production")
public class ProductionSecurityConfig {

    @Bean
    public WebSecurityCustomizer webSecurityCustomizer() {
        return web -> web.ignoring()
                .requestMatchers(
                        PathRequest.toStaticResources().atCommonLocations()
                );
    }

    @Bean
    public SecurityFilterChain productionSecurityFilterChain(HttpSecurity http)
throws Exception {
        http
            // Require HTTPS for all requests
            .requiresChannel(channel -> channel
                .anyRequest().requiresSecure()
            )

            // Session management
            .sessionManagement(session -> session
                .sessionFixation().migrateSession()
                .maximumSessions(1)
                .maxSessionsPreventsLogin(true)
                .expiredUrl("/login?expired=true")
            )

            // Enable enhanced protection against common vulnerabilities
            .headers(headers -> headers
                .contentSecurityPolicy(csp -> csp
                    .policyDirectives("default-src 'self'; script-src 'self'")
```

```
                )
                .httpStrictTransportSecurity(hsts -> hsts
                    .includeSubDomains(true)
                    .maxAgeInSeconds(31536000)
                )
                .frameOptions(frame -> frame.deny())
                .xssProtection(xss -> xss.block(true))
                .contentTypeOptions(Customizer.withDefaults())
            );

        return http.build();
    }

    @Bean
    public HttpFirewall httpFirewall() {
        // Prevent URL path traversal and other URL-based attacks
        StrictHttpFirewall firewall = new StrictHttpFirewall();
        firewall.setAllowUrlEncodedSlash(false);
        firewall.setAllowSemicolon(false);
        firewall.setAllowBackSlash(false);
        firewall.setAllowUrlEncodedPercent(false);
        firewall.setAllowUrlEncodedPeriod(false);
        return firewall;
    }

    @Bean
    public WebSecurityCustomizer firewallCustomizer() {
        return web -> web.httpFirewall(httpFirewall());
    }
}

// Secure file uploads
@Service
public class SecureFileUploadService {

    private static final long MAX_FILE_SIZE = 10 * 1024 * 1024; // 10 MB
    private static final Set<String> ALLOWED_EXTENSIONS = Set.of(
            "jpg", "jpeg", "png", "pdf", "doc", "docx", "xls", "xlsx");
    private static final Set<String> ALLOWED_CONTENT_TYPES = Set.of(
            "image/jpeg", "image/png", "application/pdf",
            "application/msword", "application/vnd.openxmlformats-
officedocument.wordprocessingml.document",
            "application/vnd.ms-excel", "application/vnd.openxmlformats-
officedocument.spreadsheetml.sheet");

    private final TikaConfig tikaConfig = TikaConfig.getDefaultConfig();

    public String storeFile(MultipartFile file) throws IOException,
SecurityException {
        // Check if file is empty
        if (file.isEmpty()) {
            throw new IllegalArgumentException("Cannot upload empty file");
        }
```

```
        // Check file size
        if (file.getSize() > MAX_FILE_SIZE) {
            throw new SecurityException("File size exceeds maximum limit");
        }

        // Check file extension
        String extension = getFileExtension(file.getOriginalFilename());
        if (!ALLOWED_EXTENSIONS.contains(extension.toLowerCase())) {
            throw new SecurityException("File type not allowed");
        }

        // Verify content type
        String contentType = getActualContentType(file);
        if (!ALLOWED_CONTENT_TYPES.contains(contentType)) {
            throw new SecurityException("File content type not allowed");
        }

        // Generate a secure random filename to prevent path traversal
        String newFilename = UUID.randomUUID().toString() + "." + extension;

        // Create the upload directory if it doesn't exist
        Path uploadDir = Paths.get("uploads");
        if (!Files.exists(uploadDir)) {
            Files.createDirectories(uploadDir);
        }

        // Save the file
        Path targetLocation = uploadDir.resolve(newFilename);
        Files.copy(file.getInputStream(), targetLocation,
StandardCopyOption.REPLACE_EXISTING);

        return newFilename;
    }

    private String getFileExtension(String filename) {
        if (filename == null || filename.lastIndexOf(".") == -1) {
            return "";
        }
        return filename.substring(filename.lastIndexOf(".") + 1);
    }

    private String getActualContentType(MultipartFile file) throws IOException {
        // Use Apache Tika to detect the actual content type
        Detector detector = tikaConfig.getDetector();
        TikaInputStream stream = TikaInputStream.get(file.getInputStream());
        Metadata metadata = new Metadata();
        metadata.add(Metadata.RESOURCE_NAME_KEY, file.getOriginalFilename());
        MediaType mediaType = detector.detect(stream, metadata);
        return mediaType.toString();
    }
}
```

E. Microservices with Spring Boot

Microservices Architecture Principles

Microservices architecture is an approach to building software systems as a collection of small, independently deployable services.

Core Principles

1. **Single Responsibility**: Each microservice should focus on a specific business capability.
2. **Autonomy**: Services can be developed, deployed, and scaled independently.
3. **Resilience**: Failure in one service should not cascade to others.
4. **Decentralization**: Teams own their services end-to-end.
5. **Domain-Driven Design**: Services are organized around business domains.

Microservices vs. Monolithic Architecture

Aspect	Monolithic Architecture	Microservices Architecture
Development	Single codebase, easier to develop initially	Multiple codebases, complex interactions
Deployment	All-or-nothing deployment	Independent service deployment
Scaling	Scale the entire application	Scale individual services as needed
Technology	Single technology stack	Technology diversity
Team Structure	Organized by technical layers	Organized by business capability
Resilience	Single point of failure	Isolated failures
Complexity	Simpler initially, complex over time	Complex initially, manageable over time

Designing Microservices

1. **Bounded Context**: Define clear boundaries between services.
2. **Service Interface**: Design APIs that are stable and backward compatible.
3. **Data Management**: Each service owns its data, no shared databases.
4. **Communication**: Services communicate via HTTP/REST, messaging, or gRPC.
5. **Authentication/Authorization**: Implement security at service and API gateway levels.

Spring Boot for Microservices

Spring Boot is well-suited for microservices development:

1. **Standalone**: Each Spring Boot application runs independently.
2. **Embedded Servers**: No need for external application servers.
3. **Auto-Configuration**: Minimal configuration required.
4. **Production-Ready**: Built-in monitoring and metrics.
5. **Spring Cloud Integration**: Support for distributed systems patterns.

Sample Microservices Application

A typical e-commerce microservices architecture:

```
E-Commerce Microservices
├── product-service          // Product catalog management
├── inventory-service        // Inventory management
├── order-service            // Order processing
├── payment-service          // Payment processing
├── shipping-service         // Shipping management
├── user-service             // User account management
├── notification-service     // Email/SMS notifications
├── api-gateway              // Single entry point for clients
├── discovery-service        // Service registry and discovery
├── config-server            // Centralized configuration
├── auth-service             // Authentication and authorization
└── monitoring-service       // System monitoring and alerts
```

Building a Microservice with Spring Boot

Basic structure of a Spring Boot microservice:

```
@SpringBootApplication
@EnableDiscoveryClient  // For service registration
public class ProductServiceApplication {

    public static void main(String[] args) {
        SpringApplication.run(ProductServiceApplication.class, args);
    }
}

@RestController
@RequestMapping("/api/products")
public class ProductController {

    private final ProductService productService;

    // Constructor injection

    @GetMapping
    public ResponseEntity<List<ProductDto>> getAllProducts() {
        return ResponseEntity.ok(productService.findAll());
    }

    @GetMapping("/{id}")
    public ResponseEntity<ProductDto> getProductById(@PathVariable String id) {
        return ResponseEntity.ok(productService.findById(id));
    }

    @PostMapping
    public ResponseEntity<ProductDto> createProduct(@Valid @RequestBody ProductDto
productDto) {
```

```
        return ResponseEntity.status(HttpStatus.CREATED)
                .body(productService.createProduct(productDto));
    }

    @PutMapping("/{id}")
    public ResponseEntity<ProductDto> updateProduct(
            @PathVariable String id,
            @Valid @RequestBody ProductDto productDto) {
        return ResponseEntity.ok(productService.updateProduct(id, productDto));
    }

    @DeleteMapping("/{id}")
    public ResponseEntity<Void> deleteProduct(@PathVariable String id) {
        productService.deleteProduct(id);
        return ResponseEntity.noContent().build();
    }
}

@Service
public class ProductService {

    private final ProductRepository productRepository;
    private final ProductMapper productMapper;

    // Constructor injection

    public List<ProductDto> findAll() {
        return productRepository.findAll().stream()
                .map(productMapper::toDto)
                .collect(Collectors.toList());
    }

    public ProductDto findById(String id) {
        Product product = productRepository.findById(id)
                .orElseThrow(() -> new ResourceNotFoundException("Product not
found"));

        return productMapper.toDto(product);
    }

    public ProductDto createProduct(ProductDto productDto) {
        // Validate product data
        // Map DTO to entity
        Product product = productMapper.toEntity(productDto);

        // Save entity
        Product savedProduct = productRepository.save(product);

        // Map entity back to DTO
        return productMapper.toDto(savedProduct);
    }

    // Other service methods
}
```

Spring Cloud Overview

Spring Cloud provides tools for common distributed system patterns needed in microservices architectures.

Spring Cloud Components

1. **Service Discovery**: Register and locate services dynamically.
2. **Configuration Management**: Externalize and centralize configuration.
3. **API Gateway**: Single entry point with routing and filtering.
4. **Circuit Breaker**: Handle failures gracefully.
5. **Distributed Tracing**: Track request flow across services.
6. **Load Balancing**: Distribute traffic across service instances.
7. **Messaging**: Facilitate asynchronous communication.

Setting Up Spring Cloud

```
<!-- Parent POM dependencies -->
<dependencyManagement>
    <dependencies>
        <dependency>
            <groupId>org.springframework.cloud</groupId>
            <artifactId>spring-cloud-dependencies</artifactId>
            <version>${spring-cloud.version}</version>
            <type>pom</type>
            <scope>import</scope>
        </dependency>
    </dependencies>
</dependencyManagement>
```

Spring Cloud Configuration Server

Centralized configuration management:

```
<!-- Configuration Server dependency -->
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-config-server</artifactId>
</dependency>
```

```
@SpringBootApplication
@EnableConfigServer
public class ConfigServerApplication {

    public static void main(String[] args) {
```

```
        SpringApplication.run(ConfigServerApplication.class, args);
    }
}
```

```
# application.properties
server.port=8888
spring.application.name=config-server

# Git backend
spring.cloud.config.server.git.uri=https://github.com/myorg/config-repo
spring.cloud.config.server.git.search-paths=config-data
spring.cloud.config.server.git.username=username
spring.cloud.config.server.git.password=password
spring.cloud.config.server.git.default-label=main
```

Spring Cloud Config Client

Consuming configuration from a Configuration Server:

```
<!-- Configuration Client dependency -->
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-config</artifactId>
</dependency>
```

```
# bootstrap.properties
spring.application.name=product-service
spring.profiles.active=dev
spring.cloud.config.uri=http://localhost:8888
spring.cloud.config.fail-fast=true
```

```
@RestController
@RefreshScope  // Enable runtime configuration refresh
public class ConfigController {

    @Value("${app.feature.enabled:false}")
    private boolean featureEnabled;

    @Value("${app.max-items:100}")
    private int maxItems;

    @GetMapping("/config")
    public Map<String, Object> getConfig() {
        return Map.of(
```

```
                "featureEnabled", featureEnabled,
                "maxItems", maxItems
        );
    }
}
```

Spring Cloud Netflix Eureka

Service discovery and registration:

```
<!-- Eureka Server dependency -->
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-netflix-eureka-server</artifactId>
</dependency>
```

```
@SpringBootApplication
@EnableEurekaServer
public class DiscoveryServiceApplication {

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

```
# application.properties
server.port=8761
eureka.client.register-with-eureka=false
eureka.client.fetch-registry=false
```

Eureka Client configuration:

```
<!-- Eureka Client dependency -->
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-netflix-eureka-client</artifactId>
</dependency>
```

```
@SpringBootApplication
@EnableDiscoveryClient
public class ProductServiceApplication {
```

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

```
# application.properties
spring.application.name=product-service
server.port=0  # Random port for multiple instances
eureka.client.service-url.defaultZone=http://localhost:8761/eureka/
eureka.instance.instance-id=${spring.application.name}:${random.value}
```

Spring Cloud Gateway

API gateway for routing and filtering:

```
<!-- Gateway dependency -->
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-gateway</artifactId>
</dependency>
```

```
@SpringBootApplication
@EnableDiscoveryClient
public class ApiGatewayApplication {

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

```
# application.yml
spring:
  cloud:
    gateway:
      routes:
        - id: product-service
          uri: lb://PRODUCT-SERVICE
          predicates:
            - Path=/api/products/**
          filters:
            - RewritePath=/api/(?<segment>.*), /$\{segment}

        - id: order-service
          uri: lb://ORDER-SERVICE
```

```
          predicates:
            - Path=/api/orders/**
          filters:
            - RewritePath=/api/(?<segment>.*), /$\{segment}

        - id: user-service
          uri: lb://USER-SERVICE
          predicates:
            - Path=/api/users/**
          filters:
            - RewritePath=/api/(?<segment>.*), /$\{segment}
```

Spring Cloud Circuit Breaker

Fault tolerance with Circuit Breaker pattern:

```
<!-- Resilience4J dependency -->
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-circuitbreaker-resilience4j</artifactId>
</dependency>
```

```
@Service
public class ProductCompositeService {

    private final RestTemplate restTemplate;
    private final CircuitBreakerFactory circuitBreakerFactory;

    // Constructor injection

    public ProductDto getProduct(String productId) {
        CircuitBreaker circuitBreaker =
circuitBreakerFactory.create("productService");

        return circuitBreaker.run(
                () -> restTemplate.getForObject(
                        "http://product-service/products/{id}",
                        ProductDto.class,
                        productId
                ),
                throwable -> getProductFallback(productId, throwable)
        );
    }

    private ProductDto getProductFallback(String productId, Throwable throwable) {
        // Log the error
        log.error("Error retrieving product {}: {}", productId,
throwable.getMessage());
```

```
        // Return a default/fallback product
        return new ProductDto(productId, "Fallback Product", "Temporary
unavailable", BigDecimal.ZERO);
    }
}
```

Spring Cloud Sleuth and Zipkin

Distributed tracing:

```
<!-- Sleuth and Zipkin dependencies -->
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-sleuth</artifactId>
</dependency>
<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
```

```
@Service
public class OrderService {

    private static final Logger log = LoggerFactory.getLogger(OrderService.class);

    private final OrderRepository orderRepository;
    private final RestTemplate restTemplate;

    // Constructor injection

    public OrderDto createOrder(OrderRequest orderRequest) {
        log.info("Creating new order for customer: {}",
orderRequest.getCustomerId());

        // Check product availability
        ProductDto product = restTemplate.getForObject(
                "http://product-service/products/{id}",
                ProductDto.class,
                orderRequest.getProductId()
        );

        log.info("Product information retrieved: {}", product.getName());
```

```
        // Create new order
        Order order = new Order();
        order.setCustomerId(orderRequest.getCustomerId());
        order.setProductId(orderRequest.getProductId());
        order.setQuantity(orderRequest.getQuantity());
        order.setTotalPrice(product.getPrice().multiply(new
BigDecimal(orderRequest.getQuantity())));
        order.setStatus(OrderStatus.CREATED);
        order.setCreatedAt(LocalDateTime.now());

        Order savedOrder = orderRepository.save(order);
        log.info("Order created with ID: {}", savedOrder.getId());

        // Process payment
        PaymentRequest paymentRequest = new PaymentRequest(
                savedOrder.getId(),
                savedOrder.getCustomerId(),
                savedOrder.getTotalPrice()
        );

        PaymentResponse paymentResponse = restTemplate.postForObject(
                "http://payment-service/payments",
                paymentRequest,
                PaymentResponse.class
        );

        log.info("Payment processed with status: {}",
paymentResponse.getStatus());

        // Update order status
        savedOrder.setStatus(OrderStatus.PAID);
        savedOrder = orderRepository.save(savedOrder);

        return orderMapper.toDto(savedOrder);
    }
}
```

With Spring Cloud Sleuth, each microservice request gets a unique trace ID and span ID, making it possible to trace requests across multiple services. Zipkin provides a visualization layer to see the complete request flow.

Spring Cloud Bus

Distributing configuration changes across services:

```
<!-- Spring Cloud Bus dependencies -->
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-bus-amqp</artifactId>
</dependency>
```

```
# application.properties
spring.rabbitmq.host=localhost
spring.rabbitmq.port=5672
spring.rabbitmq.username=guest
spring.rabbitmq.password=guest
```

Spring Cloud Stream

Event-driven communication between services:

```
<!-- Spring Cloud Stream dependencies -->
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-stream</artifactId>
</dependency>
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-stream-binder-rabbit</artifactId>
</dependency>
```

```
@Configuration
public class StreamConfig {

    @Bean
    public Function<OrderCreatedEvent, ProductReservationEvent> processOrder() {
        return orderCreatedEvent -> {
            // Process the order event
            log.info("Processing order: {}", orderCreatedEvent.getOrderId());

            // Return a product reservation event
            return new ProductReservationEvent(
                    orderCreatedEvent.getOrderId(),
                    orderCreatedEvent.getProductId(),
                    orderCreatedEvent.getQuantity()
            );
        };
    }
}
```

```
# application.yml
spring:
  cloud:
    stream:
      function:
        definition: processOrder
      bindings:
```

```
        processOrder-in-0:
          destination: order-events
          group: order-processing-group
        processOrder-out-0:
          destination: product-events
```

Spring Cloud Security

Securing microservices with OAuth2 and JWT:

```
<!-- Spring Cloud Security dependencies -->
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-oauth2-resource-server</artifactId>
</dependency>
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-oauth2-client</artifactId>
</dependency>
```

```
@Configuration
@EnableWebSecurity
public class ResourceServerConfig extends WebSecurityConfigurerAdapter {

    @Override
    protected void configure(HttpSecurity http) throws Exception {
        http
            .authorizeRequests(authorize -> authorize
                .anyRequest().authenticated()
            )
            .oauth2ResourceServer(oauth2 -> oauth2
                .jwt(jwt -> jwt
                    .jwtAuthenticationConverter(jwtAuthenticationConverter())
                )
            );
    }

    private JwtAuthenticationConverter jwtAuthenticationConverter() {
        JwtGrantedAuthoritiesConverter jwtGrantedAuthoritiesConverter = new
JwtGrantedAuthoritiesConverter();
        jwtGrantedAuthoritiesConverter.setAuthoritiesClaimName("roles");
        jwtGrantedAuthoritiesConverter.setAuthorityPrefix("ROLE_");

        JwtAuthenticationConverter jwtAuthenticationConverter = new
JwtAuthenticationConverter();

jwtAuthenticationConverter.setJwtGrantedAuthoritiesConverter(jwtGrantedAuthorities
Converter);
```

```
        return jwtAuthenticationConverter;
    }
}
```

```
# application.properties
spring.security.oauth2.resourceserver.jwt.issuer-uri=http://auth-
server:9000/auth/realms/microservices
```

Service Discovery

Service Discovery allows microservices to find and communicate with each other without hardcoding hostnames and ports.

Implementing Eureka Server

Eureka is Netflix's service discovery server and client:

```
<!-- Eureka Server dependency -->
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-netflix-eureka-server</artifactId>
</dependency>
```

```
@SpringBootApplication
@EnableEurekaServer
public class DiscoveryServiceApplication {

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

```
# application.properties
server.port=8761
eureka.client.register-with-eureka=false
eureka.client.fetch-registry=false
eureka.server.wait-time-in-ms-when-sync-empty=0

# Security (optional)
spring.security.user.name=eureka
spring.security.user.password=password
```

Implementing Eureka Client

Each microservice registers with Eureka:

```
<!-- Eureka Client dependency -->
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-netflix-eureka-client</artifactId>
</dependency>
```

```
@SpringBootApplication
@EnableDiscoveryClient  // or @EnableEurekaClient
public class ProductServiceApplication {

    public static void main(String[] args) {
        SpringApplication.run(ProductServiceApplication.class, args);
    }

    @Bean
    @LoadBalanced  // Enables client-side load balancing
    public RestTemplate restTemplate() {
        return new RestTemplate();
    }
}
```

```
# application.properties
spring.application.name=product-service
server.port=0  # Random port for multiple instances

# Eureka client configuration
eureka.client.service-
url.defaultZone=http://eureka:password@localhost:8761/eureka/
eureka.instance.instance-id=${spring.application.name}:${random.uuid}
eureka.instance.prefer-ip-address=true

# Health check
eureka.client.healthcheck.enabled=true
```

Service Discovery Patterns

1. **Client-Side Discovery**:

```
@Service
public class OrderService {

    private final RestTemplate restTemplate;
```

```
    // Constructor injection

    public OrderDto getOrderWithProductDetails(String orderId) {
        // Find order
        Order order = orderRepository.findById(orderId)
                .orElseThrow(() -> new ResourceNotFoundException("Order not
found"));

        // Call product service (client-side discovery via @LoadBalanced
RestTemplate)
        ProductDto product = restTemplate.getForObject(
                "http://product-service/products/{id}",  // Service name, not
hostname
                ProductDto.class,
                order.getProductId()
        );

        // Build order DTO with product details
        OrderDto orderDto = orderMapper.toDto(order);
        orderDto.setProductDetails(product);

        return orderDto;
    }
}
```

2. **Server-Side Discovery**:

With an API Gateway or Load Balancer in front of services, clients make requests to the gateway which performs service discovery and routing.

```
// This happens in the API Gateway
@Bean
public RouteLocator customRouteLocator(RouteLocatorBuilder builder) {
    return builder.routes()
            .route("product_route", r -> r.path("/api/products/**")
                    .filters(f -> f.rewritePath("/api/(?<segment>.*)",
"/${segment}"))
                    .uri("lb://product-service"))  // Load balanced URI
            .route("order_route", r -> r.path("/api/orders/**")
                    .filters(f -> f.rewritePath("/api/(?<segment>.*)",
"/${segment}"))
                    .uri("lb://order-service"))
            .build();
}
```

Service Instance Metadata

Adding custom metadata to service instances:

```
# application.properties
eureka.instance.metadata-map.zone=zone1
eureka.instance.metadata-map.version=1.0
eureka.instance.metadata-map.environment=production
```

Health Checks and Status Updates

```
@Component
public class EurekaHealthCheckHandler extends HealthCheckHandler {

    private final HealthAggregator healthAggregator;
    private final List<HealthIndicator> healthIndicators;

    // Constructor injection with all HealthIndicator beans

    @Override
    public Status getStatus(String instanceId) {
        // Get health from all indicators
        Map<String, Status> healthResults = new HashMap<>();

        for (HealthIndicator indicator : healthIndicators) {
            String name = indicator.getClass().getSimpleName();
            healthResults.put(name, indicator.health().getStatus());
        }

        // Aggregate health statuses
        return healthAggregator.aggregateStatus(healthResults);
    }
}

@Component
public class DatabaseHealthIndicator implements HealthIndicator {

    private final DataSource dataSource;

    // Constructor injection

    @Override
    public Health health() {
        try (Connection conn = dataSource.getConnection()) {
            // Execute simple query to check database
            try (Statement stmt = conn.createStatement()) {
                stmt.execute("SELECT 1");
            }

            return Health.up().withDetail("database", "Available").build();
        } catch (Exception e) {
            return Health.down()
                    .withDetail("database", "Unavailable")
                    .withDetail("error", e.getMessage())
```

```
                    .build();
        }
    }
}
```

API Gateway Implementation

API Gateway serves as a single entry point for client applications to access microservices.

Implementing Spring Cloud Gateway

Spring Cloud Gateway provides a modern, non-blocking API Gateway:

```
<!-- Spring Cloud Gateway dependency -->
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-gateway</artifactId>
</dependency>
```

```
@SpringBootApplication
@EnableDiscoveryClient
public class ApiGatewayApplication {

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

Routing Configuration

Configuring routes to microservices:

```
# application.yml
spring:
  cloud:
    gateway:
      discovery:
        locator:
          enabled: true
          lower-case-service-id: true
      routes:
        - id: product-service
          uri: lb://product-service
          predicates:
            - Path=/api/products/**
          filters:
```

```
            - RewritePath=/api/(?<segment>.*), /$\{segment}
            - name: CircuitBreaker
              args:
                name: productServiceCircuitBreaker
                fallbackUri: forward:/fallback/products

        - id: order-service
          uri: lb://order-service
          predicates:
            - Path=/api/orders/**
          filters:
            - RewritePath=/api/(?<segment>.*), /$\{segment}
            - name: CircuitBreaker
              args:
                name: orderServiceCircuitBreaker
                fallbackUri: forward:/fallback/orders

        - id: user-service
          uri: lb://user-service
          predicates:
            - Path=/api/users/**
            - Method=GET,POST,PUT,DELETE
          filters:
            - RewritePath=/api/(?<segment>.*), /$\{segment}
```

Predicate Factories

Using predicates to route requests based on various conditions:

```
# application.yml
spring:
  cloud:
    gateway:
      routes:
        - id: path-route
          uri: lb://service-a
          predicates:
            - Path=/service-a/**

        - id: method-route
          uri: lb://service-b
          predicates:
            - Method=GET,POST

        - id: host-route
          uri: lb://service-c
          predicates:
            - Host=**.example.org

        - id: header-route
          uri: lb://service-d
```

```
          predicates:
            - Header=X-API-Version, v2

        - id: query-route
          uri: lb://service-e
          predicates:
            - Query=version, v2

        - id: datetime-route
          uri: lb://service-f
          predicates:
            - Between=2023-01-01T00:00:00+00:00, 2023-12-31T23:59:59+00:00

        - id: cookie-route
          uri: lb://service-g
          predicates:
            - Cookie=session, user-.*

        - id: weight-route
          uri: lb://service-h
          predicates:
            - Weight=group1, 8
```

Gateway Filter Factories

Using filters to modify requests and responses:

```
# application.yml
spring:
  cloud:
    gateway:
      routes:
        - id: product-service
          uri: lb://product-service
          predicates:
            - Path=/api/products/**
          filters:
            # Modify request path
            - RewritePath=/api/(?<segment>.*), /$\{segment}

            # Add request headers
            - AddRequestHeader=X-Request-Source, api-gateway

            # Add response headers
            - AddResponseHeader=X-Response-Time, $\{responseTime}

            # Add query parameters
            - AddRequestParameter=source, gateway

            # Set status
            - SetStatus=200
```

```
            # Request rate limiter
            - name: RequestRateLimiter
              args:
                redis-rate-limiter.replenishRate: 10
                redis-rate-limiter.burstCapacity: 20
                key-resolver: '#{@userKeyResolver}'

            # Request size limiter
            - name: RequestSize
              args:
                maxSize: 5MB

            # Circuit breaker
            - name: CircuitBreaker
              args:
                name: productServiceCircuitBreaker
                fallbackUri: forward:/fallback/products

            # Retry filter
            - name: Retry
              args:
                retries: 3
                statuses: BAD_GATEWAY
                methods: GET
                backoff:
                  firstBackoff: 50ms
                  maxBackoff: 500ms
                  factor: 2
                  basedOnPreviousValue: false
```

Custom Gateway Filters

Creating custom filters for specific requirements:

```
@Component
public class LoggingFilter implements GlobalFilter, Ordered {

    private static final Logger log =
LoggerFactory.getLogger(LoggingFilter.class);

    @Override
    public Mono<Void> filter(ServerWebExchange exchange, GatewayFilterChain chain)
{
        // Get request path
        String path = exchange.getRequest().getPath().toString();

        // Log incoming request
        log.info("Incoming request: {} {}",
                exchange.getRequest().getMethod(),
                path);
```

```
        // Record start time
        long startTime = System.currentTimeMillis();

        // Continue filter chain
        return chain.filter(exchange)
                .then(Mono.fromRunnable(() -> {
                    // Calculate request processing time
                    long duration = System.currentTimeMillis() - startTime;

                    // Log response details
                    log.info("Response for {} {} - Status: {}, Duration: {}ms",
                            exchange.getRequest().getMethod(),
                            path,
                            exchange.getResponse().getStatusCode(),
                            duration);
                }));
    }

    @Override
    public int getOrder() {
        // Set filter order (lower values have higher priority)
        return Ordered.LOWEST_PRECEDENCE;
    }
}

@Configuration
public class CustomFilterConfig {

    @Bean
    public RouteLocator routes(RouteLocatorBuilder builder) {
        return builder.routes()
                .route("custom_filter_route", r -> r
                        .path("/api/products/**")
                        .filters(f -> f
                                .rewritePath("/api/(?<segment>.*)", "/${segment}")
                                .filter(new AuthorizationFilter())  // Custom
filter
                                .filter(new RateLimitingFilter())   // Custom
filter
                        )
                        .uri("lb://product-service")
                )
                .build();
    }

    // Custom filter to check authorization
    public class AuthorizationFilter implements GatewayFilter, Ordered {

        @Override
        public Mono<Void> filter(ServerWebExchange exchange, GatewayFilterChain
chain) {
            // Get authorization header
            String authHeader =
```

```
exchange.getRequest().getHeaders().getFirst("Authorization");

            // Check if authorization is required for this path
            if (isSecuredPath(exchange.getRequest().getPath().toString()) &&
                    (authHeader == null || !authHeader.startsWith("Bearer "))) {

                // Return 401 Unauthorized
                exchange.getResponse().setStatusCode(HttpStatus.UNAUTHORIZED);
                return exchange.getResponse().setComplete();
            }

            // Continue filter chain
            return chain.filter(exchange);
        }

        private boolean isSecuredPath(String path) {
            // Define paths that require authorization
            return path.startsWith("/api/products/admin") ||
                   path.startsWith("/api/users");
        }

        @Override
        public int getOrder() {
            return Ordered.HIGHEST_PRECEDENCE;  // Execute first
        }
    }

    // Custom filter for rate limiting
    public class RateLimitingFilter implements GatewayFilter, Ordered {

        private final Map<String, TokenBucket> buckets = new ConcurrentHashMap<>
();

        @Override
        public Mono<Void> filter(ServerWebExchange exchange, GatewayFilterChain
chain) {
            // Get client IP
            String clientIp =
exchange.getRequest().getRemoteAddress().getAddress().getHostAddress();

            // Get or create token bucket for this client
            TokenBucket bucket = buckets.computeIfAbsent(clientIp,
                    ip -> new TokenBucket(10, 10, System.currentTimeMillis()));

            // Try to consume a token
            synchronized (bucket) {
                if (bucket.tryConsume()) {
                    // Allow request
                    return chain.filter(exchange);
                } else {
                    // Return 429 Too Many Requests

exchange.getResponse().setStatusCode(HttpStatus.TOO_MANY_REQUESTS);
                    return exchange.getResponse().setComplete();
```

```
                }
            }
        }

        @Override
        public int getOrder() {
            return Ordered.HIGHEST_PRECEDENCE + 1;  // Execute after
AuthorizationFilter
        }

        // Simple token bucket algorithm
        private static class TokenBucket {
            private final int capacity;
            private final double refillRate;  // tokens per second
            private double tokens;
            private long lastRefillTimestamp;

            public TokenBucket(int capacity, double refillRate, long
currentTimeMillis) {
                this.capacity = capacity;
                this.refillRate = refillRate;
                this.tokens = capacity;
                this.lastRefillTimestamp = currentTimeMillis;
            }

            public boolean tryConsume() {
                refill();

                if (tokens >= 1) {
                    tokens--;
                    return true;
                }

                return false;
            }

            private void refill() {
                long now = System.currentTimeMillis();
                double elapsedSeconds = (now - lastRefillTimestamp) / 1000.0;

                if (elapsedSeconds > 0) {
                    // Calculate tokens to add
                    double tokensToAdd = elapsedSeconds * refillRate;

                    // Add tokens up to capacity
                    tokens = Math.min(capacity, tokens + tokensToAdd);

                    // Update last refill timestamp
                    lastRefillTimestamp = now;
                }
            }
        }
    }
}
```

Cross-Cutting Concerns in API Gateway

Implementing common cross-cutting concerns at the gateway level:

1. **Authentication and Authorization**:

```
@Configuration
public class GatewaySecurityConfig {

    @Bean
    public SecurityWebFilterChain securityWebFilterChain(ServerHttpSecurity http)
{
        return http
                .csrf().disable()
                .authorizeExchange()
                    .pathMatchers("/api/auth/**", "/actuator/**").permitAll()
                    .pathMatchers("/api/admin/**").hasRole("ADMIN")
                    .anyExchange().authenticated()
                .and()
                .oauth2ResourceServer()
                    .jwt()
                .and()
                .and()
                .build();
    }

    @Bean
    public ReactiveJwtDecoder jwtDecoder() {
        return ReactiveJwtDecoders.fromIssuerLocation("http://auth-
server:8080/auth/realms/microservices");
    }
}
```

2. **Request/Response Logging**:

```
@Component
public class LoggingGlobalFilter implements GlobalFilter, Ordered {

    private static final Logger log =
LoggerFactory.getLogger(LoggingGlobalFilter.class);

    @Override
    public Mono<Void> filter(ServerWebExchange exchange, GatewayFilterChain chain)
{
        ServerHttpRequest request = exchange.getRequest();

        // Log request details
        log.info("Request: [{}] {} from {}",
```

```
                request.getMethod(),
                request.getURI(),
                request.getRemoteAddress());

        // Log request headers
        request.getHeaders().forEach((name, values) ->
                log.debug("Request Header: {} = {}", name, values));

        // Get start time
        long startTime = System.currentTimeMillis();

        return chain.filter(exchange)
                .then(Mono.fromRunnable(() -> {
                    // Log response details
                    ServerHttpResponse response = exchange.getResponse();

                    log.info("Response: {} for [{}] {} - Duration: {}ms",
                            response.getStatusCode(),
                            request.getMethod(),
                            request.getURI(),
                            System.currentTimeMillis() - startTime);

                    // Log response headers
                    response.getHeaders().forEach((name, values) ->
                            log.debug("Response Header: {} = {}", name, values));
                }));
    }

    @Override
    public int getOrder() {
        // Set highest priority
        return Ordered.HIGHEST_PRECEDENCE;
    }
}
```

3. **CORS Configuration**:

```
@Configuration
public class CorsGatewayConfig {

    @Bean
    public CorsWebFilter corsWebFilter() {
        CorsConfiguration corsConfig = new CorsConfiguration();
        corsConfig.setAllowedOrigins(List.of("https://example.com",
"https://dev.example.com"));
        corsConfig.setAllowedMethods(List.of("GET", "POST", "PUT", "DELETE",
"OPTIONS"));
        corsConfig.setAllowedHeaders(List.of("*"));
        corsConfig.setMaxAge(3600L);
        corsConfig.setAllowCredentials(true);

        UrlBasedCorsConfigurationSource source = new
```

```
UrlBasedCorsConfigurationSource();
        source.registerCorsConfiguration("/**", corsConfig);

        return new CorsWebFilter(source);
    }
}
```

4. **Request Transformation**:

```
@Component
public class RequestTransformationFilter implements GlobalFilter, Ordered {

    @Override
    public Mono<Void> filter(ServerWebExchange exchange, GatewayFilterChain chain)
{
        ServerHttpRequest request = exchange.getRequest();

        // Add common headers
        ServerHttpRequest modifiedRequest = request.mutate()
                .header("X-Trace-Id", UUID.randomUUID().toString())
                .header("X-Forwarded-For",
request.getRemoteAddress().getAddress().getHostAddress())
                .build();

        // Create new exchange with modified request
        ServerWebExchange modifiedExchange = exchange.mutate()
                .request(modifiedRequest)
                .build();

        return chain.filter(modifiedExchange);
    }

    @Override
    public int getOrder() {
        return Ordered.HIGHEST_PRECEDENCE + 1;
    }
}
```

Gateway Resilience Patterns

Implementing resilience patterns in API Gateway:

1. **Circuit Breaker**:

```
# application.yml
spring:
  cloud:
    gateway:
      routes:
```

```
        - id: product-service
          uri: lb://product-service
          predicates:
            - Path=/api/products/**
          filters:
            - name: CircuitBreaker
              args:
                name: productServiceCircuitBreaker
                fallbackUri: forward:/fallback/products
```

```
@RestController
public class FallbackController {

    @GetMapping("/fallback/products")
    public ResponseEntity<Map<String, String>> productServiceFallback() {
        Map<String, String> response = new HashMap<>();
        response.put("status", "error");
        response.put("message", "Product service is currently unavailable. Please
try again later.");
        return
ResponseEntity.status(HttpStatus.SERVICE_UNAVAILABLE).body(response);
    }

    @GetMapping("/fallback/orders")
    public ResponseEntity<Map<String, String>> orderServiceFallback() {
        Map<String, String> response = new HashMap<>();
        response.put("status", "error");
        response.put("message", "Order service is currently unavailable. Please
try again later.");
        return
ResponseEntity.status(HttpStatus.SERVICE_UNAVAILABLE).body(response);
    }
}
```

2. **Retry Mechanism**:

```
# application.yml
spring:
  cloud:
    gateway:
      routes:
        - id: product-service
          uri: lb://product-service
          predicates:
            - Path=/api/products/**
          filters:
            - name: Retry
              args:
                retries: 3
```

```
                statuses: BAD_GATEWAY,SERVICE_UNAVAILABLE
                methods: GET,POST
                backoff:
                  firstBackoff: 50ms
                  maxBackoff: 500ms
                  factor: 2
                  basedOnPreviousValue: false
```

3. **Rate Limiting**:

```
# application.yml
spring:
  cloud:
    gateway:
      routes:
        - id: product-service
          uri: lb://product-service
          predicates:
            - Path=/api/products/**
          filters:
            - name: RequestRateLimiter
              args:
                redis-rate-limiter.replenishRate: 10
                redis-rate-limiter.burstCapacity: 20
                key-resolver: '#{@ipKeyResolver}'
```

```
@Configuration
public class RateLimiterConfig {

    @Bean
    public RedisRateLimiter redisRateLimiter() {
        return new RedisRateLimiter(10, 20);
    }

    @Bean
    public KeyResolver ipKeyResolver() {
        return exchange -> Mono.just(
                Objects.requireNonNull(exchange.getRequest().getRemoteAddress())
                        .getAddress().getHostAddress()
        );
    }

    @Bean
    public KeyResolver userKeyResolver() {
        return exchange -> {
            String userId = exchange.getRequest().getHeaders().getFirst("X-User-
ID");
            if (userId != null) {
                return Mono.just(userId);
```

```
            }
            // Default to IP address if no user ID
            return Mono.just(

Objects.requireNonNull(exchange.getRequest().getRemoteAddress())
                            .getAddress().getHostAddress()
            );
        };
    }
}
```

Circuit Breaker Patterns

Circuit breakers prevent cascading failures in distributed systems.

Implementing Resilience4j

Resilience4j is a lightweight fault tolerance library:

```
<!-- Resilience4j dependencies -->
<dependency>
    <groupId>io.github.resilience4j</groupId>
    <artifactId>resilience4j-spring-boot2</artifactId>
</dependency>
<dependency>
    <groupId>io.github.resilience4j</groupId>
    <artifactId>resilience4j-circuitbreaker</artifactId>
</dependency>
<dependency>
    <groupId>io.github.resilience4j</groupId>
    <artifactId>resilience4j-timelimiter</artifactId>
</dependency>
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-aop</artifactId>
</dependency>
```

Circuit Breaker Configuration

```
# application.yml
resilience4j:
  circuitbreaker:
    instances:
      productService:
        registerHealthIndicator: true
        slidingWindowSize: 10
        slidingWindowType: COUNT_BASED
        minimumNumberOfCalls: 5
```

```
        permittedNumberOfCallsInHalfOpenState: 3
        automaticTransitionFromOpenToHalfOpenEnabled: true
        waitDurationInOpenState: 5s
        failureRateThreshold: 50
        eventConsumerBufferSize: 10
  timelimiter:
    instances:
      productService:
        timeoutDuration: 2s
        cancelRunningFuture: true
  retry:
    instances:
      productService:
        maxAttempts: 3
        waitDuration: 1s
        enableExponentialBackoff: true
        exponentialBackoffMultiplier: 2
        retryExceptions:
          - org.springframework.web.client.HttpServerErrorException
          - java.io.IOException
  bulkhead:
    instances:
      productService:
        maxConcurrentCalls: 10
        maxWaitDuration: 10ms
  ratelimiter:
    instances:
      productService:
        limitForPeriod: 10
        limitRefreshPeriod: 1s
        timeoutDuration: 3s
```

Using Circuit Breaker Annotations

```
@Service
public class ProductService {

    private final RestTemplate restTemplate;

    // Constructor injection

    @CircuitBreaker(name = "productService", fallbackMethod =
"getProductFallback")
    @Retry(name = "productService", fallbackMethod = "getProductFallback")
    @Bulkhead(name = "productService", fallbackMethod = "getProductFallback")
    @TimeLimiter(name = "productService", fallbackMethod = "getProductFallback")
    public ProductDto getProduct(String productId) {
        return restTemplate.getForObject(
                "http://product-service/products/{id}",
                ProductDto.class,
                productId
```

```
        );
    }

    public ProductDto getProductFallback(String productId, Exception ex) {
        log.error("Fallback for product retrieval. Product ID: {}, Error: {}",
productId, ex.getMessage());
        return new ProductDto(productId, "Fallback Product", "Product temporarily
unavailable", BigDecimal.ZERO);
    }
}
```

Manual Circuit Breaker Usage

```
@Service
public class InventoryService {

    private final CircuitBreakerRegistry circuitBreakerRegistry;
    private final RetryRegistry retryRegistry;
    private final InventoryClient inventoryClient;

    // Constructor injection

    public boolean checkProductAvailability(String productId, int quantity) {
        // Get circuit breaker and retry instances
        CircuitBreaker circuitBreaker =
circuitBreakerRegistry.circuitBreaker("inventoryService");
        Retry retry = retryRegistry.retry("inventoryService");

        // Create supplier function that calls external service
        Supplier<Boolean> inventorySupplier = () ->
inventoryClient.checkAvailability(productId, quantity);

        // Apply retry and circuit breaker patterns
        Supplier<Boolean> retryableSupplier = Retry.decorateSupplier(retry,
inventorySupplier);
        Supplier<Boolean> resilientSupplier =
CircuitBreaker.decorateSupplier(circuitBreaker, retryableSupplier);

        try {
            // Execute call with resilience patterns
            return resilientSupplier.get();
        } catch (Exception e) {
            log.error("Error checking inventory: {}", e.getMessage());
            return false;  // Fallback value
        }
    }
}
```

Circuit Breaker States

Monitoring and managing circuit breaker states:

```
@RestController
@RequestMapping("/admin/circuit-breakers")
public class CircuitBreakerController {

    private final CircuitBreakerRegistry circuitBreakerRegistry;

    // Constructor injection

    @GetMapping
    public List<CircuitBreakerStatus> getAllCircuitBreakers() {
        return circuitBreakerRegistry.getAllCircuitBreakers().stream()
                .map(this::getCircuitBreakerStatus)
                .collect(Collectors.toList());
    }

    @GetMapping("/{name}")
    public CircuitBreakerStatus getCircuitBreaker(@PathVariable String name) {
        CircuitBreaker circuitBreaker =
circuitBreakerRegistry.circuitBreaker(name);
        return getCircuitBreakerStatus(circuitBreaker);
    }

    @PostMapping("/{name}/reset")
    public ResponseEntity<String> resetCircuitBreaker(@PathVariable String name) {
        CircuitBreaker circuitBreaker =
circuitBreakerRegistry.circuitBreaker(name);
        circuitBreaker.reset();
        return ResponseEntity.ok("Circuit breaker reset successfully");
    }

    @PostMapping("/{name}/state/{state}")
    public ResponseEntity<String> changeCircuitBreakerState(
            @PathVariable String name,
            @PathVariable String state) {

        CircuitBreaker circuitBreaker =
circuitBreakerRegistry.circuitBreaker(name);

        switch (state.toLowerCase()) {
            case "open":
                circuitBreaker.transitionToOpenState();
                break;
            case "halfopen":
                circuitBreaker.transitionToHalfOpenState();
                break;
            case "closed":
                circuitBreaker.transitionToClosedState();
                break;
            case "disabled":
                circuitBreaker.transitionToDisabledState();
                break;
```

```
            case "forcedopen":
                circuitBreaker.transitionToForcedOpenState();
                break;
            default:
                return ResponseEntity.badRequest().body("Invalid state: " +
state);
        }

        return ResponseEntity.ok("Circuit breaker state changed successfully");
    }

    private CircuitBreakerStatus getCircuitBreakerStatus(CircuitBreaker
circuitBreaker) {
        CircuitBreaker.Metrics metrics = circuitBreaker.getMetrics();
        CircuitBreakerStatus status = new CircuitBreakerStatus();

        status.setName(circuitBreaker.getName());
        status.setState(circuitBreaker.getState().name());
        status.setFailureRate(metrics.getFailureRate());
        status.setNumberOfSuccessfulCalls(metrics.getNumberOfSuccessfulCalls());
        status.setNumberOfFailedCalls(metrics.getNumberOfFailedCalls());

status.setNumberOfNotPermittedCalls(metrics.getNumberOfNotPermittedCalls());

        return status;
    }

    @Data
    public static class CircuitBreakerStatus {
        private String name;
        private String state;
        private float failureRate;
        private int numberOfSuccessfulCalls;
        private int numberOfFailedCalls;
        private int numberOfNotPermittedCalls;
    }
}
```

Circuit Breaker Events

Subscribing to circuit breaker events:

```
@Component
public class CircuitBreakerEventsListener {

    private static final Logger log =
LoggerFactory.getLogger(CircuitBreakerEventsListener.class);

    @EventListener
    public void onStateTransition(CircuitBreakerOnStateTransitionEvent event) {
        log.info("Circuit breaker '{}' state changed from {} to {}",
```

```
                event.getCircuitBreakerName(),
                event.getStateTransition().getFromState(),
                event.getStateTransition().getToState());
    }

    @EventListener
    public void onError(CircuitBreakerOnErrorEvent event) {
        log.error("Circuit breaker '{}' error: {}, duration: {} ms",
                event.getCircuitBreakerName(),
                event.getThrowable().getMessage(),
                event.getElapsedDuration().toMillis());
    }

    @EventListener
    public void onSuccess(CircuitBreakerOnSuccessEvent event) {
        log.debug("Circuit breaker '{}' success, duration: {} ms",
                event.getCircuitBreakerName(),
                event.getElapsedDuration().toMillis());
    }

    @EventListener
    public void onIgnoredError(CircuitBreakerOnIgnoredErrorEvent event) {
        log.warn("Circuit breaker '{}' ignored error: {}",
                event.getCircuitBreakerName(),
                event.getThrowable().getMessage());
    }
}
```

Distributed Configuration

Centralized configuration management for microservices.

Implementing Spring Cloud Config Server

```
<!-- Config Server dependency -->
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-config-server</artifactId>
</dependency>
```

```
@SpringBootApplication
@EnableConfigServer
public class ConfigServerApplication {

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

```
# application.properties
server.port=8888
spring.application.name=config-server

# Git backend configuration
spring.cloud.config.server.git.uri=https://github.com/company/config-repo
spring.cloud.config.server.git.search-paths=config-files
spring.cloud.config.server.git.clone-on-start=true
spring.cloud.config.server.git.default-label=main

# Security (optional)
spring.security.user.name=config
spring.security.user.password=password
```

Config Server Backends

1. **Git Backend**:

```
# application.properties
spring.cloud.config.server.git.uri=https://github.com/company/config-repo
spring.cloud.config.server.git.search-paths=config-files
spring.cloud.config.server.git.username=username
spring.cloud.config.server.git.password=password
```

2. **File System Backend**:

```
# application.properties
spring.profiles.active=native
spring.cloud.config.server.native.search-locations=file:///path/to/config-files
```

3. **Vault Backend**:

```
# application.properties
spring.profiles.active=vault
spring.cloud.config.server.vault.host=localhost
spring.cloud.config.server.vault.port=8200
spring.cloud.config.server.vault.token=your-token
```

4. **JDBC Backend**:

```
# application.properties
spring.profiles.active=jdbc
spring.datasource.url=jdbc:mysql://localhost:3306/config
```

```
spring.datasource.username=root
spring.datasource.password=password
```

Config Client Implementation

```
<!-- Config Client dependency -->
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-config</artifactId>
</dependency>
```

```
# bootstrap.properties
spring.application.name=product-service
spring.profiles.active=dev
spring.cloud.config.uri=http://localhost:8888
spring.cloud.config.username=config
spring.cloud.config.password=password
spring.cloud.config.fail-fast=true
spring.cloud.config.retry.max-attempts=6
spring.cloud.config.retry.initial-interval=1000
spring.cloud.config.retry.max-interval=2000
spring.cloud.config.retry.multiplier=1.1
```

Configuration Encryption

Securing sensitive configuration data:

```
# Config Server with encryption key
encrypt.key=your-strong-encryption-key

# Alternatively, use a keystore
encrypt.keyStore.location=classpath:keystore.jks
encrypt.keyStore.password=keystorepass
encrypt.keyStore.alias=configkey
encrypt.keyStore.secret=keysecret
```

Client-side usage of encrypted values:

```
# Property in Git repo (encrypted)
database.password={cipher}AQA...encrypted-value...

# Will be decrypted by Config Server
```

Dynamic Configuration Refresh

Refreshing configuration without restarting:

```
<!-- Actuator dependency for refresh endpoint -->
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-actuator</artifactId>
</dependency>
```

```
# application.properties
management.endpoints.web.exposure.include=refresh,health,info
```

```
@RefreshScope
@RestController
@RequestMapping("/api/products")
public class ProductController {

    @Value("${product.feature.enabled:false}")
    private boolean featureEnabled;

    @Value("${product.max-results:100}")
    private int maxResults;

    @GetMapping("/settings")
    public Map<String, Object> getSettings() {
        return Map.of(
                "featureEnabled", featureEnabled,
                "maxResults", maxResults
        );
    }
}
```

Triggering configuration refresh:

```
# Manual refresh via actuator endpoint
curl -X POST http://localhost:8080/actuator/refresh
```

Spring Cloud Bus

Automatically refreshing all instances:

```
<!-- Spring Cloud Bus with RabbitMQ -->
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-bus-amqp</artifactId>
</dependency>
```

```
# application.properties
spring.rabbitmq.host=localhost
spring.rabbitmq.port=5672
spring.rabbitmq.username=guest
spring.rabbitmq.password=guest

management.endpoints.web.exposure.include=refresh,bus-refresh,health,info
```

Triggering refresh for all instances:

```
# Refresh all instances via bus-refresh endpoint
curl -X POST http://localhost:8080/actuator/bus-refresh
```

Distributed Tracing

Tracking requests across microservices for observability.

Implementing Spring Cloud Sleuth and Zipkin

```
<!-- Sleuth and Zipkin dependencies -->
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-sleuth</artifactId>
</dependency>
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-sleuth-zipkin</artifactId>
</dependency>
```

```
# application.properties
spring.application.name=product-service
spring.sleuth.sampler.probability=1.0
spring.zipkin.base-url=http://localhost:9411
```

Trace Context Propagation

```
@Service
public class OrderService {

    private static final Logger log = LoggerFactory.getLogger(OrderService.class);

    private final RestTemplate restTemplate;
    private final OrderRepository orderRepository;

    // Constructor injection

    public OrderResponse processOrder(OrderRequest request) {
        log.info("Processing order for customer: {}", request.getCustomerId());

        // Create order
        Order order = createOrder(request);

        // Call product service to check inventory
        ProductResponse product = restTemplate.getForObject(
                "http://product-service/products/{id}/inventory?quantity=
{quantity}",
                ProductResponse.class,
                request.getProductId(),
                request.getQuantity()
        );

        log.info("Inventory check response: {}", product.isAvailable());

        if (!product.isAvailable()) {
            throw new InsufficientInventoryException("Not enough inventory for
product: " + request.getProductId());
        }

        // Call payment service to process payment
        PaymentRequest paymentRequest = new PaymentRequest(
                order.getId(),
                request.getCustomerId(),
                order.getTotalAmount()
        );

        PaymentResponse payment = restTemplate.postForObject(
                "http://payment-service/payments",
                paymentRequest,
                PaymentResponse.class
        );

        log.info("Payment processed with status: {}", payment.getStatus());

        // Update order status
        order.setStatus(OrderStatus.PAID);
        order = orderRepository.save(order);

        // Return response
        return new OrderResponse(
```

```
                order.getId(),
                order.getStatus().name(),
                order.getTotalAmount()
        );
    }

    private Order createOrder(OrderRequest request) {
        Order order = new Order();
        order.setCustomerId(request.getCustomerId());
        order.setProductId(request.getProductId());
        order.setQuantity(request.getQuantity());
        order.setStatus(OrderStatus.CREATED);
        order.setCreatedAt(LocalDateTime.now());

        // Calculate total amount by calling product service
        ProductResponse product = restTemplate.getForObject(
                "http://product-service/products/{id}",
                ProductResponse.class,
                request.getProductId()
        );

        order.setTotalAmount(product.getPrice().multiply(new
BigDecimal(request.getQuantity())));

        return orderRepository.save(order);
    }
}
```

Adding Custom Spans

```
@Service
public class InventoryService {

    private static final Logger log =
LoggerFactory.getLogger(InventoryService.class);

    private final Tracer tracer;
    private final InventoryRepository inventoryRepository;

    // Constructor injection

    public boolean reserveInventory(String productId, int quantity) {
        Span span = tracer.startScopedSpan("reserveInventory");
        try {
            span.tag("productId", productId);
            span.tag("quantity", String.valueOf(quantity));

            log.info("Reserving inventory for product: {}, quantity: {}",
productId, quantity);

            // First check if there's enough inventory
```

```
            Span checkSpan =
tracer.currentTracer().startScopedSpan("checkInventory");
            boolean hasInventory = false;
            try {
                hasInventory = checkInventory(productId, quantity);
                checkSpan.tag("hasInventory", String.valueOf(hasInventory));
            } finally {
                checkSpan.finish();
            }

            if (!hasInventory) {
                log.warn("Insufficient inventory for product: {}", productId);
                return false;
            }

            // Update inventory
            Span updateSpan =
tracer.currentTracer().startScopedSpan("updateInventory");
            try {
                updateInventory(productId, quantity);
            } finally {
                updateSpan.finish();
            }

            log.info("Inventory reserved successfully for product: {}",
productId);
            return true;
        } catch (Exception e) {
            span.tag("error", e.getMessage());
            log.error("Error reserving inventory: {}", e.getMessage());
            return false;
        } finally {
            span.finish();
        }
    }

    private boolean checkInventory(String productId, int quantity) {
        Inventory inventory = inventoryRepository.findByProductId(productId)
                .orElseThrow(() -> new ResourceNotFoundException("Inventory not
found for product: " + productId));

        return inventory.getQuantity() >= quantity;
    }

    private void updateInventory(String productId, int quantity) {
        Inventory inventory = inventoryRepository.findByProductId(productId)
                .orElseThrow(() -> new ResourceNotFoundException("Inventory not
found for product: " + productId));

        inventory.setQuantity(inventory.getQuantity() - quantity);
        inventoryRepository.save(inventory);
    }
}
```

Tracing Asynchronous Operations

```
@Service
public class AsyncNotificationService {

    private static final Logger log =
LoggerFactory.getLogger(AsyncNotificationService.class);

    private final Tracer tracer;
    private final EmailService emailService;

    // Constructor injection

    @Async
    public CompletableFuture<Boolean> sendOrderConfirmation(Order order) {
        // Extract current trace context
        Span currentSpan = tracer.currentSpan();

        return CompletableFuture.supplyAsync(() -> {
            // Create new child span continuing from parent
            Span span = tracer.toSpan(tracer.createSpan("sendOrderConfirmation",
currentSpan.context()));

            try (SpanInScope ws = tracer.withSpan(span.start())) {
                span.tag("orderId", String.valueOf(order.getId()));

                log.info("Sending order confirmation email for order: {}",
order.getId());

                // Get customer details
                Customer customer = getCustomerDetails(order.getCustomerId());

                // Send email
                boolean success = emailService.sendEmail(
                        customer.getEmail(),
                        "Order Confirmation - " + order.getId(),
                        createOrderConfirmationText(order)
                );

                span.tag("success", String.valueOf(success));

                if (success) {
                    log.info("Order confirmation email sent successfully");
                } else {
                    log.warn("Failed to send order confirmation email");
                }

                return success;
            } catch (Exception e) {
                span.tag("error", e.getMessage());
                log.error("Error sending confirmation email: {}", e.getMessage());
```

```
                return false;
            } finally {
                span.finish();
            }
        });
    }

    private Customer getCustomerDetails(String customerId) {
        // Implementation to get customer details
        return null;
    }

    private String createOrderConfirmationText(Order order) {
        // Implementation to create email text
        return null;
    }
}
```

Custom Trace Exporters

```
@Configuration
public class TracingConfig {

    @Bean
    public SpanExporter customSpanExporter() {
        return new CustomSpanExporter();
    }

    @Bean
    public SpanHandler customSpanHandler() {
        return new CustomSpanHandler();
    }

    public static class CustomSpanExporter implements SpanExporter {

        private static final Logger log =
LoggerFactory.getLogger(CustomSpanExporter.class);

        @Override
        public void exportSpan(Span span) {
            if (log.isDebugEnabled()) {
                log.debug("Exporting span: {}, duration: {} ms",
                        span.name(), span.duration().toMillis());

                span.tags().forEach((key, value) ->
                        log.debug("  Tag: {} = {}", key, value));
            }

            // Additional custom export logic
            // - Send to custom monitoring system
            // - Store in database
```

```
            // - Publish to messaging system
        }
    }

    public static class CustomSpanHandler extends SpanHandler {

        @Override
        public Span handleReceive(Map<String, String> extractedContext) {
            // Custom logic when a span is received from upstream
            return super.handleReceive(extractedContext);
        }

        @Override
        public Span handleCreate(ScopedSpan scopedSpan) {
            // Custom logic when a span is created
            return super.handleCreate(scopedSpan);
        }

        @Override
        public Span handleStart(Span span) {
            // Custom logic when a span is started
            return super.handleStart(span);
        }

        @Override
        public void handleFinish(Span span, Map<String, String> extractedContext)
{
            // Custom logic when a span is finished
            super.handleFinish(span, extractedContext);
        }
    }
}
```

Event-driven Architecture

Building loosely coupled microservices with event-driven communication.

Implementing Spring Cloud Stream

Spring Cloud Stream provides a unified programming model for message-driven applications:

```
<!-- Spring Cloud Stream dependencies -->
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-stream</artifactId>
</dependency>
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-stream-binder-rabbit</artifactId>
</dependency>
```

```
@SpringBootApplication
public class OrderServiceApplication {

    public static void main(String[] args) {
        SpringApplication.run(OrderServiceApplication.class, args);
    }

    @Bean
    public Function<OrderCreatedEvent, OrderProcessingEvent> processOrder() {
        return orderCreatedEvent -> {
            // Process the order event
            String orderId = orderCreatedEvent.getOrderId();
            log.info("Processing order: {}", orderId);

            // Return a processing event
            return new OrderProcessingEvent(
                    orderId,
                    "Order processing started",
                    LocalDateTime.now()
            );
        };
    }

    @Bean
    public Consumer<PaymentCompletedEvent> handlePaymentCompleted() {
        return paymentCompletedEvent -> {
            String orderId = paymentCompletedEvent.getOrderId();
            String paymentId = paymentCompletedEvent.getPaymentId();

            log.info("Payment completed for order: {}, payment: {}", orderId,
paymentId);

            // Update order status
            orderService.updateOrderStatus(orderId, OrderStatus.PAID);
        };
    }

    @Bean
    public Supplier<OrderStatisticsEvent> orderStatistics() {
        return () -> {
            // Generate periodic statistics
            Map<OrderStatus, Long> counts = orderService.getOrderCounts();
            BigDecimal totalRevenue = orderService.calculateTotalRevenue();

            return new OrderStatisticsEvent(
                    counts,
                    totalRevenue,
                    LocalDateTime.now()
            );
        };
    }
}
```

Spring Cloud Stream Configuration

```
# application.yml
spring:
  cloud:
    stream:
      function:
        definition: processOrder;handlePaymentCompleted;orderStatistics
      bindings:
        # Input binding for order processing
        processOrder-in-0:
          destination: order-events
          group: order-processing
          consumer:
            maxAttempts: 3
            backOffInitialInterval: 1000
            backOffMaxInterval: 10000
            backOffMultiplier: 2.0

        # Output binding for order processing
        processOrder-out-0:
          destination: order-processing-events

        # Input binding for payment events
        handlePaymentCompleted-in-0:
          destination: payment-events
          group: order-service

        # Output binding for statistics
        orderStatistics-out-0:
          destination: order-statistics
      rabbit:
        bindings:
          processOrder-in-0:
            consumer:
              autoBindDlq: true
              republishToDlq: true
          handlePaymentCompleted-in-0:
            consumer:
              autoBindDlq: true
              republishToDlq: true
```

Event Message Models

```
@Data
@NoArgsConstructor
@AllArgsConstructor
public class OrderCreatedEvent {
```

```
    private String orderId;
    private String customerId;
    private List<OrderItem> items;
    private BigDecimal totalAmount;
    private LocalDateTime createdAt;
}

@Data
@NoArgsConstructor
@AllArgsConstructor
public class OrderProcessingEvent {
    private String orderId;
    private String status;
    private LocalDateTime timestamp;
}

@Data
@NoArgsConstructor
@AllArgsConstructor
public class PaymentCompletedEvent {
    private String paymentId;
    private String orderId;
    private BigDecimal amount;
    private String status;
    private LocalDateTime timestamp;
}

@Data
@NoArgsConstructor
@AllArgsConstructor
public class OrderStatisticsEvent {
    private Map<OrderStatus, Long> orderCounts;
    private BigDecimal totalRevenue;
    private LocalDateTime timestamp;
}
```

Event Production

```
@Service
public class OrderEventProducer {

    private final StreamBridge streamBridge;

    // Constructor injection

    public void publishOrderCreatedEvent(Order order) {
        OrderCreatedEvent event = mapToOrderCreatedEvent(order);

        boolean sent = streamBridge.send("order-events", event);

        if (sent) {
```

```
            log.info("Order created event published for order: {}",
order.getId());
        } else {
            log.error("Failed to publish order created event for order: {}",
order.getId());
        }
    }

    public void publishOrderStatusChangedEvent(Order order, OrderStatus
previousStatus) {
        OrderStatusChangedEvent event = new OrderStatusChangedEvent(
                order.getId(),
                previousStatus.name(),
                order.getStatus().name(),
                order.getUpdatedAt()
        );

        streamBridge.send("order-status-events", event);
    }

    private OrderCreatedEvent mapToOrderCreatedEvent(Order order) {
        // Map order to event
        List<OrderItem> items = order.getItems().stream()
                .map(item -> new OrderItem(
                        item.getProductId(),
                        item.getQuantity(),
                        item.getPrice()
                ))
                .collect(Collectors.toList());

        return new OrderCreatedEvent(
                order.getId(),
                order.getCustomerId(),
                items,
                order.getTotalAmount(),
                order.getCreatedAt()
        );
    }
}
```

Event Consumption

```
@Service
public class PaymentEventConsumer {

    private static final Logger log =
LoggerFactory.getLogger(PaymentEventConsumer.class);

    private final OrderService orderService;

    // Constructor injection
```

```
    @Bean
    public Consumer<PaymentCompletedEvent> processPaymentCompleted() {
        return event -> {
            log.info("Received payment completed event - Order: {}, Payment: {}",
                    event.getOrderId(), event.getPaymentId());

            if ("COMPLETED".equals(event.getStatus())) {
                try {
                    orderService.confirmPayment(event.getOrderId(),
event.getPaymentId());
                    log.info("Order payment confirmed for order: {}",
event.getOrderId());
                } catch (Exception e) {
                    log.error("Error confirming order payment: {}",
e.getMessage());
                    throw e;  // Retry handling will be triggered
                }
            } else if ("FAILED".equals(event.getStatus())) {
                orderService.handleFailedPayment(event.getOrderId(),
event.getPaymentId());
                log.warn("Order payment failed for order: {}",
event.getOrderId());
            }
        };
    }
}
```

Error Handling and DLQ

```
@Configuration
public class StreamErrorHandlingConfig {

    @Bean
    public ListenerContainerCustomizer<AbstractMessageListenerContainer>
customizer() {
        return (container, destinationName, group) -> {
            container.setErrorHandler(new DefaultErrorHandler(
                    // Recoverer to handle messages after retries are exhausted
                    new DeadLetterPublishingRecoverer(),
                    // Exponential backoff for retries
                    new ExponentialBackOffWithMaxRetries(3)
                            .setInitialInterval(1000)
                            .setMultiplier(2.0)
                            .setMaxInterval(10000)
            ));
        };
    }

    @Bean
    public DLQHandler dlqHandler() {
```

```
        return new DLQHandler();
    }

    public static class DLQHandler {

        private static final Logger log =
LoggerFactory.getLogger(DLQHandler.class);

        @ServiceActivator(inputChannel = "order-events.order-processing.errors")
        public void handleOrderEventDlq(Message<?> message) {
            log.error("Handling dead letter: {}", message.getPayload());

            // Access original headers
            MessageHeaders headers = message.getHeaders();
            log.info("Original topic: {}", headers.get("x-original-topic"));
            log.info("Exception message: {}", headers.get("x-exception-message"));

            // Additional handling for dead-lettered messages
            // - Store in database for manual processing
            // - Notify operations team
            // - Attempt alternate processing strategy
        }
    }
}
```

Event Sourcing Pattern

```
@Entity
@Table(name = "orders")
public class Order {

    @Id
    private String id;

    @Column(nullable = false)
    private String customerId;

    @Column(nullable = false)
    @Enumerated(EnumType.STRING)
    private OrderStatus status;

    @Column(nullable = false, precision = 10, scale = 2)
    private BigDecimal totalAmount;

    @Column(nullable = false)
    private LocalDateTime createdAt;

    @Column
    private LocalDateTime updatedAt;

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

```
    @JoinColumn(name = "order_id")
    private List<OrderItem> items = new ArrayList<>();

    @Version
    private Long version;

    // Getters and setters
}

@Entity
@Table(name = "order_events")
public class OrderEvent {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    @Column(nullable = false)
    private String orderId;

    @Column(nullable = false)
    @Enumerated(EnumType.STRING)
    private OrderEventType eventType;

    @Column(nullable = false)
    private String eventData;  // JSON data

    @Column(nullable = false)
    private LocalDateTime timestamp;

    @Column(nullable = false)
    private Long version;

    // Getters and setters
}

public enum OrderEventType {
    ORDER_CREATED,
    ORDER_ITEM_ADDED,
    ORDER_ITEM_REMOVED,
    ORDER_PAYMENT_CONFIRMED,
    ORDER_CANCELLED,
    ORDER_SHIPPED,
    ORDER_DELIVERED
}

@Service
public class OrderEventSourceService {

    private final OrderEventRepository eventRepository;
    private final ObjectMapper objectMapper;

    // Constructor injection
```

```
    @Transactional
    public void createOrder(OrderCreatedEvent event) throws
JsonProcessingException {
        // Create order event
        OrderEvent orderEvent = new OrderEvent();
        orderEvent.setOrderId(event.getOrderId());
        orderEvent.setEventType(OrderEventType.ORDER_CREATED);
        orderEvent.setEventData(objectMapper.writeValueAsString(event));
        orderEvent.setTimestamp(LocalDateTime.now());
        orderEvent.setVersion(1L);

        // Save event
        eventRepository.save(orderEvent);
    }

    @Transactional
    public void addOrderItem(String orderId, OrderItem item) throws
JsonProcessingException {
        // Get current order version
        Long currentVersion = eventRepository.findMaxVersionByOrderId(orderId);

        // Create order event
        OrderEvent orderEvent = new OrderEvent();
        orderEvent.setOrderId(orderId);
        orderEvent.setEventType(OrderEventType.ORDER_ITEM_ADDED);
        orderEvent.setEventData(objectMapper.writeValueAsString(item));
        orderEvent.setTimestamp(LocalDateTime.now());
        orderEvent.setVersion(currentVersion + 1);

        // Save event
        eventRepository.save(orderEvent);
    }

    @Transactional(readOnly = true)
    public Order reconstructOrder(String orderId) throws JsonProcessingException {
        // Get all events for this order sorted by version
        List<OrderEvent> events =
eventRepository.findByOrderIdOrderByVersion(orderId);

        if (events.isEmpty()) {
            throw new ResourceNotFoundException("Order not found: " + orderId);
        }

        // Initialize order
        Order order = null;

        // Apply events to rebuild the order state
        for (OrderEvent event : events) {
            switch (event.getEventType()) {
                case ORDER_CREATED:
                    OrderCreatedEvent createdEvent = objectMapper.readValue(
                            event.getEventData(), OrderCreatedEvent.class);
                    order = createOrderFromEvent(createdEvent);
                    break;
```

```
                case ORDER_ITEM_ADDED:
                    OrderItem addedItem = objectMapper.readValue(
                            event.getEventData(), OrderItem.class);
                    order.getItems().add(addedItem);
                    recalculateTotal(order);
                    break;

                case ORDER_ITEM_REMOVED:
                    OrderItem removedItem = objectMapper.readValue(
                            event.getEventData(), OrderItem.class);
                    order.getItems().removeIf(item ->

item.getProductId().equals(removedItem.getProductId()));
                    recalculateTotal(order);
                    break;

                case ORDER_PAYMENT_CONFIRMED:
                    order.setStatus(OrderStatus.PAID);
                    order.setUpdatedAt(event.getTimestamp());
                    break;

                case ORDER_CANCELLED:
                    order.setStatus(OrderStatus.CANCELLED);
                    order.setUpdatedAt(event.getTimestamp());
                    break;

                case ORDER_SHIPPED:
                    order.setStatus(OrderStatus.SHIPPED);
                    order.setUpdatedAt(event.getTimestamp());
                    break;

                case ORDER_DELIVERED:
                    order.setStatus(OrderStatus.DELIVERED);
                    order.setUpdatedAt(event.getTimestamp());
                    break;
            }

            // Update version
            order.setVersion(event.getVersion());
        }

        return order;
    }

    private Order createOrderFromEvent(OrderCreatedEvent event) {
        Order order = new Order();
        order.setId(event.getOrderId());
        order.setCustomerId(event.getCustomerId());
        order.setStatus(OrderStatus.CREATED);
        order.setTotalAmount(event.getTotalAmount());
        order.setCreatedAt(event.getCreatedAt());

        // Add items
```

```
        for (OrderItem item : event.getItems()) {
            order.getItems().add(item);
        }

        return order;
    }

    private void recalculateTotal(Order order) {
        BigDecimal total = order.getItems().stream()
                .map(item -> item.getPrice().multiply(new
BigDecimal(item.getQuantity())))
                .reduce(BigDecimal.ZERO, BigDecimal::add);

        order.setTotalAmount(total);
    }
}
```

CQRS Pattern

Command Query Responsibility Segregation pattern separates read and write operations:

```
// Command models
@Data
@NoArgsConstructor
@AllArgsConstructor
public class CreateOrderCommand {
    private String customerId;
    private List<OrderItemDto> items;
    private String shippingAddress;
}

@Data
@NoArgsConstructor
@AllArgsConstructor
public class AddOrderItemCommand {
    private String orderId;
    private String productId;
    private int quantity;
    private BigDecimal price;
}

@Data
@NoArgsConstructor
@AllArgsConstructor
public class CancelOrderCommand {
    private String orderId;
    private String reason;
}

// Query models
@Data
```

```
@NoArgsConstructor
@AllArgsConstructor
public class OrderSummary {
    private String orderId;
    private String customerId;
    private OrderStatus status;
    private BigDecimal totalAmount;
    private LocalDateTime createdAt;
    private int itemCount;
}

@Data
@NoArgsConstructor
@AllArgsConstructor
public class OrderDetail {
    private String orderId;
    private String customerId;
    private OrderStatus status;
    private BigDecimal totalAmount;
    private LocalDateTime createdAt;
    private LocalDateTime updatedAt;
    private List<OrderItemDto> items;
    private String shippingAddress;
    private PaymentInfo payment;
}

// Command service
@Service
public class OrderCommandService {

    private final OrderRepository orderRepository;
    private final OrderEventProducer eventProducer;

    // Constructor injection

    @Transactional
    public String createOrder(CreateOrderCommand command) {
        // Generate new order ID
        String orderId = UUID.randomUUID().toString();

        // Create order entity
        Order order = new Order();
        order.setId(orderId);
        order.setCustomerId(command.getCustomerId());
        order.setShippingAddress(command.getShippingAddress());
        order.setStatus(OrderStatus.CREATED);
        order.setCreatedAt(LocalDateTime.now());

        // Add items
        for (OrderItemDto itemDto : command.getItems()) {
            OrderItem item = new OrderItem();
            item.setProductId(itemDto.getProductId());
            item.setQuantity(itemDto.getQuantity());
            item.setPrice(itemDto.getPrice());
```

```
            order.getItems().add(item);
        }

        // Calculate total
        BigDecimal total = order.getItems().stream()
                .map(item -> item.getPrice().multiply(new
BigDecimal(item.getQuantity())))
                .reduce(BigDecimal.ZERO, BigDecimal::add);

        order.setTotalAmount(total);

        // Save order
        orderRepository.save(order);

        // Publish event
        eventProducer.publishOrderCreatedEvent(order);

        return orderId;
    }

    @Transactional
    public void addOrderItem(AddOrderItemCommand command) {
        // Find order
        Order order = orderRepository.findById(command.getOrderId())
                .orElseThrow(() -> new ResourceNotFoundException("Order not found:
" + command.getOrderId()));

        // Check if order can be modified
        if (order.getStatus() != OrderStatus.CREATED) {
            throw new OrderStatusException("Cannot modify order in status: " +
order.getStatus());
        }

        // Create new item
        OrderItem newItem = new OrderItem();
        newItem.setProductId(command.getProductId());
        newItem.setQuantity(command.getQuantity());
        newItem.setPrice(command.getPrice());

        // Add item to order
        order.getItems().add(newItem);

        // Recalculate total
        BigDecimal total = order.getItems().stream()
                .map(item -> item.getPrice().multiply(new
BigDecimal(item.getQuantity())))
                .reduce(BigDecimal.ZERO, BigDecimal::add);

        order.setTotalAmount(total);
        order.setUpdatedAt(LocalDateTime.now());

        // Save order
        orderRepository.save(order);
```

```
        // Publish event
        eventProducer.publishOrderItemAddedEvent(order, newItem);
    }

    @Transactional
    public void cancelOrder(CancelOrderCommand command) {
        // Find order
        Order order = orderRepository.findById(command.getOrderId())
                .orElseThrow(() -> new ResourceNotFoundException("Order not found:
" + command.getOrderId()));

        // Check if order can be cancelled
        if (order.getStatus() == OrderStatus.DELIVERED || order.getStatus() ==
OrderStatus.CANCELLED) {
            throw new OrderStatusException("Cannot cancel order in status: " +
order.getStatus());
        }

        // Update order status
        OrderStatus previousStatus = order.getStatus();
        order.setStatus(OrderStatus.CANCELLED);
        order.setCancellationReason(command.getReason());
        order.setUpdatedAt(LocalDateTime.now());

        // Save order
        orderRepository.save(order);

        // Publish event
        eventProducer.publishOrderStatusChangedEvent(order, previousStatus);
    }
}

// Query service
@Service
public class OrderQueryService {

    private final OrderRepository orderRepository;
    private final OrderSummaryRepository orderSummaryRepository;

    // Constructor injection

    @Transactional(readOnly = true)
    public Page<OrderSummary> findOrdersByCustomerId(String customerId, Pageable
pageable) {
        return orderSummaryRepository.findByCustomerId(customerId, pageable);
    }

    @Transactional(readOnly = true)
    public OrderDetail getOrderDetail(String orderId) {
        Order order = orderRepository.findById(orderId)
                .orElseThrow(() -> new ResourceNotFoundException("Order not found:
" + orderId));

        // Map order to detail DTO
```

```
        OrderDetail detail = new OrderDetail();
        detail.setOrderId(order.getId());
        detail.setCustomerId(order.getCustomerId());
        detail.setStatus(order.getStatus());
        detail.setTotalAmount(order.getTotalAmount());
        detail.setCreatedAt(order.getCreatedAt());
        detail.setUpdatedAt(order.getUpdatedAt());
        detail.setShippingAddress(order.getShippingAddress());

        // Map items
        List<OrderItemDto> itemDtos = order.getItems().stream()
                .map(item -> new OrderItemDto(
                        item.getProductId(),
                        item.getQuantity(),
                        item.getPrice()
                ))
                .collect(Collectors.toList());

        detail.setItems(itemDtos);

        // Get payment info if available
        order.getPayment().ifPresent(payment -> {
            PaymentInfo paymentInfo = new PaymentInfo(
                    payment.getPaymentId(),
                    payment.getAmount(),
                    payment.getMethod(),
                    payment.getStatus(),
                    payment.getTimestamp()
            );

            detail.setPayment(paymentInfo);
        });

        return detail;
    }

    @Transactional(readOnly = true)
    public Page<OrderSummary> searchOrders(OrderSearchCriteria criteria, Pageable
pageable) {
        // Use specification for dynamic querying
        Specification<OrderSummary> spec = Specification.where(null);

        if (criteria.getCustomerId() != null) {
            spec =
spec.and(OrderSpecifications.hasCustomerId(criteria.getCustomerId()));
        }

        if (criteria.getStatus() != null) {
            spec = spec.and(OrderSpecifications.hasStatus(criteria.getStatus()));
        }

        if (criteria.getFromDate() != null) {
            spec =
spec.and(OrderSpecifications.createdAfter(criteria.getFromDate()));
```

```
        }

        if (criteria.getToDate() != null) {
            spec =
spec.and(OrderSpecifications.createdBefore(criteria.getToDate()));
        }

        if (criteria.getMinAmount() != null) {
            spec =
spec.and(OrderSpecifications.hasMinAmount(criteria.getMinAmount()));
        }

        return orderSummaryRepository.findAll(spec, pageable);
    }
}

// Event handler to keep read model updated
@Service
public class OrderEventHandler {

    private final OrderSummaryRepository summaryRepository;
    private final ObjectMapper objectMapper;

    // Constructor injection

    @EventListener
    public void handleOrderCreatedEvent(OrderCreatedEvent event) {
        // Create order summary for read model
        OrderSummary summary = new OrderSummary();
        summary.setOrderId(event.getOrderId());
        summary.setCustomerId(event.getCustomerId());
        summary.setStatus(OrderStatus.CREATED);
        summary.setTotalAmount(event.getTotalAmount());
        summary.setCreatedAt(event.getCreatedAt());
        summary.setItemCount(event.getItems().size());

        summaryRepository.save(summary);
    }

    @EventListener
    public void handleOrderStatusChangedEvent(OrderStatusChangedEvent event) {
        // Update order summary
        OrderSummary summary = summaryRepository.findById(event.getOrderId())
                .orElseThrow(() -> new ResourceNotFoundException("Order summary
not found: " + event.getOrderId()));

        summary.setStatus(OrderStatus.valueOf(event.getNewStatus()));
        summary.setUpdatedAt(event.getTimestamp());

        summaryRepository.save(summary);
    }

    @StreamListener(target = "order-events")
    public void handleStreamEvent(Message<String> message) throws
```

```
JsonProcessingException {
        // Parse event type from headers
        String eventType = message.getHeaders().get("eventType", String.class);

        if (eventType == null) {
            log.warn("Event received without eventType header");
            return;
        }

        // Process based on event type
        switch (eventType) {
            case "ORDER_CREATED":
                OrderCreatedEvent createdEvent = objectMapper.readValue(
                        message.getPayload(), OrderCreatedEvent.class);
                handleOrderCreatedEvent(createdEvent);
                break;

            case "ORDER_STATUS_CHANGED":
                OrderStatusChangedEvent statusEvent = objectMapper.readValue(
                        message.getPayload(), OrderStatusChangedEvent.class);
                handleOrderStatusChangedEvent(statusEvent);
                break;

            default:
                log.warn("Unknown event type: {}", eventType);
        }
    }
}
```

F. Testing Spring Boot Applications

Unit Testing

Unit testing focuses on testing individual components in isolation.

Setting Up JUnit 5

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

Testing Service Layer

```
@ExtendWith(MockitoExtension.class)
public class ProductServiceTest {
```

```
    @Mock
    private ProductRepository productRepository;

    @Mock
    private CategoryRepository categoryRepository;

    @InjectMocks
    private ProductServiceImpl productService;

    @Test
    void createProduct_ValidData_ReturnsProduct() {
        // Arrange
        Long categoryId = 1L;
        Category category = new Category();
        category.setId(categoryId);
        category.setName("Electronics");

        ProductDto productDto = new ProductDto();
        productDto.setName("Smartphone");
        productDto.setDescription("Latest model");
        productDto.setPrice(new BigDecimal("999.99"));
        productDto.setCategoryId(categoryId);

        Product product = new Product();
        product.setName(productDto.getName());
        product.setDescription(productDto.getDescription());
        product.setPrice(productDto.getPrice());
        product.setCategory(category);

        Product savedProduct = new Product();
        savedProduct.setId(1L);
        savedProduct.setName(product.getName());
        savedProduct.setDescription(product.getDescription());
        savedProduct.setPrice(product.getPrice());
        savedProduct.setCategory(category);

        // Mock repository behavior

when(categoryRepository.findById(categoryId)).thenReturn(Optional.of(category));
        when(productRepository.save(any(Product.class))).thenReturn(savedProduct);

        // Act
        Product result = productService.createProduct(productDto);

        // Assert
        assertNotNull(result);
        assertEquals(savedProduct.getId(), result.getId());
        assertEquals(productDto.getName(), result.getName());
        assertEquals(productDto.getDescription(), result.getDescription());
        assertEquals(productDto.getPrice(), result.getPrice());
        assertEquals(categoryId, result.getCategory().getId());

        // Verify repository interactions
        verify(categoryRepository).findById(categoryId);
```

```
        verify(productRepository).save(any(Product.class));
    }

    @Test
    void createProduct_CategoryNotFound_ThrowsResourceNotFoundException() {
        // Arrange
        Long categoryId = 1L;

        ProductDto productDto = new ProductDto();
        productDto.setName("Smartphone");
        productDto.setDescription("Latest model");
        productDto.setPrice(new BigDecimal("999.99"));
        productDto.setCategoryId(categoryId);

        // Mock repository behavior

when(categoryRepository.findById(categoryId)).thenReturn(Optional.empty());

        // Act & Assert
        ResourceNotFoundException exception =
assertThrows(ResourceNotFoundException.class, () -> {
            productService.createProduct(productDto);
        });

        assertEquals("Category not found with id: " + categoryId,
exception.getMessage());

        // Verify repository interactions
        verify(categoryRepository).findById(categoryId);
        verify(productRepository, never()).save(any(Product.class));
    }

    @Test
    void findById_ExistingProduct_ReturnsProduct() {
        // Arrange
        Long productId = 1L;

        Product product = new Product();
        product.setId(productId);
        product.setName("Smartphone");
        product.setDescription("Latest model");
        product.setPrice(new BigDecimal("999.99"));

        // Mock repository behavior

when(productRepository.findById(productId)).thenReturn(Optional.of(product));

        // Act
        Product result = productService.findById(productId);

        // Assert
        assertNotNull(result);
        assertEquals(productId, result.getId());
        assertEquals(product.getName(), result.getName());
```

```
        // Verify repository interactions
        verify(productRepository).findById(productId);
    }

    @Test
    void findById_NonExistingProduct_ThrowsResourceNotFoundException() {
        // Arrange
        Long productId = 1L;

        // Mock repository behavior
        when(productRepository.findById(productId)).thenReturn(Optional.empty());

        // Act & Assert
        ResourceNotFoundException exception =
assertThrows(ResourceNotFoundException.class, () -> {
            productService.findById(productId);
        });

        assertEquals("Product not found with id: " + productId,
exception.getMessage());

        // Verify repository interactions
        verify(productRepository).findById(productId);
    }

    @Test
    void deleteProduct_ExistingProduct_DeletesProduct() {
        // Arrange
        Long productId = 1L;

        Product product = new Product();
        product.setId(productId);

        // Mock repository behavior

when(productRepository.findById(productId)).thenReturn(Optional.of(product));
        doNothing().when(productRepository).delete(product);

        // Act
        productService.deleteProduct(productId);

        // Verify repository interactions
        verify(productRepository).findById(productId);
        verify(productRepository).delete(product);
    }

    @Test
    void updateProduct_ValidData_ReturnsUpdatedProduct() {
        // Arrange
        Long productId = 1L;
        Long categoryId = 2L;

        Category category = new Category();
```

```
        category.setId(categoryId);
        category.setName("Electronics");

        Product existingProduct = new Product();
        existingProduct.setId(productId);
        existingProduct.setName("Old Name");
        existingProduct.setDescription("Old Description");
        existingProduct.setPrice(new BigDecimal("899.99"));

        ProductDto updateDto = new ProductDto();
        updateDto.setName("Updated Name");
        updateDto.setDescription("Updated Description");
        updateDto.setPrice(new BigDecimal("999.99"));
        updateDto.setCategoryId(categoryId);

        // Mock repository behavior

when(productRepository.findById(productId)).thenReturn(Optional.of(existingProduct
));

when(categoryRepository.findById(categoryId)).thenReturn(Optional.of(category));
        when(productRepository.save(any(Product.class))).thenAnswer(invocation ->
invocation.getArgument(0));

        // Act
        Product result = productService.updateProduct(productId, updateDto);

        // Assert
        assertNotNull(result);
        assertEquals(productId, result.getId());
        assertEquals(updateDto.getName(), result.getName());
        assertEquals(updateDto.getDescription(), result.getDescription());
        assertEquals(updateDto.getPrice(), result.getPrice());
        assertEquals(category, result.getCategory());

        // Verify repository interactions
        verify(productRepository).findById(productId);
        verify(categoryRepository).findById(categoryId);
        verify(productRepository).save(any(Product.class));
    }
}
```

Testing Repositories

```
@DataJpaTest
@AutoConfigureTestDatabase(replace = AutoConfigureTestDatabase.Replace.NONE)
public class ProductRepositoryTest {

    @Autowired
    private ProductRepository productRepository;
```

```
    @Autowired
    private CategoryRepository categoryRepository;

    @Test
    void findByName_ExistingName_ReturnsProduct() {
        // Arrange
        String productName = "Test Product";

        Category category = new Category();
        category.setName("Test Category");
        categoryRepository.save(category);

        Product product = new Product();
        product.setName(productName);
        product.setDescription("Test Description");
        product.setPrice(new BigDecimal("99.99"));
        product.setCategory(category);
        productRepository.save(product);

        // Act
        Optional<Product> result = productRepository.findByName(productName);

        // Assert
        assertTrue(result.isPresent());
        assertEquals(productName, result.get().getName());
    }

    @Test
    void findByName_NonExistingName_ReturnsEmptyOptional() {
        // Arrange
        String productName = "Non-existing Product";

        // Act
        Optional<Product> result = productRepository.findByName(productName);

        // Assert
        assertFalse(result.isPresent());
    }

    @Test
    void findByCategoryId_ExistingCategory_ReturnsProducts() {
        // Arrange
        Category category = new Category();
        category.setName("Test Category");
        categoryRepository.save(category);

        Product product1 = new Product();
        product1.setName("Product 1");
        product1.setDescription("Description 1");
        product1.setPrice(new BigDecimal("99.99"));
        product1.setCategory(category);
        productRepository.save(product1);

        Product product2 = new Product();
```

```
        product2.setName("Product 2");
        product2.setDescription("Description 2");
        product2.setPrice(new BigDecimal("199.99"));
        product2.setCategory(category);
        productRepository.save(product2);

        // Act
        List<Product> results =
productRepository.findByCategoryId(category.getId());

        // Assert
        assertEquals(2, results.size());
        assertTrue(results.stream().anyMatch(p -> p.getName().equals("Product
1")));
        assertTrue(results.stream().anyMatch(p -> p.getName().equals("Product
2")));
    }

    @Test
    void findByPriceBetween_ProductsInRange_ReturnsProducts() {
        // Arrange
        Category category = new Category();
        category.setName("Test Category");
        categoryRepository.save(category);

        Product product1 = new Product();
        product1.setName("Cheap Product");
        product1.setDescription("Description 1");
        product1.setPrice(new BigDecimal("10.00"));
        product1.setCategory(category);
        productRepository.save(product1);

        Product product2 = new Product();
        product2.setName("Medium Product");
        product2.setDescription("Description 2");
        product2.setPrice(new BigDecimal("50.00"));
        product2.setCategory(category);
        productRepository.save(product2);

        Product product3 = new Product();
        product3.setName("Expensive Product");
        product3.setDescription("Description 3");
        product3.setPrice(new BigDecimal("100.00"));
        product3.setCategory(category);
        productRepository.save(product3);

        // Act
        List<Product> results = productRepository.findByPriceBetween(
                new BigDecimal("20.00"), new BigDecimal("80.00"));

        // Assert
        assertEquals(1, results.size());
        assertEquals("Medium Product", results.get(0).getName());
```

```
    }
}
```

Testing Controllers

```
@WebMvcTest(ProductController.class)
public class ProductControllerTest {

    @Autowired
    private MockMvc mockMvc;

    @MockBean
    private ProductService productService;

    @Autowired
    private ObjectMapper objectMapper;

    @Test
    void getAllProducts_ReturnsProductList() throws Exception {
        // Arrange
        List<ProductDto> products = List.of(
                new ProductDto(1L, "Product 1", "Description 1", new
BigDecimal("99.99"), 1L),
                new ProductDto(2L, "Product 2", "Description 2", new
BigDecimal("199.99"), 2L)
        );

        when(productService.findAll()).thenReturn(products);

        // Act & Assert
        mockMvc.perform(get("/api/products")
                .contentType(MediaType.APPLICATION_JSON))
                .andExpect(status().isOk())
                .andExpect(content().contentType(MediaType.APPLICATION_JSON))
                .andExpect(jsonPath("$.size()").value(2))
                .andExpect(jsonPath("$[0].id").value(1))
                .andExpect(jsonPath("$[0].name").value("Product 1"))
                .andExpect(jsonPath("$[1].id").value(2))
                .andExpect(jsonPath("$[1].name").value("Product 2"));

        verify(productService).findAll();
    }

    @Test
    void getProductById_ExistingProduct_ReturnsProduct() throws Exception {
        // Arrange
        Long productId = 1L;
        ProductDto product = new ProductDto(
                productId, "Test Product", "Test Description", new
BigDecimal("99.99"), 1L);
```

```
        when(productService.findById(productId)).thenReturn(product);

        // Act & Assert
        mockMvc.perform(get("/api/products/{id}", productId)
                .contentType(MediaType.APPLICATION_JSON))
                .andExpect(status().isOk())
                .andExpect(content().contentType(MediaType.APPLICATION_JSON))
                .andExpect(jsonPath("$.id").value(productId))
                .andExpect(jsonPath("$.name").value("Test Product"))
                .andExpect(jsonPath("$.description").value("Test Description"))
                .andExpect(jsonPath("$.price").value(99.99))
                .andExpect(jsonPath("$.categoryId").value(1));

        verify(productService).findById(productId);
    }

    @Test
    void getProductById_NonExistingProduct_ReturnsNotFound() throws Exception {
        // Arrange
        Long productId = 99L;

        when(productService.findById(productId))
                .thenThrow(new ResourceNotFoundException("Product not found with
id: " + productId));

        // Act & Assert
        mockMvc.perform(get("/api/products/{id}", productId)
                .contentType(MediaType.APPLICATION_JSON))
                .andExpect(status().isNotFound())
                .andExpect(jsonPath("$.status").value(404))
                .andExpect(jsonPath("$.message").value("Product not found with id:
" + productId));

        verify(productService).findById(productId);
    }

    @Test
    void createProduct_ValidData_ReturnsCreatedProduct() throws Exception {
        // Arrange
        ProductDto productDto = new ProductDto(
                null, "New Product", "New Description", new BigDecimal("149.99"),
1L);

        ProductDto createdProductDto = new ProductDto(
                1L, "New Product", "New Description", new BigDecimal("149.99"),
1L);


when(productService.createProduct(any(ProductDto.class))).thenReturn(createdProduc
tDto);

        // Act & Assert
        mockMvc.perform(post("/api/products")
                .contentType(MediaType.APPLICATION_JSON)
```

```
                .content(objectMapper.writeValueAsString(productDto)))
                .andExpect(status().isCreated())
                .andExpect(content().contentType(MediaType.APPLICATION_JSON))
                .andExpect(jsonPath("$.id").value(1))
                .andExpect(jsonPath("$.name").value("New Product"))
                .andExpect(jsonPath("$.description").value("New Description"))
                .andExpect(jsonPath("$.price").value(149.99))
                .andExpect(jsonPath("$.categoryId").value(1))
                .andExpect(header().string("Location",
containsString("/api/products/1")));

        verify(productService).createProduct(any(ProductDto.class));
    }

    @Test
    void createProduct_InvalidData_ReturnsBadRequest() throws Exception {
        // Arrange
        ProductDto invalidProductDto = new ProductDto(
                null, "", null, new BigDecimal("-10.00"), null);

        // Act & Assert
        mockMvc.perform(post("/api/products")
                .contentType(MediaType.APPLICATION_JSON)
                .content(objectMapper.writeValueAsString(invalidProductDto)))
                .andExpect(status().isBadRequest())
                .andExpect(jsonPath("$.status").value(400))
                .andExpect(jsonPath("$.errors").isArray())
                .andExpect(jsonPath("$.errors", hasSize(greaterThan(0))));

        verify(productService, never()).createProduct(any(ProductDto.class));
    }

    @Test
    void updateProduct_ValidData_ReturnsUpdatedProduct() throws Exception {
        // Arrange
        Long productId = 1L;
        ProductDto updateDto = new ProductDto(
                productId, "Updated Product", "Updated Description", new
BigDecimal("199.99"), 2L);

        when(productService.updateProduct(eq(productId),
any(ProductDto.class))).thenReturn(updateDto);

        // Act & Assert
        mockMvc.perform(put("/api/products/{id}", productId)
                .contentType(MediaType.APPLICATION_JSON)
                .content(objectMapper.writeValueAsString(updateDto)))
                .andExpect(status().isOk())
                .andExpect(content().contentType(MediaType.APPLICATION_JSON))
                .andExpect(jsonPath("$.id").value(productId))
                .andExpect(jsonPath("$.name").value("Updated Product"))
                .andExpect(jsonPath("$.description").value("Updated Description"))
                .andExpect(jsonPath("$.price").value(199.99))
                .andExpect(jsonPath("$.categoryId").value(2));
```

```
        verify(productService).updateProduct(eq(productId),
any(ProductDto.class));
    }

    @Test
    void deleteProduct_ExistingProduct_ReturnsNoContent() throws Exception {
        // Arrange
        Long productId = 1L;

        doNothing().when(productService).deleteProduct(productId);

        // Act & Assert
        mockMvc.perform(delete("/api/products/{id}", productId))
                .andExpect(status().isNoContent());

        verify(productService).deleteProduct(productId);
    }

    @Test
    @WithMockUser(roles = "ADMIN")
    void adminEndpoint_WithAdminRole_ReturnsSuccess() throws Exception {
        // Arrange
        when(productService.getAdminStatistics()).thenReturn(Map.of(
                "totalProducts", 10,
                "totalCategories", 5,
                "averagePrice", 199.99
        ));

        // Act & Assert
        mockMvc.perform(get("/api/products/admin/statistics")
                .contentType(MediaType.APPLICATION_JSON))
                .andExpect(status().isOk())
                .andExpect(jsonPath("$.totalProducts").value(10))
                .andExpect(jsonPath("$.totalCategories").value(5))
                .andExpect(jsonPath("$.averagePrice").value(199.99));

        verify(productService).getAdminStatistics();
    }

    @Test
    @WithMockUser(roles = "USER")
    void adminEndpoint_WithUserRole_ReturnsForbidden() throws Exception {
        // Act & Assert
        mockMvc.perform(get("/api/products/admin/statistics")
                .contentType(MediaType.APPLICATION_JSON))
                .andExpect(status().isForbidden());

        verify(productService, never()).getAdminStatistics();
    }

    @Test
    void searchProducts_ReturnsFilteredResults() throws Exception {
        // Arrange
```

```
        List<ProductDto> filteredProducts = List.of(
                new ProductDto(1L, "Smartphone", "Latest model", new
BigDecimal("999.99"), 1L),
                new ProductDto(3L, "Smart TV", "4K Ultra HD", new
BigDecimal("1299.99"), 1L)
        );

        when(productService.searchProducts(eq("smart"), isNull(), isNull()))
                .thenReturn(filteredProducts);

        // Act & Assert
        mockMvc.perform(get("/api/products/search")
                .param("keyword", "smart")
                .contentType(MediaType.APPLICATION_JSON))
                .andExpect(status().isOk())
                .andExpect(content().contentType(MediaType.APPLICATION_JSON))
                .andExpect(jsonPath("$.size()").value(2))
                .andExpect(jsonPath("$[0].name").value("Smartphone"))
                .andExpect(jsonPath("$[1].name").value("Smart TV"));

        verify(productService).searchProducts(eq("smart"), isNull(), isNull());
    }
}
```

Testing with WebTestClient

For reactive applications or to test controllers with a more fluent API:

```
@WebFluxTest(ProductController.class)
public class ProductControllerWebFluxTest {

    @Autowired
    private WebTestClient webTestClient;

    @MockBean
    private ProductService productService;

    @Test
    void getAllProducts_ReturnsProductList() {
        // Arrange
        List<ProductDto> products = List.of(
                new ProductDto(1L, "Product 1", "Description 1", new
BigDecimal("99.99"), 1L),
                new ProductDto(2L, "Product 2", "Description 2", new
BigDecimal("199.99"), 2L)
        );

        when(productService.findAll()).thenReturn(products);

        // Act & Assert
        webTestClient.get()
```

```
                .uri("/api/products")
                .accept(MediaType.APPLICATION_JSON)
                .exchange()
                .expectStatus().isOk()
                .expectHeader().contentType(MediaType.APPLICATION_JSON)
                .expectBodyList(ProductDto.class)
                .hasSize(2)
                .contains(products.toArray(new ProductDto[0]));

        verify(productService).findAll();
    }

    @Test
    void getProductById_ExistingProduct_ReturnsProduct() {
        // Arrange
        Long productId = 1L;
        ProductDto product = new ProductDto(
                productId, "Test Product", "Test Description", new
BigDecimal("99.99"), 1L);

        when(productService.findById(productId)).thenReturn(product);

        // Act & Assert
        webTestClient.get()
                .uri("/api/products/{id}", productId)
                .accept(MediaType.APPLICATION_JSON)
                .exchange()
                .expectStatus().isOk()
                .expectHeader().contentType(MediaType.APPLICATION_JSON)
                .expectBody(ProductDto.class)
                .isEqualTo(product);

        verify(productService).findById(productId);
    }

    @Test
    void createProduct_ValidData_ReturnsCreatedProduct() {
        // Arrange
        ProductDto productDto = new ProductDto(
                null, "New Product", "New Description", new BigDecimal("149.99"),
1L);

        ProductDto createdProductDto = new ProductDto(
                1L, "New Product", "New Description", new BigDecimal("149.99"),
1L);


when(productService.createProduct(any(ProductDto.class))).thenReturn(createdProduc
tDto);

        // Act & Assert
        webTestClient.post()
                .uri("/api/products")
                .contentType(MediaType.APPLICATION_JSON)
```

```
                .bodyValue(productDto)
                .exchange()
                .expectStatus().isCreated()
                .expectHeader().contentType(MediaType.APPLICATION_JSON)
                .expectHeader().location("/api/products/1")
                .expectBody(ProductDto.class)
                .isEqualTo(createdProductDto);

        verify(productService).createProduct(any(ProductDto.class));
    }
}
```

Testing Exception Handling

```
@ExtendWith(MockitoExtension.class)
public class GlobalExceptionHandlerTest {

    private MockMvc mockMvc;

    @Mock
    private ProductService productService;

    @InjectMocks
    private ProductController productController;

    @BeforeEach
    void setUp() {
        mockMvc = MockMvcBuilders.standaloneSetup(productController)
                .setControllerAdvice(new GlobalExceptionHandler())
                .build();
    }

    @Test
    void handleResourceNotFoundException_ReturnsNotFound() throws Exception {
        // Arrange
        Long productId = 99L;

        when(productService.findById(productId))
                .thenThrow(new ResourceNotFoundException("Product not found with
id: " + productId));

        // Act & Assert
        mockMvc.perform(get("/api/products/{id}", productId)
                .contentType(MediaType.APPLICATION_JSON))
                .andExpect(status().isNotFound())
                .andExpect(jsonPath("$.status").value(404))
                .andExpect(jsonPath("$.message").value("Product not found with id:
" + productId))
                .andExpect(jsonPath("$.timestamp").exists());

        verify(productService).findById(productId);
```

```
    }

    @Test
    void handleValidationException_ReturnsBadRequest() throws Exception {
        // Arrange
        ProductDto invalidProductDto = new ProductDto();
        // Empty product with no values set

        // Act & Assert
        mockMvc.perform(post("/api/products")
                .contentType(MediaType.APPLICATION_JSON)
                .content(new
ObjectMapper().writeValueAsString(invalidProductDto)))
                .andExpect(status().isBadRequest())
                .andExpect(jsonPath("$.status").value(400))
                .andExpect(jsonPath("$.errors").isArray());
    }

    @Test
    void handleAccessDeniedException_ReturnsForbidden() throws Exception {
        // Arrange
        when(productService.getAdminStatistics())
                .thenThrow(new AccessDeniedException("Access denied"));

        // Act & Assert
        mockMvc.perform(get("/api/products/admin/statistics")
                .contentType(MediaType.APPLICATION_JSON))
                .andExpect(status().isForbidden())
                .andExpect(jsonPath("$.status").value(403))
                .andExpect(jsonPath("$.message").value("Access denied"));
    }
}
```

Integration Testing

Integration testing verifies that different components work together correctly.

Setting Up Integration Tests

```
@SpringBootTest
@AutoConfigureMockMvc
public class ProductIntegrationTest {

    @Autowired
    private MockMvc mockMvc;

    @Autowired
    private ProductRepository productRepository;

    @Autowired
    private CategoryRepository categoryRepository;
```

```
    @Autowired
    private ObjectMapper objectMapper;

    @BeforeEach
    void setUp() {
        productRepository.deleteAll();
        categoryRepository.deleteAll();
    }

    @Test
    void createAndGetProduct_Success() throws Exception {
        // Create a category first
        Category category = new Category();
        category.setName("Electronics");
        category = categoryRepository.save(category);

        // Create product dto
        ProductDto productDto = new ProductDto();
        productDto.setName("Smartphone");
        productDto.setDescription("Latest model");
        productDto.setPrice(new BigDecimal("999.99"));
        productDto.setCategoryId(category.getId());

        // Create product via API
        String productJson = objectMapper.writeValueAsString(productDto);

        MvcResult createResult = mockMvc.perform(post("/api/products")
                .contentType(MediaType.APPLICATION_JSON)
                .content(productJson))
                .andExpect(status().isCreated())
                .andExpect(content().contentType(MediaType.APPLICATION_JSON))
                .andExpect(jsonPath("$.id").exists())
                .andExpect(jsonPath("$.name").value("Smartphone"))
                .andReturn();

        // Extract product ID from response
        String createResponseJson =
createResult.getResponse().getContentAsString();
        ProductDto createdProduct = objectMapper.readValue(createResponseJson,
ProductDto.class);
        Long productId = createdProduct.getId();

        // Get product via API
        mockMvc.perform(get("/api/products/{id}", productId)
                .contentType(MediaType.APPLICATION_JSON))
                .andExpect(status().isOk())
                .andExpect(content().contentType(MediaType.APPLICATION_JSON))
                .andExpect(jsonPath("$.id").value(productId))
                .andExpect(jsonPath("$.name").value("Smartphone"))
                .andExpect(jsonPath("$.description").value("Latest model"))
                .andExpect(jsonPath("$.price").value(999.99))
                .andExpect(jsonPath("$.categoryId").value(category.getId()));
```

```
        // Verify product is in database
        assertTrue(productRepository.findById(productId).isPresent());
    }

    @Test
    void createUpdateAndDeleteProduct_Success() throws Exception {
        // Create a category
        Category category = new Category();
        category.setName("Electronics");
        category = categoryRepository.save(category);

        // Create another category for update
        Category newCategory = new Category();
        newCategory.setName("Phones");
        newCategory = categoryRepository.save(newCategory);

        // Create product
        ProductDto productDto = new ProductDto();
        productDto.setName("Old Smartphone");
        productDto.setDescription("Basic model");
        productDto.setPrice(new BigDecimal("499.99"));
        productDto.setCategoryId(category.getId());

        String productJson = objectMapper.writeValueAsString(productDto);

        MvcResult createResult = mockMvc.perform(post("/api/products")
                .contentType(MediaType.APPLICATION_JSON)
                .content(productJson))
                .andExpect(status().isCreated())
                .andReturn();

        String createResponseJson =
createResult.getResponse().getContentAsString();
        ProductDto createdProduct = objectMapper.readValue(createResponseJson,
ProductDto.class);
        Long productId = createdProduct.getId();

        // Update product
        productDto.setId(productId);
        productDto.setName("New Smartphone");
        productDto.setDescription("Premium model");
        productDto.setPrice(new BigDecimal("999.99"));
        productDto.setCategoryId(newCategory.getId());

        String updatedProductJson = objectMapper.writeValueAsString(productDto);

        mockMvc.perform(put("/api/products/{id}", productId)
                .contentType(MediaType.APPLICATION_JSON)
                .content(updatedProductJson))
                .andExpect(status().isOk())
                .andExpect(jsonPath("$.id").value(productId))
                .andExpect(jsonPath("$.name").value("New Smartphone"))
                .andExpect(jsonPath("$.description").value("Premium model"))
                .andExpect(jsonPath("$.price").value(999.99))
```

```
                .andExpect(jsonPath("$.categoryId").value(newCategory.getId()));

        // Verify product updated in database
        Optional<Product> updatedProductOpt =
productRepository.findById(productId);
        assertTrue(updatedProductOpt.isPresent());
        Product updatedProduct = updatedProductOpt.get();
        assertEquals("New Smartphone", updatedProduct.getName());
        assertEquals("Premium model", updatedProduct.getDescription());
        assertEquals(0, new
BigDecimal("999.99").compareTo(updatedProduct.getPrice()));
        assertEquals(newCategory.getId(), updatedProduct.getCategory().getId());

        // Delete product
        mockMvc.perform(delete("/api/products/{id}", productId))
                .andExpect(status().isNoContent());

        // Verify product deleted from database
        assertFalse(productRepository.findById(productId).isPresent());
    }

    @Test
    void searchProducts_ReturnsMatchingProducts() throws Exception {
        // Create a category
        Category category = new Category();
        category.setName("Electronics");
        category = categoryRepository.save(category);

        // Create several products
        Product product1 = new Product();
        product1.setName("Smartphone X");
        product1.setDescription("Latest model");
        product1.setPrice(new BigDecimal("999.99"));
        product1.setCategory(category);
        productRepository.save(product1);

        Product product2 = new Product();
        product2.setName("Laptop Pro");
        product2.setDescription("Powerful laptop");
        product2.setPrice(new BigDecimal("1499.99"));
        product2.setCategory(category);
        productRepository.save(product2);

        Product product3 = new Product();
        product3.setName("Smart Watch");
        product3.setDescription("Fitness tracker");
        product3.setPrice(new BigDecimal("299.99"));
        product3.setCategory(category);
        productRepository.save(product3);

        // Search for products containing "smart"
        mockMvc.perform(get("/api/products/search")
                .param("keyword", "smart")
                .contentType(MediaType.APPLICATION_JSON))
```

```
                .andExpect(status().isOk())
                .andExpect(jsonPath("$", hasSize(2)))
                .andExpect(jsonPath("$[0].name", anyOf(is("Smartphone X"),
is("Smart Watch"))))
                .andExpect(jsonPath("$[1].name", anyOf(is("Smartphone X"),
is("Smart Watch"))));

        // Search by price range
        mockMvc.perform(get("/api/products/search")
                .param("minPrice", "500")
                .param("maxPrice", "1200")
                .contentType(MediaType.APPLICATION_JSON))
                .andExpect(status().isOk())
                .andExpect(jsonPath("$", hasSize(1)))
                .andExpect(jsonPath("$[0].name").value("Smartphone X"));
    }
}
```

Testing with TestRestTemplate

For full integration testing against a real HTTP server:

```
@SpringBootTest(webEnvironment = SpringBootTest.WebEnvironment.RANDOM_PORT)
public class ProductApiIntegrationTest {

    @Autowired
    private TestRestTemplate restTemplate;

    @Autowired
    private ProductRepository productRepository;

    @Autowired
    private CategoryRepository categoryRepository;

    @BeforeEach
    void setUp() {
        productRepository.deleteAll();
        categoryRepository.deleteAll();
    }

    @Test
    void crudOperations_Success() {
        // Create a category
        Category category = new Category();
        category.setName("Electronics");
        category = categoryRepository.save(category);

        // Create a product using the API
        ProductDto newProductDto = new ProductDto();
        newProductDto.setName("Test Product");
        newProductDto.setDescription("Test Description");
```

```
        newProductDto.setPrice(new BigDecimal("99.99"));
        newProductDto.setCategoryId(category.getId());

        ResponseEntity<ProductDto> createResponse = restTemplate.postForEntity(
                "/api/products", newProductDto, ProductDto.class);

        // Verify creation
        assertEquals(HttpStatus.CREATED, createResponse.getStatusCode());
        assertNotNull(createResponse.getBody());
        assertNotNull(createResponse.getBody().getId());
        assertEquals("Test Product", createResponse.getBody().getName());

        Long productId = createResponse.getBody().getId();

        // Get the product by ID
        ResponseEntity<ProductDto> getResponse = restTemplate.getForEntity(
                "/api/products/{id}", ProductDto.class, productId);

        // Verify retrieval
        assertEquals(HttpStatus.OK, getResponse.getStatusCode());
        assertNotNull(getResponse.getBody());
        assertEquals(productId, getResponse.getBody().getId());
        assertEquals("Test Product", getResponse.getBody().getName());

        // Update the product
        ProductDto updateDto = new ProductDto();
        updateDto.setId(productId);
        updateDto.setName("Updated Product");
        updateDto.setDescription("Updated Description");
        updateDto.setPrice(new BigDecimal("149.99"));
        updateDto.setCategoryId(category.getId());

        restTemplate.put("/api/products/{id}", updateDto, productId);

        // Verify update
        ResponseEntity<ProductDto> updatedGetResponse = restTemplate.getForEntity(
                "/api/products/{id}", ProductDto.class, productId);

        assertEquals(HttpStatus.OK, updatedGetResponse.getStatusCode());
        assertNotNull(updatedGetResponse.getBody());
        assertEquals("Updated Product", updatedGetResponse.getBody().getName());
        assertEquals(0, new
BigDecimal("149.99").compareTo(updatedGetResponse.getBody().getPrice()));

        // Delete the product
        restTemplate.delete("/api/products/{id}", productId);

        // Verify deletion
        ResponseEntity<ProductDto> deletedGetResponse = restTemplate.getForEntity(
                "/api/products/{id}", ProductDto.class, productId);

        assertEquals(HttpStatus.NOT_FOUND, deletedGetResponse.getStatusCode());
    }
```

```
    @Test
    void getAllProducts_ReturnsProductList() {
        // Create a category
        Category category = new Category();
        category.setName("Electronics");
        category = categoryRepository.save(category);

        // Create sample products
        Product product1 = new Product();
        product1.setName("Product 1");
        product1.setDescription("Description 1");
        product1.setPrice(new BigDecimal("99.99"));
        product1.setCategory(category);
        productRepository.save(product1);

        Product product2 = new Product();
        product2.setName("Product 2");
        product2.setDescription("Description 2");
        product2.setPrice(new BigDecimal("199.99"));
        product2.setCategory(category);
        productRepository.save(product2);

        // Get all products
        ResponseEntity<ProductDto[]> response = restTemplate.getForEntity(
                "/api/products", ProductDto[].class);

        // Verify response
        assertEquals(HttpStatus.OK, response.getStatusCode());
        assertNotNull(response.getBody());
        assertEquals(2, response.getBody().length);
    }

    @Test
    void createProduct_InvalidData_ReturnsBadRequest() {
        // Create an invalid product with missing required fields
        ProductDto invalidProductDto = new ProductDto();
        // No name, description, price, or categoryId set

        ResponseEntity<ErrorResponse> response = restTemplate.postForEntity(
                "/api/products", invalidProductDto, ErrorResponse.class);

        // Verify bad request response
        assertEquals(HttpStatus.BAD_REQUEST, response.getStatusCode());
        assertNotNull(response.getBody());
        assertEquals(400, response.getBody().getStatus());
        assertNotNull(response.getBody().getErrors());
        assertFalse(response.getBody().getErrors().isEmpty());
    }

    // Helper class for error responses
    @Data
    static class ErrorResponse {
        private int status;
        private String message;
```

```
        private List<String> errors;
        private LocalDateTime timestamp;
    }
}
```

Testing with @DataJpaTest and @WebMvcTest

Using test slices for focused testing:

```
@DataJpaTest
public class ProductRepositoryIntegrationTest {

    @Autowired
    private ProductRepository productRepository;

    @Autowired
    private CategoryRepository categoryRepository;

    @Autowired
    private TestEntityManager entityManager;

    @Test
    void findByCategory_ReturnsCategoryProducts() {
        // Create a category
        Category category = new Category();
        category.setName("Electronics");
        category = entityManager.persistAndFlush(category);

        // Create products in that category
        Product product1 = new Product();
        product1.setName("Product 1");
        product1.setDescription("Description 1");
        product1.setPrice(new BigDecimal("99.99"));
        product1.setCategory(category);
        entityManager.persistAndFlush(product1);

        Product product2 = new Product();
        product2.setName("Product 2");
        product2.setDescription("Description 2");
        product2.setPrice(new BigDecimal("199.99"));
        product2.setCategory(category);
        entityManager.persistAndFlush(product2);

        // Create a product in a different category
        Category otherCategory = new Category();
        otherCategory.setName("Other");
        otherCategory = entityManager.persistAndFlush(otherCategory);

        Product product3 = new Product();
        product3.setName("Product 3");
        product3.setDescription("Description 3");
```

```
        product3.setPrice(new BigDecimal("299.99"));
        product3.setCategory(otherCategory);
        entityManager.persistAndFlush(product3);

        // Test findByCategoryId
        List<Product> products =
productRepository.findByCategoryId(category.getId());

        // Verify results
        assertEquals(2, products.size());
        assertTrue(products.stream().anyMatch(p -> p.getName().equals("Product
1")));
        assertTrue(products.stream().anyMatch(p -> p.getName().equals("Product
2")));
        assertFalse(products.stream().anyMatch(p -> p.getName().equals("Product
3")));
    }

    @Test
    void findByPriceRange_ReturnsMatchingProducts() {
        // Create a category
        Category category = new Category();
        category.setName("Electronics");
        category = entityManager.persistAndFlush(category);

        // Create products with different prices
        Product product1 = new Product();
        product1.setName("Budget Product");
        product1.setDescription("Affordable");
        product1.setPrice(new BigDecimal("49.99"));
        product1.setCategory(category);
        entityManager.persistAndFlush(product1);

        Product product2 = new Product();
        product2.setName("Mid-range Product");
        product2.setDescription("Good value");
        product2.setPrice(new BigDecimal("149.99"));
        product2.setCategory(category);
        entityManager.persistAndFlush(product2);

        Product product3 = new Product();
        product3.setName("Premium Product");
        product3.setDescription("High-end");
        product3.setPrice(new BigDecimal("299.99"));
        product3.setCategory(category);
        entityManager.persistAndFlush(product3);

        // Test findByPriceBetween
        List<Product> products = productRepository.findByPriceBetween(
                new BigDecimal("100"), new BigDecimal("200"));

        // Verify results
        assertEquals(1, products.size());
        assertEquals("Mid-range Product", products.get(0).getName());
```

```
    }

    @Test
    void customQuery_findProductAboveAveragePrice() {
        // Create a category
        Category category = new Category();
        category.setName("Electronics");
        category = entityManager.persistAndFlush(category);

        // Create products with different prices
        Product product1 = new Product();
        product1.setName("Budget Product");
        product1.setPrice(new BigDecimal("49.99"));
        product1.setCategory(category);
        entityManager.persistAndFlush(product1);

        Product product2 = new Product();
        product2.setName("Mid-range Product");
        product2.setPrice(new BigDecimal("149.99"));
        product2.setCategory(category);
        entityManager.persistAndFlush(product2);

        Product product3 = new Product();
        product3.setName("Premium Product");
        product3.setPrice(new BigDecimal("299.99"));
        product3.setCategory(category);
        entityManager.persistAndFlush(product3);

        // Average price is (49.99 + 149.99 + 299.99) / 3 = 166.66

        // Test custom query to find products above average price
        List<Product> expensiveProducts =
productRepository.findProductsAboveAveragePrice();

        // Verify results
        assertEquals(1, expensiveProducts.size());
        assertEquals("Premium Product", expensiveProducts.get(0).getName());
    }
}
```

Test Slices

Spring Boot provides specialized test slices for testing specific layers of your application.

@WebMvcTest

Testing controllers in isolation from the rest of the application:

```
@WebMvcTest(ProductController.class)
public class ProductControllerSliceTest {
```

```
    @Autowired
    private MockMvc mockMvc;

    @MockBean
    private ProductService productService;

    @Autowired
    private ObjectMapper objectMapper;

    @Test
    void getAllProducts_ReturnsProductList() throws Exception {
        // This test is identical to the one in the unit test section
        // @WebMvcTest loads only what's needed for testing controllers
    }
}
```

@DataJpaTest

Testing repositories in isolation:

```
@DataJpaTest
@AutoConfigureTestDatabase(replace = AutoConfigureTestDatabase.Replace.NONE)
public class ProductRepositorySliceTest {

    @Autowired
    private ProductRepository productRepository;

    @Autowired
    private TestEntityManager entityManager;

    @Test
    void saveAndFindById_Success() {
        // Create a category
        Category category = new Category();
        category.setName("Electronics");
        category = entityManager.persistAndFlush(category);

        // Create a product
        Product product = new Product();
        product.setName("Test Product");
        product.setDescription("Test Description");
        product.setPrice(new BigDecimal("99.99"));
        product.setCategory(category);

        // Save the product
        product = productRepository.save(product);

        // Clear persistence context to ensure data is fetched from the database
        entityManager.clear();

        // Fetch the product by ID
```

```
        Optional<Product> foundProduct =
productRepository.findById(product.getId());

        // Verify the retrieved product
        assertTrue(foundProduct.isPresent());
        assertEquals("Test Product", foundProduct.get().getName());
        assertEquals("Test Description", foundProduct.get().getDescription());
        assertEquals(0, new
BigDecimal("99.99").compareTo(foundProduct.get().getPrice()));
        assertEquals(category.getId(), foundProduct.get().getCategory().getId());
    }
}
```

@JsonTest

Testing JSON serialization and deserialization:

```
@JsonTest
public class ProductDtoJsonTest {

    @Autowired
    private JacksonTester<ProductDto> json;

    @Test
    void serialize_ReturnsExpectedJson() throws Exception {
        // Create a DTO
        ProductDto productDto = new ProductDto(
                1L, "Test Product", "Test Description", new BigDecimal("99.99"),
2L);

        // Serialize to JSON
        JsonContent<ProductDto> jsonContent = json.write(productDto);

        // Verify the JSON structure

assertThat(jsonContent).extractingJsonPathNumberValue("$.id").isEqualTo(1);

assertThat(jsonContent).extractingJsonPathStringValue("$.name").isEqualTo("Test
Product");

assertThat(jsonContent).extractingJsonPathStringValue("$.description").isEqualTo("
Test Description");

assertThat(jsonContent).extractingJsonPathNumberValue("$.price").isEqualTo(99.99);

assertThat(jsonContent).extractingJsonPathNumberValue("$.categoryId").isEqualTo(2)
;
    }

    @Test
    void deserialize_ReturnsExpectedObject() throws Exception {
```

```
        // Create a JSON string
        String jsonContent = """
                {
                    "id": 1,
                    "name": "Test Product",
                    "description": "Test Description",
                    "price": 99.99,
                    "categoryId": 2
                }
                """;

        // Deserialize from JSON
        ProductDto productDto = json.parse(jsonContent).getObject();

        // Verify the deserialized object
        assertEquals(1L, productDto.getId());
        assertEquals("Test Product", productDto.getName());
        assertEquals("Test Description", productDto.getDescription());
        assertEquals(0, new BigDecimal("99.99").compareTo(productDto.getPrice()));
        assertEquals(2L, productDto.getCategoryId());
    }
}
```

@RestClientTest

Testing REST clients in isolation:

```
@RestClientTest(CategoryClient.class)
public class CategoryClientTest {

    @Autowired
    private CategoryClient categoryClient;

    @Autowired
    private MockRestServiceServer server;

    @Autowired
    private ObjectMapper objectMapper;

    @Test
    void getCategory_ReturnsCategory() throws Exception {
        // Create a category
        CategoryDto category = new CategoryDto(1L, "Electronics");

        // Configure mock server
        server.expect(requestTo("/api/categories/1"))
                .andRespond(withSuccess(objectMapper.writeValueAsString(category),
MediaType.APPLICATION_JSON));

        // Call the client
        CategoryDto result = categoryClient.getCategory(1L);
```

```
        // Verify the result
        assertEquals(1L, result.getId());
        assertEquals("Electronics", result.getName());
    }

    @Test
    void getCategory_ServerError_ThrowsException() {
        // Configure mock server to return an error
        server.expect(requestTo("/api/categories/1"))
                .andRespond(withServerError());

        // Call the client and verify exception is thrown
        assertThrows(RestClientException.class, () ->
categoryClient.getCategory(1L));
    }
}
```

@WebFluxTest

Testing reactive controllers:

```
@WebFluxTest(ProductReactiveController.class)
public class ProductReactiveControllerTest {

    @Autowired
    private WebTestClient webTestClient;

    @MockBean
    private ProductReactiveService productService;

    @Test
    void getAllProducts_ReturnsProducts() {
        // Create test data
        List<ProductDto> products = List.of(
                new ProductDto(1L, "Product 1", "Description 1", new
BigDecimal("99.99"), 1L),
                new ProductDto(2L, "Product 2", "Description 2", new
BigDecimal("199.99"), 1L)
        );

        // Configure mock service
        when(productService.findAll()).thenReturn(Flux.fromIterable(products));

        // Call the controller
        webTestClient.get()
                .uri("/api/reactive/products")
                .accept(MediaType.APPLICATION_JSON)
                .exchange()
                .expectStatus().isOk()
                .expectHeader().contentType(MediaType.APPLICATION_JSON)
```

```
                .expectBodyList(ProductDto.class)
                .hasSize(2)
                .contains(products.toArray(new ProductDto[0]));
    }

    @Test
    void getProductById_ExistingProduct_ReturnsProduct() {
        // Create test data
        ProductDto product = new ProductDto(
                1L, "Product 1", "Description 1", new BigDecimal("99.99"), 1L);

        // Configure mock service
        when(productService.findById(1L)).thenReturn(Mono.just(product));

        // Call the controller
        webTestClient.get()
                .uri("/api/reactive/products/1")
                .accept(MediaType.APPLICATION_JSON)
                .exchange()
                .expectStatus().isOk()
                .expectBody(ProductDto.class)
                .isEqualTo(product);
    }

    @Test
    void getProductById_NonExistingProduct_ReturnsNotFound() {
        // Configure mock service
        when(productService.findById(99L)).thenReturn(Mono.empty());

        // Call the controller
        webTestClient.get()
                .uri("/api/reactive/products/99")
                .accept(MediaType.APPLICATION_JSON)
                .exchange()
                .expectStatus().isNotFound();
    }
}
```

TestContainers

TestContainers allow you to use real infrastructure components in your tests.

Setting Up TestContainers

```
<dependency>
    <groupId>org.testcontainers</groupId>
    <artifactId>testcontainers</artifactId>
    <version>1.17.3</version>
    <scope>test</scope>
</dependency>
<dependency>
```

```
    <groupId>org.testcontainers</groupId>
    <artifactId>junit-jupiter</artifactId>
    <version>1.17.3</version>
    <scope>test</scope>
</dependency>
<dependency>
    <groupId>org.testcontainers</groupId>
    <artifactId>postgresql</artifactId>
    <version>1.17.3</version>
    <scope>test</scope>
</dependency>
```

Testing with a Real Database

```
@SpringBootTest
@Testcontainers
public class ProductServicePostgresqlTest {

    @Container
    private static final PostgreSQLContainer<?> postgresContainer = new
PostgreSQLContainer<>("postgres:14.1")
            .withDatabaseName("testdb")
            .withUsername("test")
            .withPassword("test");

    @DynamicPropertySource
    static void postgresqlProperties(DynamicPropertyRegistry registry) {
        registry.add("spring.datasource.url", postgresContainer::getJdbcUrl);
        registry.add("spring.datasource.username",
postgresContainer::getUsername);
        registry.add("spring.datasource.password",
postgresContainer::getPassword);
        registry.add("spring.jpa.hibernate.ddl-auto", () -> "create-drop");
    }

    @Autowired
    private ProductService productService;

    @Autowired
    private ProductRepository productRepository;

    @Autowired
    private CategoryRepository categoryRepository;

    @BeforeEach
    void setUp() {
        productRepository.deleteAll();
        categoryRepository.deleteAll();
    }

    @Test
```

```
    void createAndRetrieveProduct_Success() {
        // Create a category
        Category category = new Category();
        category.setName("Electronics");
        category = categoryRepository.save(category);

        // Create a product DTO
        ProductDto productDto = new ProductDto();
        productDto.setName("Test Product");
        productDto.setDescription("Test Description");
        productDto.setPrice(new BigDecimal("99.99"));
        productDto.setCategoryId(category.getId());

        // Create the product
        ProductDto createdProductDto = productService.createProduct(productDto);

        // Verify product was created with an ID
        assertNotNull(createdProductDto.getId());
        assertEquals("Test Product", createdProductDto.getName());

        // Retrieve the product
        ProductDto retrievedProductDto =
productService.findById(createdProductDto.getId());

        // Verify retrieved product
        assertEquals(createdProductDto.getId(), retrievedProductDto.getId());
        assertEquals("Test Product", retrievedProductDto.getName());
        assertEquals("Test Description", retrievedProductDto.getDescription());
        assertEquals(0, new
BigDecimal("99.99").compareTo(retrievedProductDto.getPrice()));
        assertEquals(category.getId(), retrievedProductDto.getCategoryId());
    }
}
```

Testing with Multiple Containers

```
@SpringBootTest
@Testcontainers
public class MicroserviceIntegrationTest {

    @Container
    private static final PostgreSQLContainer<?> postgresContainer = new
PostgreSQLContainer<>("postgres:14.1")
            .withDatabaseName("testdb")
            .withUsername("test")
            .withPassword("test");

    @Container
    private static final RabbitMQContainer rabbitMQContainer = new
RabbitMQContainer("rabbitmq:3.9");
```

```
    @Container
    private static final RedisContainer redisContainer = new
RedisContainer("redis:6.2");

    @DynamicPropertySource
    static void configureProperties(DynamicPropertyRegistry registry) {
        // PostgreSQL properties
        registry.add("spring.datasource.url", postgresContainer::getJdbcUrl);
        registry.add("spring.datasource.username",
postgresContainer::getUsername);
        registry.add("spring.datasource.password",
postgresContainer::getPassword);

        // RabbitMQ properties
        registry.add("spring.rabbitmq.host", rabbitMQContainer::getHost);
        registry.add("spring.rabbitmq.port", rabbitMQContainer::getAmqpPort);
        registry.add("spring.rabbitmq.username",
rabbitMQContainer::getAdminUsername);
        registry.add("spring.rabbitmq.password",
rabbitMQContainer::getAdminPassword);

        // Redis properties
        registry.add("spring.redis.host", redisContainer::getHost);
        registry.add("spring.redis.port", redisContainer::getFirstMappedPort);
    }

    @Autowired
    private OrderService orderService;

    @Autowired
    private OrderRepository orderRepository;

    @Autowired
    private ProductRepository productRepository;

    @Autowired
    private RabbitTemplate rabbitTemplate;

    @Autowired
    private RedisTemplate<String, Object> redisTemplate;

    @BeforeEach
    void setUp() {
        orderRepository.deleteAll();
        productRepository.deleteAll();
        redisTemplate.getConnectionFactory().getConnection().flushAll();
    }

    @Test
    void createOrder_ProcessesThroughMicroservices() throws Exception {
        // Create test data
        Product product = new Product();
        product.setName("Test Product");
        product.setPrice(new BigDecimal("99.99"));
```

```
        product.setStockQuantity(10);
        product = productRepository.save(product);

        // Create order request
        OrderRequest orderRequest = new OrderRequest();
        orderRequest.setCustomerId("customer123");
        orderRequest.setProductId(product.getId());
        orderRequest.setQuantity(2);

        // Place order
        OrderResponse orderResponse = orderService.createOrder(orderRequest);

        // Verify order is created
        assertNotNull(orderResponse.getOrderId());
        assertEquals("CREATED", orderResponse.getStatus());

        // Verify order event is published to RabbitMQ
        CountDownLatch latch = new CountDownLatch(1);
        AtomicReference<OrderCreatedEvent> receivedEvent = new AtomicReference<>
();

        // Register a message listener to capture the event
        rabbitTemplate.setMessageConverter(new Jackson2JsonMessageConverter());
        SimpleMessageListenerContainer container = new
SimpleMessageListenerContainer(rabbitTemplate.getConnectionFactory());
        container.setQueueNames("order-events");
        container.setMessageListener(message -> {
            try {
                String json = new String(message.getBody());
                OrderCreatedEvent event = new ObjectMapper().readValue(json,
OrderCreatedEvent.class);
                receivedEvent.set(event);
                latch.countDown();
            } catch (Exception e) {
                e.printStackTrace();
            }
        });
        container.start();

        // Wait for the event to be received
        boolean messageReceived = latch.await(5, TimeUnit.SECONDS);
        container.stop();

        assertTrue(messageReceived, "Order event was not received");
        assertEquals(orderResponse.getOrderId(),
receivedEvent.get().getOrderId());
        assertEquals("customer123", receivedEvent.get().getCustomerId());

        // Verify inventory is updated
        Product updatedProduct =
productRepository.findById(product.getId()).orElseThrow();
        assertEquals(8, updatedProduct.getStockQuantity());

        // Verify order data is cached in Redis
```

```
        String cacheKey = "order:" + orderResponse.getOrderId();
        assertTrue(redisTemplate.hasKey(cacheKey));
    }
}
```

Custom TestContainers

Creating custom containers for specific needs:

```
public class KeycloakContainer extends GenericContainer<KeycloakContainer> {

    private static final String KEYCLOAK_IMAGE = "jboss/keycloak:16.1.1";
    private static final int KEYCLOAK_PORT = 8080;

    private String realm = "master";
    private String username = "admin";
    private String password = "admin";

    public KeycloakContainer() {
        super(KEYCLOAK_IMAGE);

        withExposedPorts(KEYCLOAK_PORT);
        withEnv("KEYCLOAK_USER", username);
        withEnv("KEYCLOAK_PASSWORD", password);
        withEnv("DB_VENDOR", "h2");
    }

    public KeycloakContainer withRealm(String realm) {
        this.realm = realm;
        return this;
    }

    public String getAuthServerUrl() {
        return String.format("http://%s:%d/auth", getHost(),
getMappedPort(KEYCLOAK_PORT));
    }

    public String getRealm() {
        return realm;
    }

    public String getUsername() {
        return username;
    }

    public String getPassword() {
        return password;
    }

    @Override
    protected void configure() {
```

```
            waitingFor(Wait.forHttp("/auth"));
    }
}

@SpringBootTest
@Testcontainers
public class SecurityIntegrationTest {

    @Container
    private static final KeycloakContainer keycloakContainer = new
KeycloakContainer()
            .withRealm("spring-boot-test");

    @DynamicPropertySource
    static void keycloakProperties(DynamicPropertyRegistry registry) {
        registry.add("spring.security.oauth2.resourceserver.jwt.issuer-uri",
                () -> keycloakContainer.getAuthServerUrl() + "/realms/" +
keycloakContainer.getRealm());
    }

    @Autowired
    private TestRestTemplate restTemplate;

    @Test
    void accessSecuredEndpoint_WithValidToken_Succeeds() {
        // Get token from Keycloak
        String token = getAccessToken(
                keycloakContainer.getAuthServerUrl(),
                keycloakContainer.getRealm(),
                keycloakContainer.getUsername(),
                keycloakContainer.getPassword()
        );

        // Set up request with token
        HttpHeaders headers = new HttpHeaders();
        headers.setBearerAuth(token);
        HttpEntity<Void> requestEntity = new HttpEntity<>(headers);

        // Call secured endpoint
        ResponseEntity<String> response = restTemplate.exchange(
                "/api/secured", HttpMethod.GET, requestEntity, String.class);

        // Verify response
        assertEquals(HttpStatus.OK, response.getStatusCode());
    }

    private String getAccessToken(String authServerUrl, String realm, String
username, String password) {
        // Implementation to get access token from Keycloak
        // ...
        return "dummy-token";
    }
}
```

Performance Testing

Evaluating the performance characteristics of your Spring Boot application.

Load Testing with JMeter

Using JMeter for load testing:

```
@SpringBootTest(webEnvironment = SpringBootTest.WebEnvironment.RANDOM_PORT)
public class JMeterIntegrationTest {

    @LocalServerPort
    private int port;

    @Autowired
    private ProductRepository productRepository;

    @Autowired
    private CategoryRepository categoryRepository;

    @BeforeEach
    void setUp() {
        // Create test data
        createTestProducts(100);
    }

    @Test
    void generateJMeterTestPlan() throws IOException {
        // Create a JMeter test plan programmatically
        TestPlan testPlan = new TestPlan("Spring Boot API Test Plan");

        // Thread group - simulates users
        ThreadGroup threadGroup = new ThreadGroup();
        threadGroup.setName("API Users");
        threadGroup.setNumThreads(50);  // 50 concurrent users
        threadGroup.setRampUp(10);      // Ramp up over 10 seconds
        threadGroup.setDuration(60);    // Run for 60 seconds
        threadGroup.setSamplerController(testPlan);

        // HTTP Request for GET /api/products
        HTTPSamplerProxy getProductsSampler = new HTTPSamplerProxy();
        getProductsSampler.setProtocol("http");
        getProductsSampler.setDomain("localhost");
        getProductsSampler.setPort(port);
        getProductsSampler.setPath("/api/products");
        getProductsSampler.setMethod("GET");
        getProductsSampler.setName("Get All Products");
        threadGroup.addTestElement(getProductsSampler);

        // HTTP Request for GET /api/products/{id}
        HTTPSamplerProxy getProductByIdSampler = new HTTPSamplerProxy();
```

```
        getProductByIdSampler.setProtocol("http");
        getProductByIdSampler.setDomain("localhost");
        getProductByIdSampler.setPort(port);
        getProductByIdSampler.setPath("/api/products/1");
        getProductByIdSampler.setMethod("GET");
        getProductByIdSampler.setName("Get Product by ID");
        threadGroup.addTestElement(getProductByIdSampler);

        // HTTP Request for POST /api/products
        HTTPSamplerProxy createProductSampler = new HTTPSamplerProxy();
        createProductSampler.setProtocol("http");
        createProductSampler.setDomain("localhost");
        createProductSampler.setPort(port);
        createProductSampler.setPath("/api/products");
        createProductSampler.setMethod("POST");
        createProductSampler.setName("Create Product");

        // Add JSON payload
        String jsonPayload = "{"
                + "\"name\": \"Test Product\","
                + "\"description\": \"Created during load test\","
                + "\"price\": 99.99,"
                + "\"categoryId\": 1"
                + "}";
        createProductSampler.addNonEncodedArgument("", jsonPayload, "");
        createProductSampler.setPostBodyRaw(true);

        // Add content type header
        HeaderManager headerManager = new HeaderManager();
        headerManager.add(new Header("Content-Type", "application/json"));
        createProductSampler.setHeaderManager(headerManager);

        threadGroup.addTestElement(createProductSampler);

        // Add results
        ResultCollector resultCollector = new ResultCollector();
        resultCollector.setFilename("jmeter-results.jtl");
        testPlan.addTestElement(resultCollector);

        // Save the test plan to a file
        SaveService.saveTree(testPlan, new FileOutputStream("spring-boot-api-test-
plan.jmx"));
    }

    private void createTestProducts(int count) {
        // Create a category
        Category category = new Category();
        category.setName("Test Category");
        category = categoryRepository.save(category);

        // Create products
        List<Product> products = new ArrayList<>();
        for (int i = 1; i <= count; i++) {
            Product product = new Product();
```

```
            product.setName("Product " + i);
            product.setDescription("Description " + i);
            product.setPrice(new BigDecimal(String.format("%.2f", 10 + i *
0.99)));
            product.setCategory(category);
            products.add(product);
        }

        productRepository.saveAll(products);
    }
}
```

Performance Metrics with Micrometer

Measuring application performance:

```
@SpringBootTest
public class PerformanceMetricsTest {

    @Autowired
    private MeterRegistry meterRegistry;

    @Autowired
    private ProductService productService;

    @Test
    void measureServicePerformance() {
        // Create a timer
        Timer timer = meterRegistry.timer("product.service.findAll");

        // Measure operation time
        timer.record(() -> {
            List<ProductDto> products = productService.findAll();
            assertNotNull(products);
        });

        // Verify the timer has recorded the operation
        Timer.Sample sample = Timer.start(meterRegistry);
        productService.findAll();
        sample.stop(timer);

        // Print metrics
        Timer.Snapshot snapshot = timer.takeSnapshot();
        System.out.println("Count: " + snapshot.count());
        System.out.println("Total time: " + snapshot.total(TimeUnit.MILLISECONDS)
+ " ms");
        System.out.println("Max: " + snapshot.max(TimeUnit.MILLISECONDS) + " ms");
        System.out.println("Mean: " + snapshot.mean(TimeUnit.MILLISECONDS) + "
ms");
    }
```

```
    @Test
    void benchmarkDatabaseOperations() {
        int iterations = 1000;

        // Create timer for database operations
        Timer createTimer = meterRegistry.timer("db.product.create");
        Timer findTimer = meterRegistry.timer("db.product.find");
        Timer updateTimer = meterRegistry.timer("db.product.update");
        Timer deleteTimer = meterRegistry.timer("db.product.delete");

        for (int i = 0; i < iterations; i++) {
            ProductDto productDto = new ProductDto();
            productDto.setName("Benchmark Product " + i);
            productDto.setDescription("Created for benchmarking");
            productDto.setPrice(new BigDecimal("99.99"));
            productDto.setCategoryId(1L);

            // Measure create operation
            ProductDto createdProduct = createTimer.record(() ->
productService.createProduct(productDto));

            // Measure find operation
            ProductDto foundProduct = findTimer.record(() ->
productService.findById(createdProduct.getId()));

            // Measure update operation
            foundProduct.setName("Updated Product " + i);
            updateTimer.record(() ->
productService.updateProduct(foundProduct.getId(), foundProduct));

            // Measure delete operation
            deleteTimer.record(() ->
productService.deleteProduct(foundProduct.getId()));
        }

        // Print metrics for each operation
        System.out.println("Create - Avg: " +
createTimer.mean(TimeUnit.MILLISECONDS) + " ms");
        System.out.println("Find - Avg: " + findTimer.mean(TimeUnit.MILLISECONDS)
+ " ms");
        System.out.println("Update - Avg: " +
updateTimer.mean(TimeUnit.MILLISECONDS) + " ms");
        System.out.println("Delete - Avg: " +
deleteTimer.mean(TimeUnit.MILLISECONDS) + " ms");
    }
}
```

Memory Usage Testing

Monitoring memory usage:

```
@SpringBootTest
public class MemoryUsageTest {

    @Autowired
    private ProductService productService;

    @Autowired
    private ProductRepository productRepository;

    @Test
    void monitorMemoryUsage() throws Exception {
        // Get memory MXBean
        MemoryMXBean memoryMXBean = ManagementFactory.getMemoryMXBean();

        // Print initial memory usage
        printMemoryUsage("Initial", memoryMXBean);

        // Create a large number of products
        int productCount = 10000;
        createTestProducts(productCount);

        // Print memory usage after creation
        printMemoryUsage("After creation", memoryMXBean);

        // Force garbage collection
        System.gc();

        // Print memory usage after GC
        printMemoryUsage("After GC", memoryMXBean);

        // Fetch all products (may consume significant memory)
        List<ProductDto> products = productService.findAll();
        assertEquals(productCount, products.size());

        // Print memory usage after fetching
        printMemoryUsage("After fetching all", memoryMXBean);
    }

    private void printMemoryUsage(String stage, MemoryMXBean memoryMXBean) {
        System.out.println("=== Memory Usage [" + stage + "] ===");

        MemoryUsage heapUsage = memoryMXBean.getHeapMemoryUsage();
        MemoryUsage nonHeapUsage = memoryMXBean.getNonHeapMemoryUsage();

        System.out.println("Heap Memory: used=" + (heapUsage.getUsed() / (1024 *
1024)) + "MB, "
                + "committed=" + (heapUsage.getCommitted() / (1024 * 1024)) + "MB,
"
                + "max=" + (heapUsage.getMax() / (1024 * 1024)) + "MB");

        System.out.println("Non-Heap Memory: used=" + (nonHeapUsage.getUsed() /
(1024 * 1024)) + "MB, "
                + "committed=" + (nonHeapUsage.getCommitted() / (1024 * 1024)) +
```

```
"MB, "
                + "max=" + (nonHeapUsage.getMax() / (1024 * 1024)) + "MB");

        System.out.println();
    }

    private void createTestProducts(int count) {
        // Create a category
        Category category = new Category();
        category.setName("Memory Test Category");
        category = categoryRepository.save(category);

        // Create products in batches to avoid excessive memory usage
        int batchSize = 1000;
        for (int i = 0; i < count; i += batchSize) {
            List<Product> batch = new ArrayList<>(batchSize);

            for (int j = 0; j < batchSize && i + j < count; j++) {
                Product product = new Product();
                product.setName("Memory Test Product " + (i + j));
                product.setDescription("Description with some data to consume
memory. ".repeat(10));
                product.setPrice(new BigDecimal(String.format("%.2f", 10 + (i + j)
* 0.99)));
                product.setCategory(category);
                batch.add(product);
            }

            productRepository.saveAll(batch);

            // Clear persistence context to free memory
            entityManager.clear();
        }
    }
}
```

Concurrency Testing

Testing concurrent operations:

```
@SpringBootTest(webEnvironment = SpringBootTest.WebEnvironment.RANDOM_PORT)
public class ConcurrencyTest {

    @Autowired
    private TestRestTemplate restTemplate;

    @Autowired
    private ProductRepository productRepository;

    @Autowired
    private CategoryRepository categoryRepository;
```

```
    @BeforeEach
    void setUp() {
        productRepository.deleteAll();
        categoryRepository.deleteAll();

        // Create a category
        Category category = new Category();
        category.setName("Concurrency Test");
        categoryRepository.save(category);
    }

    @Test
    void concurrentProductCreation() throws Exception {
        int threadCount = 20;
        CountDownLatch startLatch = new CountDownLatch(1);
        CountDownLatch endLatch = new CountDownLatch(threadCount);
        AtomicInteger successCount = new AtomicInteger(0);
        AtomicInteger failureCount = new AtomicInteger(0);

        // Create threads
        for (int i = 0; i < threadCount; i++) {
            int threadNumber = i;
            new Thread(() -> {
                try {
                    // Wait for all threads to be ready
                    startLatch.await();

                    // Create product
                    ProductDto productDto = new ProductDto();
                    productDto.setName("Concurrent Product " + threadNumber);
                    productDto.setDescription("Created by thread " +
threadNumber);
                    productDto.setPrice(new BigDecimal("99.99"));
                    productDto.setCategoryId(1L);

                    // Send request
                    ResponseEntity<ProductDto> response =
restTemplate.postForEntity(
                            "/api/products", productDto, ProductDto.class);

                    // Check result
                    if (response.getStatusCode() == HttpStatus.CREATED) {
                        successCount.incrementAndGet();
                    } else {
                        failureCount.incrementAndGet();
                    }
                } catch (Exception e) {
                    failureCount.incrementAndGet();
                } finally {
                    // Signal thread completion
                    endLatch.countDown();
                }
            }).start();
```

```
        }

        // Start all threads simultaneously
        startLatch.countDown();

        // Wait for all threads to complete
        boolean completed = endLatch.await(30, TimeUnit.SECONDS);
        assertTrue(completed, "Not all threads completed in time");

        // Verify results
        assertEquals(threadCount, successCount.get() + failureCount.get());
        assertEquals(threadCount, successCount.get(), "Some concurrent requests
failed");
        assertEquals(threadCount, productRepository.count(), "Not all products
were created");
    }

    @Test
    void concurrentUpdatesToSameProduct() throws Exception {
        // Create a product
        Product product = new Product();
        product.setName("Original Product");
        product.setDescription("Original Description");
        product.setPrice(new BigDecimal("100.00"));

        Category category = categoryRepository.findById(1L).orElseThrow();
        product.setCategory(category);
        product = productRepository.save(product);

        Long productId = product.getId();

        // Set up concurrent updates
        int threadCount = 10;
        CountDownLatch startLatch = new CountDownLatch(1);
        CountDownLatch endLatch = new CountDownLatch(threadCount);
        AtomicInteger successCount = new AtomicInteger(0);
        AtomicInteger failureCount = new AtomicInteger(0);

        // Create threads
        for (int i = 0; i < threadCount; i++) {
            int threadNumber = i;
            new Thread(() -> {
                try {
                    // Wait for all threads to be ready
                    startLatch.await();

                    // Get product first
                    ResponseEntity<ProductDto> getResponse =
restTemplate.getForEntity(
                            "/api/products/{id}", ProductDto.class, productId);

                    ProductDto productDto = getResponse.getBody();
                    assertNotNull(productDto);
```

```
                    // Update product
                    productDto.setName("Updated by thread " + threadNumber);
                    productDto.setPrice(new BigDecimal("10" + threadNumber +
".99"));

                    // Send update request
                    HttpHeaders headers = new HttpHeaders();
                    headers.setContentType(MediaType.APPLICATION_JSON);
                    HttpEntity<ProductDto> requestEntity = new HttpEntity<>
(productDto, headers);

                    ResponseEntity<ProductDto> updateResponse =
restTemplate.exchange(
                            "/api/products/{id}", HttpMethod.PUT, requestEntity,
ProductDto.class, productId);

                    // Check result
                    if (updateResponse.getStatusCode() == HttpStatus.OK) {
                        successCount.incrementAndGet();
                    } else {
                        failureCount.incrementAndGet();
                    }
                } catch (Exception e) {
                    failureCount.incrementAndGet();
                } finally {
                    // Signal thread completion
                    endLatch.countDown();
                }
            }).start();
        }

        // Start all threads simultaneously
        startLatch.countDown();

        // Wait for all threads to complete
        boolean completed = endLatch.await(30, TimeUnit.SECONDS);
        assertTrue(completed, "Not all threads completed in time");

        // With optimistic locking, not all concurrent updates should succeed
        assertTrue(successCount.get() > 0, "No updates succeeded");
        assertTrue(failureCount.get() > 0, "All updates succeeded, which is
unexpected with optimistic locking");
        assertEquals(threadCount, successCount.get() + failureCount.get());

        // Verify final product state
        Product updatedProduct =
productRepository.findById(productId).orElseThrow();
        assertTrue(updatedProduct.getName().startsWith("Updated by thread"),
                "Product name was not updated: " + updatedProduct.getName());
    }
}
```

Best Practices for Testing

Following best practices ensures effective and maintainable tests.

Test Naming and Organization

```
// Use descriptive class names
public class ProductServiceCreateOperationsTest {

    // Use descriptive method names with the pattern:
methodName_condition_expectedResult
    @Test
    void createProduct_ValidData_ReturnsProduct() {
        // Test implementation
    }

    @Test
    void createProduct_InvalidCategory_ThrowsException() {
        // Test implementation
    }
}

// Organize by functionality
public class ProductServiceSearchOperationsTest {

    @Test
    void searchProducts_ByKeyword_ReturnsMatchingProducts() {
        // Test implementation
    }

    @Test
    void searchProducts_ByPriceRange_ReturnsProductsInRange() {
        // Test implementation
    }
}

// Organize by layer
public class ProductControllerTest {
    // Controller tests
}

public class ProductServiceTest {
    // Service tests
}

public class ProductRepositoryTest {
    // Repository tests
}

// Organize by test type
public class ProductUnitTest {
    // Unit tests
```

```
}

public class ProductIntegrationTest {
    // Integration tests
}

public class ProductE2ETest {
    // End-to-end tests
}
```

Test Fixture Setup

```
public class ProductTestFixtures {

    // Factory methods to create test objects
    public static Product createSampleProduct(Long id, String name) {
        Product product = new Product();
        product.setId(id);
        product.setName(name);
        product.setDescription("Sample product description");
        product.setPrice(new BigDecimal("99.99"));
        product.setCategory(createSampleCategory(1L, "Sample Category"));
        return product;
    }

    public static Category createSampleCategory(Long id, String name) {
        Category category = new Category();
        category.setId(id);
        category.setName(name);
        return category;
    }

    public static ProductDto createSampleProductDto(Long id, String name) {
        ProductDto dto = new ProductDto();
        dto.setId(id);
        dto.setName(name);
        dto.setDescription("Sample product description");
        dto.setPrice(new BigDecimal("99.99"));
        dto.setCategoryId(1L);
        return dto;
    }

    public static List<Product> createSampleProductList(int count) {
        List<Product> products = new ArrayList<>();
        for (int i = 1; i <= count; i++) {
            products.add(createSampleProduct((long) i, "Product " + i));
        }
        return products;
    }
}
```

```
public class ProductServiceTest {

    @Test
    void findById_ExistingProduct_ReturnsProduct() {
        // Use fixture to create test data
        Product product = ProductTestFixtures.createSampleProduct(1L, "Test
Product");

        // Set up mock
        when(productRepository.findById(1L)).thenReturn(Optional.of(product));

        // Execute test and verify
        ProductDto result = productService.findById(1L);
        assertEquals(1L, result.getId());
        assertEquals("Test Product", result.getName());
    }
}
```

Using Assertions Effectively

```
@Test
void createProduct_ValidData_ReturnsProduct() {
    // Arrange
    ProductDto productDto = ProductTestFixtures.createSampleProductDto(null, "New
Product");
    Product savedProduct = ProductTestFixtures.createSampleProduct(1L, "New
Product");


when(categoryRepository.findById(1L)).thenReturn(Optional.of(savedProduct.getCateg
ory()));
    when(productRepository.save(any(Product.class))).thenReturn(savedProduct);

    // Act
    ProductDto result = productService.createProduct(productDto);

    // Assert - Basic assertions
    assertNotNull(result);
    assertEquals(1L, result.getId());
    assertEquals("New Product", result.getName());
    assertEquals(0, new BigDecimal("99.99").compareTo(result.getPrice()));

    // Assert - Grouped assertions (all are executed even if one fails)
    assertAll("Product properties",
            () -> assertEquals(1L, result.getId()),
            () -> assertEquals("New Product", result.getName()),
            () -> assertEquals("Sample product description",
result.getDescription()),
            () -> assertEquals(0, new
BigDecimal("99.99").compareTo(result.getPrice())),
            () -> assertEquals(1L, result.getCategoryId())
```

```
    );

    // Assert - Collection assertions
    List<ProductDto> products = List.of(result);
    assertThat(products)
            .hasSize(1)
            .extracting(ProductDto::getName)
            .containsExactly("New Product");

    // Assert - Complex object assertions
    assertThat(result)
            .isNotNull()
            .hasFieldOrPropertyWithValue("id", 1L)
            .hasFieldOrPropertyWithValue("name", "New Product")
            .satisfies(dto -> {
                assertEquals(0, new
BigDecimal("99.99").compareTo(dto.getPrice()));
                assertNotNull(dto.getDescription());
            });

    // Assert - Exception assertions
    assertThrows(ResourceNotFoundException.class, () -> {
        productService.findById(99L);
    });
}
```

Testing Asynchronous Code

```
@Test
void processOrderAsync_CompletesSuccessfully() throws Exception {
    // Arrange
    OrderRequest orderRequest = new OrderRequest();
    orderRequest.setCustomerId("customer123");
    orderRequest.setProductId(1L);
    orderRequest.setQuantity(2);

    Product product = ProductTestFixtures.createSampleProduct(1L, "Test Product");
    product.setStockQuantity(10);

    when(productRepository.findById(1L)).thenReturn(Optional.of(product));

    // Act - Call async method
    CompletableFuture<OrderResult> future =
orderService.processOrderAsync(orderRequest);

    // Assert - Wait for completion and check result
    OrderResult result = future.get(5, TimeUnit.SECONDS);

    assertNotNull(result);
    assertEquals("SUCCESS", result.getStatus());
```

```
    // Verify product stock was updated
    verify(productRepository).save(any(Product.class));
}

@Test
void processOrderAsync_WithTimeout() {
    // Arrange
    OrderRequest orderRequest = new OrderRequest();
    orderRequest.setCustomerId("customer123");
    orderRequest.setProductId(1L);
    orderRequest.setQuantity(2);

    // Simulate slow response
    when(productRepository.findById(1L)).thenAnswer(invocation -> {
        Thread.sleep(1000); // Delay
        return Optional.of(ProductTestFixtures.createSampleProduct(1L, "Test
Product"));
    });

    // Act & Assert - Check that it times out
    assertTimeoutPreemptively(Duration.ofMillis(500), () -> {
        try {
            orderService.processOrderAsync(orderRequest).get();
            fail("Should have timed out");
        } catch (TimeoutException e) {
            // Expected
        }
    });
}
```

Test Parameterization

```
@ParameterizedTest
@ValueSource(strings = {"smartphone", "smart watch", "SMART tv"})
void searchProducts_MatchesKeywordIgnoringCase(String keyword) {
    // Arrange
    List<Product> products = List.of(
            ProductTestFixtures.createSampleProduct(1L, "Smartphone"),
            ProductTestFixtures.createSampleProduct(2L, "Smart Watch"),
            ProductTestFixtures.createSampleProduct(3L, "Smart TV")
    );


when(productRepository.findByNameContainingIgnoreCase(any())).thenReturn(products)
;

    // Act
    List<ProductDto> results = productService.searchProductsByKeyword(keyword);

    // Assert
    assertEquals(3, results.size());
```

```
verify(productRepository).findByNameContainingIgnoreCase(keyword.toLowerCase());
}

@ParameterizedTest
@CsvSource({
    "1, 100, 2",
    "50, 150, 1",
    "200, 300, 0"
})
void searchByPriceRange_ReturnsMatchingProducts(BigDecimal min, BigDecimal max,
int expectedCount) {
    // Arrange
    List<Product> products = List.of(
            createProductWithPrice("Budget Product", new BigDecimal("99.99")),
            createProductWithPrice("Mid-range Product", new BigDecimal("149.99")),
            createProductWithPrice("Premium Product", new BigDecimal("299.99"))
    );

    when(productRepository.findByPriceBetween(min, max)).thenReturn(
            products.stream()
                    .filter(p -> p.getPrice().compareTo(min) >= 0 &&
p.getPrice().compareTo(max) <= 0)
                    .collect(Collectors.toList())
    );

    // Act
    List<ProductDto> results = productService.searchProductsByPriceRange(min,
max);

    // Assert
    assertEquals(expectedCount, results.size());
    verify(productRepository).findByPriceBetween(min, max);
}

@ParameterizedTest
@MethodSource("validationCases")
void validateProduct_ReturnsValidationResults(ProductDto productDto, boolean
expected, String fieldWithError) {
    // Act
    ValidationResult result = productValidator.validate(productDto);

    // Assert
    assertEquals(expected, result.isValid());

    if (!expected) {
        assertTrue(result.getErrors().containsKey(fieldWithError));
    }
}

static Stream<Arguments> validationCases() {
    return Stream.of(
            // Valid product
            Arguments.of(
```

```
                    ProductTestFixtures.createSampleProductDto(null, "Valid
Product"),
                    true,
                    null
            ),
            // Missing name
            Arguments.of(
                    createInvalidProductDto(null, "", "Description", new
BigDecimal("99.99"), 1L),
                    false,
                    "name"
            ),
            // Negative price
            Arguments.of(
                    createInvalidProductDto(null, "Product", "Description", new
BigDecimal("-1.00"), 1L),
                    false,
                    "price"
            ),
            // Missing category
            Arguments.of(
                    createInvalidProductDto(null, "Product", "Description", new
BigDecimal("99.99"), null),
                    false,
                    "categoryId"
            )
    );
}

private static ProductDto createInvalidProductDto(Long id, String name, String
description,
                                                   BigDecimal price, Long categoryId)
{
    ProductDto dto = new ProductDto();
    dto.setId(id);
    dto.setName(name);
    dto.setDescription(description);
    dto.setPrice(price);
    dto.setCategoryId(categoryId);
    return dto;
}
```

Clean Test Data

```
@SpringBootTest
public class ProductServiceIntegrationTest {

    @Autowired
    private ProductService productService;

    @Autowired
```

```
    private ProductRepository productRepository;

    @Autowired
    private CategoryRepository categoryRepository;

    private Long testCategoryId;
    private final List<Long> createdProductIds = new ArrayList<>();

    @BeforeEach
    void setUp() {
        // Create a test category
        Category category = new Category();
        category.setName("Test Category " + UUID.randomUUID());
        category = categoryRepository.save(category);
        testCategoryId = category.getId();
    }

    @AfterEach
    void tearDown() {
        // Clean up test data
        productRepository.deleteAllById(createdProductIds);
        createdProductIds.clear();

        if (testCategoryId != null) {
            categoryRepository.deleteById(testCategoryId);
            testCategoryId = null;
        }
    }

    @Test
    void createAndFindProduct_Success() {
        // Arrange
        ProductDto productDto = new ProductDto();
        productDto.setName("Integration Test Product");
        productDto.setDescription("Test Description");
        productDto.setPrice(new BigDecimal("99.99"));
        productDto.setCategoryId(testCategoryId);

        // Act
        ProductDto createdProductDto = productService.createProduct(productDto);

        // Track created product for cleanup
        createdProductIds.add(createdProductDto.getId());

        // Find the product
        ProductDto foundProductDto =
productService.findById(createdProductDto.getId());

        // Assert
        assertNotNull(foundProductDto);
        assertEquals(createdProductDto.getId(), foundProductDto.getId());
        assertEquals("Integration Test Product", foundProductDto.getName());
    }
}
```

G. Spring Boot in Production

Application Packaging and Deployment

Preparing and deploying Spring Boot applications to production environments.

Building Executable JARs

```
<build>
    <plugins>
        <plugin>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-maven-plugin</artifactId>
            <configuration>
                <excludes>
                    <exclude>
                        <groupId>org.projectlombok</groupId>
                        <artifactId>lombok</artifactId>
                    </exclude>
                </excludes>
                <layers>
                    <enabled>true</enabled>
                </layers>
            </configuration>
        </plugin>
    </plugins>
</build>
```

Executing the build:

```
# Maven build
mvn clean package

# Run the executable JAR
java -jar target/myapp-0.0.1-SNAPSHOT.jar
```

Using Profiles for Environment-Specific Configuration

```
# application.properties (common configuration)
spring.application.name=product-service
```

```
# application-dev.properties
server.port=8080
```

```
spring.datasource.url=jdbc:h2:mem:devdb
spring.jpa.hibernate.ddl-auto=create-drop
```

```
# application-test.properties
server.port=8081
spring.datasource.url=jdbc:h2:mem:testdb
spring.jpa.hibernate.ddl-auto=create
```

```
# application-prod.properties
server.port=80
spring.datasource.url=jdbc:mysql://productdb.example.com:3306/product_db
spring.datasource.username=${DB_USERNAME}
spring.datasource.password=${DB_PASSWORD}
spring.jpa.hibernate.ddl-auto=validate
```

Running with a specific profile:

```
java -jar myapp.jar --spring.profiles.active=prod
```

External Configuration

Using external configuration sources:

```
# Command line arguments
java -jar myapp.jar --server.port=8080 --spring.profiles.active=prod

# Environment variables
export SPRING_DATASOURCE_URL=jdbc:mysql://productdb.example.com:3306/product_db
export SPRING_DATASOURCE_USERNAME=dbuser
export SPRING_DATASOURCE_PASSWORD=dbpass
java -jar myapp.jar

# External properties file
java -jar myapp.jar --
spring.config.location=file:/etc/myapp/application.properties

# Cloud platform environment variables
# Automatically detected by Spring Boot
```

Traditional WAR Deployment

For deployment to external servlet containers:

```
<packaging>war</packaging>

<dependencies>
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-web</artifactId>
    </dependency>
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-tomcat</artifactId>
        <scope>provided</scope>
    </dependency>
</dependencies>
```

```
@SpringBootApplication
public class Application extends SpringBootServletInitializer {

    @Override
    protected SpringApplicationBuilder configure(SpringApplicationBuilder
application) {
        return application.sources(Application.class);
    }

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

Layered JARs

Creating efficient layered JARs for containerization:

```
<build>
    <plugins>
        <plugin>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-maven-plugin</artifactId>
            <configuration>
                <layers>
                    <enabled>true</enabled>
                </layers>
            </configuration>
        </plugin>
    </plugins>
</build>
```

Extract and view layers:

```
# Extract layers
java -Djarmode=layertools -jar myapp.jar extract

# This creates directories:
# - dependencies (rarely changes)
# - spring-boot-loader (rarely changes)
# - snapshot-dependencies (changes during development)
# - application (changes frequently)
```

Buildpacks

Using Cloud Native Buildpacks with Spring Boot:

```
# Build image with Buildpacks
./mvnw spring-boot:build-image -Dspring-boot.build-image.imageName=myorg/myapp

# Or with Maven plugin configuration
```

```
<build>
    <plugins>
        <plugin>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-maven-plugin</artifactId>
            <configuration>
                <image>
                    <name>myorg/myapp:${project.version}</name>
                    <env>
                        <BP_JVM_VERSION>17</BP_JVM_VERSION>
                    </env>
                </image>
            </configuration>
        </plugin>
    </plugins>
</build>
```

Containerization with Docker

Creating Docker images for Spring Boot applications.

Basic Dockerfile

```
FROM eclipse-temurin:17-jdk

WORKDIR /app
```

```
COPY target/myapp-0.0.1-SNAPSHOT.jar app.jar

ENTRYPOINT ["java", "-jar", "app.jar"]
```

Build and run:

```
docker build -t myorg/myapp:latest .
docker run -p 8080:8080 myorg/myapp:latest
```

Optimized Dockerfile

```
# Build stage
FROM eclipse-temurin:17-jdk AS build
WORKDIR /workspace/app

COPY mvnw .
COPY .mvn .mvn
COPY pom.xml .
COPY src src

RUN ./mvnw package -DskipTests
RUN mkdir -p target/extracted && java -Djarmode=layertools -jar target/*.jar
extract --destination target/extracted

# Run stage
FROM eclipse-temurin:17-jre
VOLUME /tmp
ARG EXTRACTED=/workspace/app/target/extracted

COPY --from=build ${EXTRACTED}/dependencies/ ./
COPY --from=build ${EXTRACTED}/spring-boot-loader/ ./
COPY --from=build ${EXTRACTED}/snapshot-dependencies/ ./
COPY --from=build ${EXTRACTED}/application/ ./

ENTRYPOINT ["java", "org.springframework.boot.loader.JarLauncher"]
```

Docker Compose

Managing multi-container applications:

```
# docker-compose.yml
version: '3.8'

services:
  app:
    build: .
```

```
    ports:
      - '8080:8080'
    environment:
      - SPRING_PROFILES_ACTIVE=prod
      - SPRING_DATASOURCE_URL=jdbc:mysql://db:3306/product_db
      - SPRING_DATASOURCE_USERNAME=root
      - SPRING_DATASOURCE_PASSWORD=password
    depends_on:
      - db
    networks:
      - backend
    healthcheck:
      test: ['CMD', 'curl', '-f', 'http://localhost:8080/actuator/health']
      interval: 30s
      timeout: 10s
      retries: 3
      start_period: 40s
    deploy:
      resources:
        limits:
          cpus: '1'
          memory: 512M

  db:
    image: mysql:8.0
    environment:
      - MYSQL_ROOT_PASSWORD=password
      - MYSQL_DATABASE=product_db
    volumes:
      - db-data:/var/lib/mysql
    networks:
      - backend
    ports:
      - '3306:3306'
    healthcheck:
      test: ['CMD', 'mysqladmin', 'ping', '-h', 'localhost']
      interval: 10s
      timeout: 5s
      retries: 5

volumes:
  db-data:

networks:
  backend:
```

Start the application:

```
docker-compose up -d
```

Managing Environment Variables

Securely managing environment variables:

```
FROM eclipse-temurin:17-jre

WORKDIR /app
COPY target/myapp-0.0.1-SNAPSHOT.jar app.jar

# Specify environment variables
ENV SPRING_PROFILES_ACTIVE=prod
ENV JAVA_OPTS="-Xms512m -Xmx512m"

# Use environment variables at runtime
ENTRYPOINT ["sh", "-c", "java $JAVA_OPTS -jar app.jar"]
```

Using Docker secrets:

```
# docker-compose.yml with secrets
version: '3.8'

services:
  app:
    build: .
    ports:
      - '8080:8080'
    environment:
      - SPRING_PROFILES_ACTIVE=prod
      - SPRING_DATASOURCE_URL=jdbc:mysql://db:3306/product_db
      - SPRING_DATASOURCE_USERNAME_FILE=/run/secrets/db_username
      - SPRING_DATASOURCE_PASSWORD_FILE=/run/secrets/db_password
    secrets:
      - db_username
      - db_password
    # Other configuration...

secrets:
  db_username:
    file: ./secrets/db_username.txt
  db_password:
    file: ./secrets/db_password.txt
```

Docker Multi-Stage Builds with Gradle

```
# Build stage
FROM eclipse-temurin:17-jdk AS build
WORKDIR /workspace/app
```

```
COPY gradlew .
COPY gradle gradle
COPY build.gradle .
COPY settings.gradle .
COPY src src

RUN ./gradlew build -x test
RUN mkdir -p build/extracted && java -Djarmode=layertools -jar build/libs/*.jar
extract --destination build/extracted

# Run stage
FROM eclipse-temurin:17-jre
VOLUME /tmp
ARG EXTRACTED=/workspace/app/build/extracted

COPY --from=build ${EXTRACTED}/dependencies/ ./
COPY --from=build ${EXTRACTED}/spring-boot-loader/ ./
COPY --from=build ${EXTRACTED}/snapshot-dependencies/ ./
COPY --from=build ${EXTRACTED}/application/ ./

ENTRYPOINT ["java", "org.springframework.boot.loader.JarLauncher"]
```

Kubernetes Deployment

Deploying Spring Boot applications to Kubernetes clusters.

Kubernetes Deployment Manifest

```
# deployment.yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: product-service
  labels:
    app: product-service
spec:
  replicas: 3
  selector:
    matchLabels: app
  ports:
    - port: 80
      targetPort: 8080
      protocol: TCP
      name: http
  selector:
    app: product-service
```

Service Manifest

```yaml
# service.yaml
apiVersion: v1
kind: Service
metadata:
  name: product-service
  labels:
    app: product-service
spec:
  ports:
    - port: 80
      targetPort: 8080
      protocol: TCP
      name: http
  selector:
    app: product-service
```

ConfigMap for Application Configuration

```yaml
# configmap.yaml
apiVersion: v1
kind: ConfigMap
metadata:
  name: product-service-config
data:
  application.yaml: |
    server:
      port: 8080
    spring:
      application:
        name: product-service
      datasource:
        url: jdbc:mysql://mysql-service:3306/product_db
        driver-class-name: com.mysql.cj.jdbc.Driver
      jpa:
        hibernate:
          ddl-auto: validate
    management:
      endpoints:
        web:
          exposure:
            include: health,info,metrics,prometheus
```

Secret for Sensitive Information

```yaml
# secret.yaml
apiVersion: v1
kind: Secret
metadata:
```

```
  name: product-service-secret
type: Opaque
data:
  # Base64 encoded values
  datasource.username: cm9vdA==
  datasource.password: cGFzc3dvcmQ=
```

Full Deployment with ConfigMap and Secret

```
# deployment-with-config.yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: product-service
  labels:
    app: product-service
spec:
  replicas: 3
  selector:
    matchLabels:
      app: product-service
  template:
    metadata:
      labels:
        app: product-service
    spec:
      containers:
        - name: product-service
          image: myorg/product-service:latest
          ports:
            - containerPort: 8080
          env:
            - name: SPRING_DATASOURCE_USERNAME
              valueFrom:
                secretKeyRef:
                  name: product-service-secret
                  key: datasource.username
            - name: SPRING_DATASOURCE_PASSWORD
              valueFrom:
                secretKeyRef:
                  name: product-service-secret
                  key: datasource.password
          volumeMounts:
            - name: config-volume
              mountPath: /config
          resources:
            requests:
              memory: '256Mi'
              cpu: '200m'
            limits:
              memory: '512Mi'
```

```
            cpu: '500m'
        readinessProbe:
          httpGet:
            path: /actuator/health
            port: 8080
          initialDelaySeconds: 30
          periodSeconds: 10
        livenessProbe:
          httpGet:
            path: /actuator/health/liveness
            port: 8080
          initialDelaySeconds: 60
          periodSeconds: 15
      volumes:
        - name: config-volume
          configMap:
            name: product-service-config
```

Kubernetes for Development vs. Production

Development (Minikube):

```
# Start Minikube
minikube start

# Build and publish Docker image
eval $(minikube docker-env)
./mvnw spring-boot:build-image -Dspring-boot.build-image.imageName=myorg/product-service:latest

# Deploy to Minikube
kubectl apply -f kubernetes/dev/

# Access the service
minikube service product-service
```

Production (Cloud provider):

```
# Build and push to container registry
./mvnw spring-boot:build-image
docker tag myorg/product-service:latest registry.example.com/myorg/product-service:v1.0.0
docker push registry.example.com/myorg/product-service:v1.0.0

# Update deployment image in production YAML
# Then apply the configuration
kubectl apply -f kubernetes/prod/

# Scale deployment
```

```
kubectl scale deployment product-service --replicas=5

# Rolling update
kubectl set image deployment/product-service product-
service=registry.example.com/myorg/product-service:v1.0.1
```

Helm Chart for Spring Boot Application

```
# Chart.yaml
apiVersion: v2
name: product-service
description: A Helm chart for Product Service
type: application
version: 0.1.0
appVersion: '1.0.0'
```

```
# values.yaml
replicaCount: 3

image:
  repository: myorg/product-service
  tag: latest
  pullPolicy: IfNotPresent

service:
  type: ClusterIP
  port: 80
  targetPort: 8080

resources:
  requests:
    cpu: 200m
    memory: 256Mi
  limits:
    cpu: 500m
    memory: 512Mi

env:
  SPRING_PROFILES_ACTIVE: prod

configMap:
  data:
    application.yaml: |
      server:
        port: 8080
      spring:
        application:
          name: product-service
      management:
```

```
        endpoints:
          web:
            exposure:
              include: health,info,metrics,prometheus

secret:
  datasourceUsername: root
  datasourcePassword: password
```

```
# templates/deployment.yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: {{ include "product-service.fullname" . }}
  labels:
    {{- include "product-service.labels" . | nindent 4 }}
spec:
  replicas: {{ .Values.replicaCount }}
  selector:
    matchLabels:
      {{- include "product-service.selectorLabels" . | nindent 6 }}
  template:
    metadata:
      labels:
        {{- include "product-service.selectorLabels" . | nindent 8 }}
    spec:
      containers:
        - name: {{ .Chart.Name }}
          image: "{{ .Values.image.repository }}:{{ .Values.image.tag }}"
          imagePullPolicy: {{ .Values.image.pullPolicy }}
          ports:
            - containerPort: {{ .Values.service.targetPort }}
          env:
            {{- range $key, $value := .Values.env }}
            - name: {{ $key }}
              value: {{ $value | quote }}
            {{- end }}
            - name: SPRING_DATASOURCE_USERNAME
              valueFrom:
                secretKeyRef:
                  name: {{ include "product-service.fullname" . }}
                  key: datasource.username
            - name: SPRING_DATASOURCE_PASSWORD
              valueFrom:
                secretKeyRef:
                  name: {{ include "product-service.fullname" . }}
                  key: datasource.password
          volumeMounts:
            - name: config-volume
              mountPath: /config
          resources:
```

```
            {{- toYaml .Values.resources | nindent 12 }}
          readinessProbe:
            httpGet:
              path: /actuator/health
              port: {{ .Values.service.targetPort }}
            initialDelaySeconds: 30
            periodSeconds: 10
          livenessProbe:
            httpGet:
              path: /actuator/health/liveness
              port: {{ .Values.service.targetPort }}
            initialDelaySeconds: 60
            periodSeconds: 15
      volumes:
        - name: config-volume
          configMap:
            name: {{ include "product-service.fullname" . }}
```

Installing the Helm chart:

```
# Install
helm install product-service ./charts/product-service

# Upgrade
helm upgrade product-service ./charts/product-service
```

Metrics, Monitoring, and Observability

Setting up comprehensive monitoring for Spring Boot applications.

Spring Boot Actuator

Enabling and configuring Actuator:

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

```
# application.properties

# Enable specific endpoints
management.endpoints.web.exposure.include=health,info,metrics,prometheus,loggers

# Health details
management.endpoint.health.show-details=always
management.endpoint.health.show-components=always
```

```
# Info details
management.info.env.enabled=true
management.info.java.enabled=true
management.info.os.enabled=true
management.info.git.enabled=true

# Application info
info.app.name=Product Service
info.app.description=Spring Boot Product Management Service
info.app.version=1.0.0
```

Custom Health Indicators

Creating application-specific health indicators:

```
@Component
public class DatabaseHealthIndicator implements HealthIndicator {

    private final DataSource dataSource;

    public DatabaseHealthIndicator(DataSource dataSource) {
        this.dataSource = dataSource;
    }

    @Override
    public Health health() {
        try (Connection connection = dataSource.getConnection()) {
            PreparedStatement statement = connection.prepareStatement("SELECT 1");
            ResultSet resultSet = statement.executeQuery();

            if (resultSet.next() && resultSet.getInt(1) == 1) {
                return Health.up()
                        .withDetail("database",
connection.getMetaData().getDatabaseProductName())
                        .withDetail("version",
connection.getMetaData().getDatabaseProductVersion())
                        .build();
            } else {
                return Health.down()
                        .withDetail("error", "Database health check failed")
                        .build();
            }
        } catch (SQLException e) {
            return Health.down(e)
                    .withDetail("error", "Database connection failed: " +
e.getMessage())
                    .build();
        }
    }
}
```

Micrometer and Prometheus Integration

```
<dependency>
    <groupId>io.micrometer</groupId>
    <artifactId>micrometer-registry-prometheus</artifactId>
</dependency>
```

Custom metrics:

```
@RestController
@RequestMapping("/api/products")
public class ProductController {

    private final ProductService productService;
    private final MeterRegistry meterRegistry;

    // Create counters, timers
    private final Counter productCreationCounter;
    private final Timer productSearchTimer;

    public ProductController(ProductService productService, MeterRegistry
meterRegistry) {
        this.productService = productService;
        this.meterRegistry = meterRegistry;

        // Initialize metrics
        this.productCreationCounter = Counter.builder("api.product.creation")
                .description("Number of products created")
                .register(meterRegistry);

        this.productSearchTimer = Timer.builder("api.product.search.time")
                .description("Time spent searching products")
                .register(meterRegistry);
    }

    @PostMapping
    public ResponseEntity<ProductDto> createProduct(@Valid @RequestBody ProductDto
productDto) {
        ProductDto createdProduct = productService.createProduct(productDto);

        // Increment counter
        productCreationCounter.increment();

        // Tag-based metrics
        meterRegistry.counter("api.product.creation.by.category", "category",
                String.valueOf(productDto.getCategoryId())).increment();

        URI location = ServletUriComponentsBuilder.fromCurrentRequest()
```

```
                .path("/{id}")
                .buildAndExpand(createdProduct.getId())
                .toUri();

        return ResponseEntity.created(location).body(createdProduct);
    }

    @GetMapping("/search")
    public ResponseEntity<List<ProductDto>> searchProducts(
            @RequestParam(required = false) String keyword,
            @RequestParam(required = false) BigDecimal minPrice,
            @RequestParam(required = false) BigDecimal maxPrice) {

        // Record timer
        return productSearchTimer.record(() -> {
            List<ProductDto> products = productService.searchProducts(keyword,
minPrice, maxPrice);

            // Record size metric
            meterRegistry.summary("api.product.search.result.size")
                    .record(products.size());

            return ResponseEntity.ok(products);
        });
    }
}
```

Prometheus Configuration

```
# prometheus.yml
scrape_configs:
  - job_name: 'spring-boot-app'
    metrics_path: '/actuator/prometheus'
    scrape_interval: 15s
    static_configs:
      - targets: ['app:8080']
```

Grafana Dashboard

```
{
  "annotations": {
    "list": [
      {
        "builtIn": 1,
        "datasource": "-- Grafana --",
        "enable": true,
        "hide": true,
        "iconColor": "rgba(0, 211, 255, 1)",
```

```
        "name": "Annotations & Alerts",
        "type": "dashboard"
      }
    ]
  },
  "editable": true,
  "gnetId": null,
  "graphTooltip": 0,
  "id": 1,
  "links": [],
  "panels": [
    {
      "aliasColors": {},
      "bars": false,
      "dashLength": 10,
      "dashes": false,
      "datasource": "Prometheus",
      "fill": 1,
      "fillGradient": 0,
      "gridPos": {
        "h": 8,
        "w": 12,
        "x": 0,
        "y": 0
      },
      "hiddenSeries": false,
      "id": 2,
      "legend": {
        "avg": false,
        "current": false,
        "max": false,
        "min": false,
        "show": true,
        "total": false,
        "values": false
      },
      "lines": true,
      "linewidth": 1,
      "nullPointMode": "null",
      "options": {
        "dataLinks": []
      },
      "percentage": false,
      "pointradius": 2,
      "points": false,
      "renderer": "flot",
      "seriesOverrides": [],
      "spaceLength": 10,
      "stack": false,
      "steppedLine": false,
      "targets": [
        {
          "expr": "rate(http_server_requests_seconds_count[1m])",
          "interval": "",
```

```
            "legendFormat": "{{uri}} - {{status}}",
            "refId": "A"
          }
        ],
        "thresholds": [],
        "timeFrom": null,
        "timeRegions": [],
        "timeShift": null,
        "title": "Request Rate",
        "tooltip": {
          "shared": true,
          "sort": 0,
          "value_type": "individual"
        },
        "type": "graph",
        "xaxis": {
          "buckets": null,
          "mode": "time",
          "name": null,
          "show": true,
          "values": []
        },
        "yaxes": [
          {
            "format": "short",
            "label": null,
            "logBase": 1,
            "max": null,
            "min": null,
            "show": true
          },
          {
            "format": "short",
            "label": null,
            "logBase": 1,
            "max": null,
            "min": null,
            "show": true
          }
        ],
        "yaxis": {
          "align": false,
          "alignLevel": null
        }
      },
      {
        "aliasColors": {},
        "bars": false,
        "dashLength": 10,
        "dashes": false,
        "datasource": "Prometheus",
        "fill": 1,
        "fillGradient": 0,
        "gridPos": {
```

```
          "h": 8,
          "w": 12,
          "x": 12,
          "y": 0
        },
        "hiddenSeries": false,
        "id": 4,
        "legend": {
          "avg": false,
          "current": false,
          "max": false,
          "min": false,
          "show": true,
          "total": false,
          "values": false
        },
        "lines": true,
        "linewidth": 1,
        "nullPointMode": "null",
        "options": {
          "dataLinks": []
        },
        "percentage": false,
        "pointradius": 2,
        "points": false,
        "renderer": "flot",
        "seriesOverrides": [],
        "spaceLength": 10,
        "stack": false,
        "steppedLine": false,
        "targets": [
          {
            "expr": "rate(http_server_requests_seconds_sum[1m]) /
rate(http_server_requests_seconds_count[1m])",
            "interval": "",
            "legendFormat": "{{uri}}",
            "refId": "A"
          }
        ],
        "thresholds": [],
        "timeFrom": null,
        "timeRegions": [],
        "timeShift": null,
        "title": "Average Response Time",
        "tooltip": {
          "shared": true,
          "sort": 0,
          "value_type": "individual"
        },
        "type": "graph",
        "xaxis": {
          "buckets": null,
          "mode": "time",
          "name": null,
```

```
          "show": true,
          "values": []
        },
        "yaxes": [
          {
            "format": "s",
            "label": null,
            "logBase": 1,
            "max": null,
            "min": null,
            "show": true
          },
          {
            "format": "short",
            "label": null,
            "logBase": 1,
            "max": null,
            "min": null,
            "show": true
          }
        ],
        "yaxis": {
          "align": false,
          "alignLevel": null
        }
      }
    ],
    "schemaVersion": 22,
    "style": "dark",
    "tags": [],
    "templating": {
      "list": []
    },
    "time": {
      "from": "now-6h",
      "to": "now"
    },
    "timepicker": {
      "refresh_intervals": [
        "5s",
        "10s",
        "30s",
        "1m",
        "5m",
        "15m",
        "30m",
        "1h",
        "2h",
        "1d"
      ]
    },
    "timezone": "",
    "title": "Spring Boot Dashboard",
    "uid": "spring-boot",
```

```
  "variables": {
    "list": []
  },
  "version": 1
}
```

Distributed Tracing with Spring Cloud Sleuth and Zipkin

```
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-sleuth</artifactId>
</dependency>
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-sleuth-zipkin</artifactId>
</dependency>
```

```
# application.properties
spring.zipkin.base-url=http://zipkin:9411
spring.sleuth.sampler.probability=1.0
```

ELK Stack Integration

Logstash configuration:

```
# logstash.conf
input {
  tcp {
    port => 5000
    codec => json_lines
  }
}

filter {
  if [type] == "spring-boot" {
    grok {
      match => { "message" => "%{TIMESTAMP_ISO8601:timestamp} %{LOGLEVEL:level} %
{DATA:thread} %{DATA:class} %{GREEDYDATA:message}" }
    }
    date {
      match => [ "timestamp", "ISO8601" ]
      target => "@timestamp"
    }
  }
}
```

```
output {
  elasticsearch {
    hosts => ["elasticsearch:9200"]
    index => "spring-boot-%{+YYYY.MM.dd}"
  }
}
```

Logback configuration:

```
<configuration>
    <include resource="org/springframework/boot/logging/logback/defaults.xml"/>

    <springProperty scope="context" name="appName"
source="spring.application.name"/>

    <appender name="LOGSTASH"
class="net.logstash.logback.appender.LogstashTcpSocketAppender">
        <destination>logstash:5000</destination>
        <encoder class="net.logstash.logback.encoder.LogstashEncoder">
            <customFields>{"app":"${appName}","type":"spring-boot"}</customFields>
        </encoder>
    </appender>

    <appender name="CONSOLE" class="ch.qos.logback.core.ConsoleAppender">
        <encoder>
            <pattern>%d{HH:mm:ss.SSS} [%thread] %-5level %logger{36} -
%msg%n</pattern>
        </encoder>
    </appender>

    <root level="INFO">
        <appender-ref ref="CONSOLE"/>
        <appender-ref ref="LOGSTASH"/>
    </root>
</configuration>
```

Logging Best Practices

Implementing effective logging strategies.

Structured Logging

Using JSON format for logs:

```
<dependency>
    <groupId>net.logstash.logback</groupId>
    <artifactId>logstash-logback-encoder</artifactId>
    <version>7.1.1</version>
</dependency>
```

```
<configuration>
    <appender name="CONSOLE" class="ch.qos.logback.core.ConsoleAppender">
        <encoder class="net.logstash.logback.encoder.LogstashEncoder"/>
    </appender>

    <root level="INFO">
        <appender-ref ref="CONSOLE"/>
    </root>
</configuration>
```

Log Correlation with Distributed Tracing

```
@RestController
@RequestMapping("/api/products")
public class ProductController {

    private static final Logger logger =
LoggerFactory.getLogger(ProductController.class);

    private final ProductService productService;

    public ProductController(ProductService productService) {
        this.productService = productService;
    }

    @GetMapping("/{id}")
    public ResponseEntity<ProductDto> getProductById(@PathVariable Long id) {
        MDC.put("productId", id.toString());
        logger.info("Retrieving product with ID: {}", id);

        try {
            ProductDto product = productService.findById(id);
            logger.info("Retrieved product: {}", product.getName());
            return ResponseEntity.ok(product);
        } catch (ResourceNotFoundException e) {
            logger.warn("Product not found with ID: {}", id);
            throw e;
        } finally {
            MDC.remove("productId");
        }
    }
}
```

Log Level Management

```
# application.properties
logging.level.root=INFO
logging.level.org.springframework=INFO
logging.level.com.example.productservice=DEBUG
logging.level.org.hibernate.SQL=DEBUG
logging.level.org.hibernate.type.descriptor.sql.BasicBinder=TRACE
```

Dynamic log level management with Actuator:

```
# Change log level at runtime
curl -X POST 'http://localhost:8080/actuator/loggers/com.example.productservice' \
  -H 'Content-Type: application/json' \
  -d '{"configuredLevel": "DEBUG"}'
```

Custom Logging Aspects

```
@Aspect
@Component
public class LoggingAspect {

    private static final Logger logger =
LoggerFactory.getLogger(LoggingAspect.class);

    @Around("execution(* com.example.productservice.service.*.*(..))")
    public Object logServiceMethods(ProceedingJoinPoint joinPoint) throws
Throwable {
        MethodSignature signature = (MethodSignature) joinPoint.getSignature();
        String methodName = signature.getMethod().getName();
        String className = signature.getDeclaringType().getSimpleName();

        Object[] args = joinPoint.getArgs();

        logger.debug("Entering: {}.{}() with arguments: {}", className,
methodName, Arrays.toString(args));

        StopWatch stopWatch = new StopWatch();
        stopWatch.start();

        try {
            Object result = joinPoint.proceed();
            stopWatch.stop();

            logger.debug("Exiting: {}.{}() with result: {} in {}ms",
                    className, methodName, result,
stopWatch.getTotalTimeMillis());

            return result;
        } catch (Exception e) {
```

```
            stopWatch.stop();
            logger.error("Exception in {}.{}() with cause: {} in {}ms",
                    className, methodName, e.getMessage(),
stopWatch.getTotalTimeMillis());
            throw e;
        }
    }

    @AfterThrowing(pointcut = "execution(*
com.example.productservice.controller.*.*(..))", throwing = "exception")
    public void logControllerExceptions(JoinPoint joinPoint, Exception exception)
{
        MethodSignature signature = (MethodSignature) joinPoint.getSignature();
        String methodName = signature.getMethod().getName();
        String className = signature.getDeclaringType().getSimpleName();

        logger.error("Exception occurred in {}.{}(): {} - {}",
                className, methodName, exception.getClass().getName(),
exception.getMessage());
    }
}
```

Log Aggregation with Fluentd

Fluentd configuration:

```
<source>
  @type forward
  port 24224
  bind 0.0.0.0
</source>

<filter **>
  @type parser
  key_name log
  reserve_data true
  <parse>
    @type json
  </parse>
</filter>

<match spring.**>
  @type copy
  <store>
    @type elasticsearch
    host elasticsearch
    port 9200
    logstash_format true
    logstash_prefix spring
    logstash_dateformat %Y.%m.%d
    include_tag_key true
```

```
    tag_key @log_name
    flush_interval 5s
  </store>
</match>
```

Performance Tuning

Optimizing Spring Boot applications for better performance.

JVM Tuning

```
# Setting JVM options
java -Xms1G -Xmx1G -XX:+UseG1GC -XX:MaxGCPauseMillis=200 -jar myapp.jar
```

Common JVM options:

```
# Memory settings
-Xms512m                  # Initial heap size
-Xmx1g                    # Maximum heap size
-XX:MetaspaceSize=128m    # Initial metaspace size
-XX:MaxMetaspaceSize=256m # Maximum metaspace size

# Garbage collection
-XX:+UseG1GC                    # Use G1 garbage collector
-XX:MaxGCPauseMillis=200        # Target max pause time of 200ms
-XX:+UseStringDeduplication     # Enable string deduplication
-XX:+HeapDumpOnOutOfMemoryError # Create heap dump on OOM
-XX:HeapDumpPath=/var/log/myapp/heap-dump.hprof # Heap dump location

# Debugging
-XX:+PrintGCDetails             # Print GC details
-XX:+PrintGCDateStamps          # Print GC date stamps
-Xloggc:/var/log/myapp/gc.log   # GC log location
```

Database Connection Pool Tuning

```
# application.properties
spring.datasource.hikari.maximum-pool-size=10
spring.datasource.hikari.minimum-idle=5
spring.datasource.hikari.idle-timeout=600000
spring.datasource.hikari.max-lifetime=1800000
spring.datasource.hikari.connection-timeout=30000
spring.datasource.hikari.leak-detection-threshold=60000
```

Caching

```
@Configuration
@EnableCaching
public class CacheConfig {

    @Bean
    public CacheManager cacheManager() {
        SimpleCacheManager cacheManager = new SimpleCacheManager();

        // Configure caches with different settings
        CaffeineCache productsCache = buildCache("products", 100, 5);
        CaffeineCache categoriesCache = buildCache("categories", 20, 30);

        cacheManager.setCaches(Arrays.asList(productsCache, categoriesCache));
        return cacheManager;
    }

    private CaffeineCache buildCache(String name, int maximumSize, int
expireAfterMinutes) {
        return new CaffeineCache(name,
                Caffeine.newBuilder()
                        .maximumSize(maximumSize)
                        .expireAfterWrite(expireAfterMinutes, TimeUnit.MINUTES)
                        .recordStats()
                        .build());
    }
}
```

Service with caching:

```
@Service
public class ProductServiceImpl implements ProductService {

    private final ProductRepository productRepository;
    private final CategoryRepository categoryRepository;
    private final ProductMapper productMapper;

    public ProductServiceImpl(ProductRepository productRepository,
                              CategoryRepository categoryRepository,
                              ProductMapper productMapper) {
        this.productRepository = productRepository;
        this.categoryRepository = categoryRepository;
        this.productMapper = productMapper;
    }

    @Override
    @Cacheable(value = "products")
    public List<ProductDto> findAll() {
        return productRepository.findAll().stream()
                .map(productMapper::toDto)
                .collect(Collectors.toList());
```

```
    }

    @Override
    @Cacheable(value = "products", key = "#id")
    public ProductDto findById(Long id) {
        return productRepository.findById(id)
                .map(productMapper::toDto)
                .orElseThrow(() -> new ResourceNotFoundException("Product not
found with id: " + id));
    }

    @Override
    @CachePut(value = "products", key = "#result.id")
    @Transactional
    public ProductDto createProduct(ProductDto productDto) {
        // Implementation
    }

    @Override
    @CachePut(value = "products", key = "#id")
    @Transactional
    public ProductDto updateProduct(Long id, ProductDto productDto) {
        // Implementation
    }

    @Override
    @CacheEvict(value = "products", key = "#id")
    @Transactional
    public void deleteProduct(Long id) {
        // Implementation
    }

    @CacheEvict(value = "products", allEntries = true)
    @Scheduled(fixedRate = 86400000) // 24 hours
    public void clearProductCache() {
        // Clear cache daily
    }
}
```

Lazy Loading and Pagination

```
@RestController
@RequestMapping("/api/products")
public class ProductController {

    private final ProductService productService;

    public ProductController(ProductService productService) {
        this.productService = productService;
    }
```

```
    @GetMapping
    public ResponseEntity<Page<ProductDto>> getAllProducts(
            @RequestParam(defaultValue = "0") int page,
            @RequestParam(defaultValue = "10") int size,
            @RequestParam(defaultValue = "id,asc") String[] sort) {

        List<Sort.Order> orders = getSortOrders(sort);

        Pageable pageable = PageRequest.of(page, size, Sort.by(orders));
        Page<ProductDto> productPage = productService.findAll(pageable);

        return ResponseEntity.ok(productPage);
    }

    private List<Sort.Order> getSortOrders(String[] sort) {
        List<Sort.Order> orders = new ArrayList<>();

        if (sort[0].contains(",")) {
            // sortOrder="field,direction"
            for (String sortOrder : sort) {
                String[] _sort = sortOrder.split(",");
                orders.add(new Sort.Order(getSortDirection(_sort[1]), _sort[0]));
            }
        } else {
            // sort=[field, direction]
            orders.add(new Sort.Order(getSortDirection(sort[1]), sort[0]));
        }

        return orders;
    }

    private Sort.Direction getSortDirection(String direction) {
        if (direction.equals("desc")) {
            return Sort.Direction.DESC;
        }
        return Sort.Direction.ASC;
    }
}
```

Async Request Processing

```
@Configuration
@EnableAsync
public class AsyncConfig {

    @Bean
    public TaskExecutor taskExecutor() {
        ThreadPoolTaskExecutor executor = new ThreadPoolTaskExecutor();
        executor.setCorePoolSize(5);
        executor.setMaxPoolSize(10);
        executor.setQueueCapacity(25);
```

```
        executor.setThreadNamePrefix("MyApp-Async-");
        executor.initialize();
        return executor;
    }
}
```

```
@Service
public class ProductServiceImpl implements ProductService {

    // Other methods...

    @Async
    @Override
    public CompletableFuture<List<ProductDto>> findAllAsync() {
        List<ProductDto> products = productRepository.findAll().stream()
                .map(productMapper::toDto)
                .collect(Collectors.toList());

        return CompletableFuture.completedFuture(products);
    }

    @Async
    @Override
    public CompletableFuture<Void> refreshProductCache() {
        // Expensive operation to refresh cache
        return CompletableFuture.runAsync(() -> {
            // Implementation
        });
    }
}
```

```
@RestController
@RequestMapping("/api/products")
public class ProductController {

    // Other methods...

    @GetMapping("/async")
    public CompletableFuture<ResponseEntity<List<ProductDto>>>
getAllProductsAsync() {
        return productService.findAllAsync()
                .thenApply(ResponseEntity::ok);
    }

    @PostMapping("/refresh-cache")
    public CompletableFuture<ResponseEntity<Void>> refreshCache() {
        return productService.refreshProductCache()
                .thenApply(v -> ResponseEntity.accepted().build());
```

```
    }
}
```

Performance Testing and Profiling

```
@SpringBootTest
public class PerformanceTest {

    @Autowired
    private ProductService productService;

    @Test
    void benchmarkProductSearch() {
        // Warm up
        for (int i = 0; i < 5; i++) {
            productService.searchProducts("phone", null, null);
        }

        // Benchmark
        long startTime = System.nanoTime();

        for (int i = 0; i < 100; i++) {
            productService.searchProducts("phone", null, null);
        }

        long endTime = System.nanoTime();
        long durationMs = TimeUnit.NANOSECONDS.toMillis(endTime - startTime);

        System.out.println("100 search operations took: " + durationMs + " ms");
        System.out.println("Average time per operation: " + (durationMs / 100.0) +
" ms");
    }

    @Test
    void profileMemoryUsage() {
        MemoryMXBean memoryBean = ManagementFactory.getMemoryMXBean();

        // Before operation
        MemoryUsage heapBefore = memoryBean.getHeapMemoryUsage();
        System.out.println("Before - Heap used: " + (heapBefore.getUsed() / (1024
* 1024)) + " MB");

        // Operation to measure
        List<ProductDto> products = productService.findAll();

        // Use the result to prevent optimization
        assertNotNull(products);

        // After operation
        System.gc(); // Request garbage collection
        MemoryUsage heapAfter = memoryBean.getHeapMemoryUsage();
```

```
        System.out.println("After - Heap used: " + (heapAfter.getUsed() / (1024 *
1024)) + " MB");
    }
}
```

CI/CD Pipeline Integration

Integrating Spring Boot applications into CI/CD pipelines.

GitHub Actions Workflow

```yaml
# .github/workflows/ci-cd.yml
name: Spring Boot CI/CD

on:
  push:
    branches: [main]
  pull_request:
    branches: [main]

jobs:
  build:
    runs-on: ubuntu-latest

    steps:
      - uses: actions/checkout@v2

      - name: Set up JDK 17
        uses: actions/setup-java@v2
        with:
          java-version: '17'
          distribution: 'temurin'
          cache: maven

      - name: Build with Maven
        run: mvn -B package --file pom.xml

      - name: Run tests
        run: mvn -B test

      - name: Run integration tests
        run: mvn -B verify -P integration-test

      - name: Analyze with SonarCloud
        env:
          GITHUB_TOKEN: ${{ secrets.GITHUB_TOKEN }}
          SONAR_TOKEN: ${{ secrets.SONAR_TOKEN }}
        run: mvn -B verify org.sonarsource.scanner.maven:sonar-maven-plugin:sonar
-Dsonar.projectKey=myorg_myapp

      - name: Build and push Docker image
```

```
        if: github.event_name == 'push' && github.ref == 'refs/heads/main'
        run: |
          echo "${{ secrets.DOCKER_PASSWORD }}" | docker login -u ${{
secrets.DOCKER_USERNAME }} --password-stdin
          ./mvnw spring-boot:build-image -Dspring-boot.build-
image.imageName=myorg/myapp:latest
          docker push myorg/myapp:latest

      - name: Deploy to Kubernetes
        if: github.event_name == 'push' && github.ref == 'refs/heads/main'
        uses: stefanprodan/kube-tools@v1
        with:
          kubectl: 1.22.2
          command: |
            echo "${{ secrets.KUBE_CONFIG_DATA }}" | base64 -d > /tmp/kubeconfig
            export KUBECONFIG=/tmp/kubeconfig
            kubectl set image deployment/product-service product-
service=myorg/myapp:latest
            kubectl rollout status deployment/product-service
```

Jenkins Pipeline

```
// Jenkinsfile
pipeline {
    agent {
        docker {
            image 'maven:3.8.4-openjdk-17'
            args '-v /root/.m2:/root/.m2'
        }
    }

    environment {
        DOCKER_REGISTRY = 'registry.example.com'
        DOCKER_IMAGE = 'myorg/product-service'
    }

    stages {
        stage('Build') {
            steps {
                sh 'mvn -B -DskipTests clean package'
            }
        }

        stage('Test') {
            steps {
                sh 'mvn test'
            }
            post {
                always {
                    junit 'target/surefire-reports/*.xml'
                }
```

```
            }
        }

        stage('Integration Test') {
            steps {
                sh 'mvn verify -P integration-test'
            }
            post {
                always {
                    junit 'target/failsafe-reports/*.xml'
                }
            }
        }

        stage('Code Quality') {
            steps {
                sh 'mvn sonar:sonar'
            }
        }

        stage('Build Image') {
            when {
                branch 'main'
            }
            steps {
                sh './mvnw spring-boot:build-image -Dspring-boot.build-
image.imageName=${DOCKER_REGISTRY}/${DOCKER_IMAGE}:${BUILD_NUMBER}'
            }
        }

        stage('Push Image') {
            when {
                branch 'main'
            }
            steps {
                withCredentials([usernamePassword(credentialsId: 'docker-
registry', usernameVariable: 'DOCKER_USER', passwordVariable: 'DOCKER_PASS')]) {
                    sh '''
                        echo $DOCKER_PASS | docker login $DOCKER_REGISTRY -u
$DOCKER_USER --password-stdin
                        docker push
${DOCKER_REGISTRY}/${DOCKER_IMAGE}:${BUILD_NUMBER}
                        docker tag
${DOCKER_REGISTRY}/${DOCKER_IMAGE}:${BUILD_NUMBER}
${DOCKER_REGISTRY}/${DOCKER_IMAGE}:latest
                        docker push ${DOCKER_REGISTRY}/${DOCKER_IMAGE}:latest
                    '''
                }
            }
        }

        stage('Deploy to Dev') {
            when {
                branch 'main'
```

```
            }
            steps {
                withKubeConfig([credentialsId: 'kubeconfig']) {
                    sh '''
                        kubectl set image deployment/product-service product-
service=${DOCKER_REGISTRY}/${DOCKER_IMAGE}:${BUILD_NUMBER} -n dev
                        kubectl rollout status deployment/product-service -n dev
                    '''
                }
            }
        }

        stage('Integration Tests on Dev') {
            when {
                branch 'main'
            }
            steps {
                sh 'mvn verify -P e2e-tests'
            }
        }

        stage('Deploy to Production') {
            when {
                branch 'main'
            }
            steps {
                timeout(time: 1, unit: 'DAYS') {
                    input message: 'Approve deployment to production?'
                }

                withKubeConfig([credentialsId: 'kubeconfig-prod']) {
                    sh '''
                        kubectl set image deployment/product-service product-
service=${DOCKER_REGISTRY}/${DOCKER_IMAGE}:${BUILD_NUMBER} -n prod
                        kubectl rollout status deployment/product-service -n prod
                    '''
                }
            }
        }
    }

    post {
        always {
            cleanWs()
        }
        success {
            slackSend channel: '#deployments', color: 'good', message: "Deployment
successful: ${env.JOB_NAME} ${env.BUILD_NUMBER} (<${env.BUILD_URL}|Open>)"
        }
        failure {
            slackSend channel: '#deployments', color: 'danger', message: "Build
failed: ${env.JOB_NAME} ${env.BUILD_NUMBER} (<${env.BUILD_URL}|Open>)"
        }
```

```
    }
}
```

GitLab CI/CD

```
# .gitlab-ci.yml
image: maven:3.8.4-openjdk-17

variables:
  MAVEN_OPTS: '-Dmaven.repo.local=.m2/repository'
  DOCKER_REGISTRY: 'registry.gitlab.com'
  DOCKER_IMAGE: '$CI_REGISTRY_IMAGE'

cache:
  paths:
    - .m2/repository

stages:
  - build
  - test
  - package
  - deploy

build:
  stage: build
  script:
    - mvn compile

test:
  stage: test
  script:
    - mvn test

code_quality:
  stage: test
  script:
    - mvn verify sonar:sonar -Dsonar.projectKey=$CI_PROJECT_PATH_SLUG -
Dsonar.host.url=$SONAR_HOST_URL -Dsonar.login=$SONAR_TOKEN
  only:
    - main
    - merge_requests

integration_test:
  stage: test
  script:
    - mvn verify -P integration-test
  artifacts:
    reports:
      junit:
        - target/failsafe-reports/TEST-*.xml
```

```
  package:
    stage: package
    script:
      - mvn package -DskipTests
      - ./mvnw spring-boot:build-image -Dspring-boot.build-
image.imageName=$DOCKER_IMAGE:$CI_COMMIT_SHORT_SHA
      - docker login -u $CI_REGISTRY_USER -p $CI_REGISTRY_PASSWORD $CI_REGISTRY
      - docker push $DOCKER_IMAGE:$CI_COMMIT_SHORT_SHA
      - if [ "$CI_COMMIT_BRANCH" = "main" ]; then docker tag
$DOCKER_IMAGE:$CI_COMMIT_SHORT_SHA $DOCKER_IMAGE:latest && docker push
$DOCKER_IMAGE:latest; fi
    only:
      - main
      - tags

  deploy_dev:
    stage: deploy
    image: bitnami/kubectl:latest
    script:
      - kubectl config use-context dev
      - kubectl set image deployment/product-service product-
service=$DOCKER_IMAGE:$CI_COMMIT_SHORT_SHA -n dev
      - kubectl rollout status deployment/product-service -n dev
    environment:
      name: dev
      url: https://dev.example.com
    only:
      - main

  deploy_prod:
    stage: deploy
    image: bitnami/kubectl:latest
    script:
      - kubectl config use-context prod
      - kubectl set image deployment/product-service product-
service=$DOCKER_IMAGE:$CI_COMMIT_SHORT_SHA -n prod
      - kubectl rollout status deployment/product-service -n prod
    environment:
      name: production
      url: https://example.com
    when: manual
    only:
      - tags
```

Artifact Repository Integration

Maven `settings.xml`:

```
<settings>
  <servers>
    <server>
```

```
      <id>nexus</id>
      <username>${env.NEXUS_USER}</username>
      <password>${env.NEXUS_PASS}</password>
    </server>
  </servers>

  <mirrors>
    <mirror>
      <id>nexus</id>
      <mirrorOf>*</mirrorOf>
      <url>https://nexus.example.com/repository/maven-public/</url>
    </mirror>
  </mirrors>

  <profiles>
    <profile>
      <id>nexus</id>
      <repositories>
        <repository>
          <id>central</id>
          <url>http://central</url>
          <releases>
            <enabled>true</enabled>
          </releases>
          <snapshots>
            <enabled>true</enabled>
          </snapshots>
        </repository>
      </repositories>
    </profile>
  </profiles>

  <activeProfiles>
    <activeProfile>nexus</activeProfile>
  </activeProfiles>
</settings>
```

Publishing artifacts:

```
<distributionManagement>
    <repository>
        <id>nexus</id>
        <name>Releases</name>
        <url>https://nexus.example.com/repository/maven-releases</url>
    </repository>
    <snapshotRepository>
        <id>nexus</id>
        <name>Snapshots</name>
        <url>https://nexus.example.com/repository/maven-snapshots</url>
    </snapshotRepository>
</distributionManagement>
```

H. Advanced Topics

Reactive Programming with Spring WebFlux

Building reactive applications with Spring WebFlux for improved scalability and resource utilization.

Setting Up a Reactive Application

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-webflux</artifactId>
</dependency>
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-data-mongodb-reactive</artifactId>
</dependency>
```

Reactive Controllers

```
@RestController
@RequestMapping("/api/products")
public class ProductReactiveController {

    private final ProductReactiveService productService;

    public ProductReactiveController(ProductReactiveService productService) {
        this.productService = productService;
    }

    @GetMapping
    public Flux<ProductDto> getAllProducts() {
        return productService.findAll();
    }

    @GetMapping("/{id}")
    public Mono<ResponseEntity<ProductDto>> getProductById(@PathVariable String
id) {
        return productService.findById(id)
                .map(ResponseEntity::ok)
                .defaultIfEmpty(ResponseEntity.notFound().build());
    }

    @PostMapping
    public Mono<ResponseEntity<ProductDto>> createProduct(@Valid @RequestBody
Mono<ProductDto> productDtoMono) {
        return productService.createProduct(productDtoMono)
                .map(productDto -> ResponseEntity.created(
                        URI.create("/api/products/" + productDto.getId()))
```

```
                        .body(productDto));
    }

    @PutMapping("/{id}")
    public Mono<ResponseEntity<ProductDto>> updateProduct(
            @PathVariable String id, @Valid @RequestBody Mono<ProductDto>
productDtoMono) {

        return productService.updateProduct(id, productDtoMono)
                .map(ResponseEntity::ok)
                .defaultIfEmpty(ResponseEntity.notFound().build());
    }

    @DeleteMapping("/{id}")
    public Mono<ResponseEntity<Void>> deleteProduct(@PathVariable String id) {
        return productService.deleteProduct(id)
                .then(Mono.just(ResponseEntity.noContent().<Void>build()))
                .defaultIfEmpty(ResponseEntity.notFound().build());
    }

    @GetMapping("/search")
    public Flux<ProductDto> searchProducts(
            @RequestParam(required = false) String keyword,
            @RequestParam(required = false) Double minPrice,
            @RequestParam(required = false) Double maxPrice) {

        return productService.searchProducts(keyword, minPrice, maxPrice);
    }
}
```

Reactive Services

```
@Service
public class ProductReactiveServiceImpl implements ProductReactiveService {

    private final ProductReactiveRepository productRepository;
    private final CategoryReactiveRepository categoryRepository;

    public ProductReactiveServiceImpl(ProductReactiveRepository productRepository,
                                      CategoryReactiveRepository
categoryRepository) {
        this.productRepository = productRepository;
        this.categoryRepository = categoryRepository;
    }

    @Override
    public Flux<ProductDto> findAll() {
        return productRepository.findAll()
                .map(this::mapToDto);
    }
```

```
    @Override
    public Mono<ProductDto> findById(String id) {
        return productRepository.findById(id)
                .map(this::mapToDto);
    }

    @Override
    public Mono<ProductDto> createProduct(Mono<ProductDto> productDtoMono) {
        return productDtoMono
                .flatMap(productDto ->
categoryRepository.findById(productDto.getCategoryId())
                        .switchIfEmpty(Mono.error(new ResourceNotFoundException(
                                "Category not found with id: " +
productDto.getCategoryId())))
                        .map(category -> {
                            Product product = mapToEntity(productDto);
                            product.setCategory(category);
                            return product;
                        }))
                .flatMap(productRepository::save)
                .map(this::mapToDto);
    }

    @Override
    public Mono<ProductDto> updateProduct(String id, Mono<ProductDto>
productDtoMono) {
        return productRepository.findById(id)
                .flatMap(existingProduct -> productDtoMono
                        .flatMap(productDto ->
categoryRepository.findById(productDto.getCategoryId())
                                .switchIfEmpty(Mono.error(new
ResourceNotFoundException(
                                        "Category not found with id: " +
productDto.getCategoryId())))
                                .map(category -> {
                                    existingProduct.setName(productDto.getName());

existingProduct.setDescription(productDto.getDescription());

existingProduct.setPrice(productDto.getPrice());
                                    existingProduct.setCategory(category);
                                    return existingProduct;
                                })))
                .flatMap(productRepository::save)
                .map(this::mapToDto);
    }

    @Override
    public Mono<Void> deleteProduct(String id) {
        return productRepository.findById(id)
                .flatMap(product -> productRepository.delete(product));
    }

    @Override
```

```
    public Flux<ProductDto> searchProducts(String keyword, Double minPrice, Double
maxPrice) {
        Flux<Product> result = Flux.empty();

        // Search by keyword
        if (keyword != null && !keyword.isEmpty()) {
            result = productRepository.findByNameContainingIgnoreCase(keyword);
        }

        // Search by price range
        if (minPrice != null && maxPrice != null) {
            Flux<Product> priceRangeProducts =
productRepository.findByPriceBetween(minPrice, maxPrice);

            if (result.collectList().block().isEmpty()) {
                result = priceRangeProducts;
            } else {
                // Combine the results (intersection)
                result = result.filter(product ->
                        priceRangeProducts.any(p ->
p.getId().equals(product.getId())).block());
            }
        } else if (minPrice != null) {
            Flux<Product> priceFloor =
productRepository.findByPriceGreaterThanEqual(minPrice);

            if (result.collectList().block().isEmpty()) {
                result = priceFloor;
            } else {
                result = result.filter(product ->
                        priceFloor.any(p ->
p.getId().equals(product.getId())).block());
            }
        } else if (maxPrice != null) {
            Flux<Product> priceCeiling =
productRepository.findByPriceLessThanEqual(maxPrice);

            if (result.collectList().block().isEmpty()) {
                result = priceCeiling;
            } else {
                result = result.filter(product ->
                        priceCeiling.any(p ->
p.getId().equals(product.getId())).block());
            }
        }

        // If no criteria specified, return all products
        if (keyword == null && minPrice == null && maxPrice == null) {
            result = productRepository.findAll();
        }

        return result.map(this::mapToDto);
    }
```

```
    private ProductDto mapToDto(Product product) {
        ProductDto dto = new ProductDto();
        dto.setId(product.getId());
        dto.setName(product.getName());
        dto.setDescription(product.getDescription());
        dto.setPrice(product.getPrice());
        dto.setCategoryId(product.getCategory().getId());
        return dto;
    }

    private Product mapToEntity(ProductDto dto) {
        Product product = new Product();
        product.setId(dto.getId());
        product.setName(dto.getName());
        product.setDescription(dto.getDescription());
        product.setPrice(dto.getPrice());
        return product;
    }
}
```

Reactive Repositories

```
@Repository
public interface ProductReactiveRepository extends
ReactiveMongoRepository<Product, String> {

    Flux<Product> findByNameContainingIgnoreCase(String name);

    Flux<Product> findByPriceBetween(Double minPrice, Double maxPrice);

    Flux<Product> findByPriceGreaterThanEqual(Double price);

    Flux<Product> findByPriceLessThanEqual(Double price);

    Flux<Product> findByCategoryId(String categoryId);
}
```

Reactive Error Handling

```
@Component
public class GlobalErrorWebExceptionHandler extends
AbstractErrorWebExceptionHandler {

    public GlobalErrorWebExceptionHandler(ErrorAttributes errorAttributes,
                                          WebProperties webProperties,
                                          ApplicationContext applicationContext) {
        super(errorAttributes, webProperties.getResources(), applicationContext);
    }
```

```
    @Override
    protected RouterFunction<ServerResponse> getRoutingFunction(ErrorAttributes
errorAttributes) {
        return RouterFunctions.route(RequestPredicates.all(),
this::renderErrorResponse);
    }

    private Mono<ServerResponse> renderErrorResponse(ServerRequest request) {
        Map<String, Object> errorPropertiesMap = getErrorAttributes(request,
ErrorAttributeOptions.defaults());

        // Customize the error response based on the exception type
        Throwable error = getError(request);
        HttpStatus status = determineHttpStatus(error);

        // Build a custom error model
        ErrorResponse errorResponse = new ErrorResponse();
        errorResponse.setStatus(status.value());
        errorResponse.setError(status.getReasonPhrase());

        if (error instanceof ResourceNotFoundException) {
            errorResponse.setMessage(error.getMessage());
        } else if (error instanceof ValidationException) {
            errorResponse.setMessage("Validation failed");
            // Add validation errors
            ValidationException validationException = (ValidationException) error;
            errorResponse.setErrors(validationException.getErrors());
        } else {
            errorResponse.setMessage("An unexpected error occurred");
        }

        // Add timestamp and path
        errorResponse.setTimestamp(LocalDateTime.now());
        errorResponse.setPath(request.path());

        return ServerResponse.status(status)
                .contentType(MediaType.APPLICATION_JSON)
                .bodyValue(errorResponse);
    }

    private HttpStatus determineHttpStatus(Throwable error) {
        if (error instanceof ResourceNotFoundException) {
            return HttpStatus.NOT_FOUND;
        } else if (error instanceof ValidationException) {
            return HttpStatus.BAD_REQUEST;
        } else if (error instanceof AccessDeniedException) {
            return HttpStatus.FORBIDDEN;
        }
        return HttpStatus.INTERNAL_SERVER_ERROR;
    }

    // Error response model
    @Data
```

```
    static class ErrorResponse {
        private int status;
        private String error;
        private String message;
        private Map<String, String> errors;
        private LocalDateTime timestamp;
        private String path;
    }
}
```

Testing Reactive Applications

```
@SpringBootTest
public class ProductReactiveServiceTest {

    @Autowired
    private ProductReactiveService productService;

    @MockBean
    private ProductReactiveRepository productRepository;

    @MockBean
    private CategoryReactiveRepository categoryRepository;

    @Test
    void findAll_ReturnsAllProducts() {
        // Arrange
        Product product1 = new Product();
        product1.setId("1");
        product1.setName("Product 1");
        product1.setDescription("Description 1");
        product1.setPrice(99.99);

        Category category1 = new Category();
        category1.setId("1");
        category1.setName("Category 1");
        product1.setCategory(category1);

        Product product2 = new Product();
        product2.setId("2");
        product2.setName("Product 2");
        product2.setDescription("Description 2");
        product2.setPrice(199.99);
        product2.setCategory(category1);

        when(productRepository.findAll()).thenReturn(Flux.just(product1,
product2));

        // Act
        Flux<ProductDto> result = productService.findAll();
```

```
        // Assert
        StepVerifier.create(result)
                .expectNextMatches(dto -> dto.getId().equals("1") &&
dto.getName().equals("Product 1"))
                .expectNextMatches(dto -> dto.getId().equals("2") &&
dto.getName().equals("Product 2"))
                .verifyComplete();

        verify(productRepository).findAll();
    }

    @Test
    void findById_ExistingProduct_ReturnsProduct() {
        // Arrange
        String productId = "1";

        Product product = new Product();
        product.setId(productId);
        product.setName("Test Product");
        product.setDescription("Test Description");
        product.setPrice(99.99);

        Category category = new Category();
        category.setId("1");
        category.setName("Test Category");
        product.setCategory(category);


when(productRepository.findById(productId)).thenReturn(Mono.just(product));

        // Act
        Mono<ProductDto> result = productService.findById(productId);

        // Assert
        StepVerifier.create(result)
                .expectNextMatches(dto ->
                        dto.getId().equals(productId) &&
                        dto.getName().equals("Test Product") &&
                        dto.getDescription().equals("Test Description") &&
                        dto.getPrice() == 99.99 &&
                        dto.getCategoryId().equals("1"))
                .verifyComplete();

        verify(productRepository).findById(productId);
    }

    @Test
    void findById_NonExistingProduct_ReturnsEmptyMono() {
        // Arrange
        String productId = "999";

        when(productRepository.findById(productId)).thenReturn(Mono.empty());

        // Act
```

```
        Mono<ProductDto> result = productService.findById(productId);

        // Assert
        StepVerifier.create(result)
                .verifyComplete();

        verify(productRepository).findById(productId);
    }

    @Test
    void createProduct_Success() {
        // Arrange
        ProductDto productDto = new ProductDto();
        productDto.setName("New Product");
        productDto.setDescription("New Description");
        productDto.setPrice(149.99);
        productDto.setCategoryId("1");

        Category category = new Category();
        category.setId("1");
        category.setName("Category");

        Product savedProduct = new Product();
        savedProduct.setId("1");
        savedProduct.setName("New Product");
        savedProduct.setDescription("New Description");
        savedProduct.setPrice(149.99);
        savedProduct.setCategory(category);

        when(categoryRepository.findById("1")).thenReturn(Mono.just(category));

when(productRepository.save(any(Product.class))).thenReturn(Mono.just(savedProduct
));

        // Act
        Mono<ProductDto> result =
productService.createProduct(Mono.just(productDto));

        // Assert
        StepVerifier.create(result)
                .expectNextMatches(dto ->
                        dto.getId().equals("1") &&
                        dto.getName().equals("New Product") &&
                        dto.getPrice() == 149.99 &&
                        dto.getCategoryId().equals("1"))
                .verifyComplete();

        verify(categoryRepository).findById("1");
        verify(productRepository).save(any(Product.class));
    }
}
```

WebClient for Reactive HTTP Requests

```
@Configuration
public class WebClientConfig {

    @Bean
    public WebClient webClient() {
        return WebClient.builder()
                .baseUrl("https://api.example.com")
                .defaultHeader(HttpHeaders.CONTENT_TYPE,
MediaType.APPLICATION_JSON_VALUE)
                .build();
    }
}
```

```
@Service
public class CategoryReactiveClient {

    private final WebClient webClient;

    public CategoryReactiveClient(WebClient webClient) {
        this.webClient = webClient;
    }

    public Mono<CategoryDto> getCategory(String id) {
        return webClient.get()
                .uri("/api/categories/{id}", id)
                .retrieve()
                .onStatus(HttpStatus::is4xxClientError, response -> Mono.error(
                        new ResourceNotFoundException("Category not found with id:
" + id)))
                .onStatus(HttpStatus::is5xxServerError, response -> Mono.error(
                        new ServiceUnavailableException("Category service is
unavailable")))
                .bodyToMono(CategoryDto.class);
    }

    public Flux<CategoryDto> getAllCategories() {
        return webClient.get()
                .uri("/api/categories")
                .retrieve()
                .onStatus(HttpStatus::isError, response ->
                        response.bodyToMono(String.class)
                                .flatMap(body -> Mono.error(new
RuntimeException(body))))
                .bodyToFlux(CategoryDto.class);
    }

    public Mono<CategoryDto> createCategory(CategoryDto categoryDto) {
        return webClient.post()
```

```
                .uri("/api/categories")
                .bodyValue(categoryDto)
                .retrieve()
                .onStatus(HttpStatus::isError, response ->
                        response.bodyToMono(String.class)
                                .flatMap(body -> Mono.error(new
RuntimeException(body))))
                .bodyToMono(CategoryDto.class);
    }
}
```

Functional Endpoints

```
@Configuration
public class ProductRouterConfig {

    @Bean
    public RouterFunction<ServerResponse> productRoutes(ProductHandler handler) {
        return RouterFunctions

.route(GET("/api/products").and(accept(MediaType.APPLICATION_JSON)),
handler::getAllProducts)

.andRoute(GET("/api/products/{id}").and(accept(MediaType.APPLICATION_JSON)),
handler::getProductById)

.andRoute(POST("/api/products").and(accept(MediaType.APPLICATION_JSON)),
handler::createProduct)

.andRoute(PUT("/api/products/{id}").and(accept(MediaType.APPLICATION_JSON)),
handler::updateProduct)
                .andRoute(DELETE("/api/products/{id}"), handler::deleteProduct)

.andRoute(GET("/api/products/search").and(accept(MediaType.APPLICATION_JSON)),
handler::searchProducts);
    }
}
```

```
@Component
public class ProductHandler {

    private final ProductReactiveService productService;
    private final Validator validator;

    public ProductHandler(ProductReactiveService productService, Validator
validator) {
        this.productService = productService;
        this.validator = validator;
    }
```

```
    public Mono<ServerResponse> getAllProducts(ServerRequest request) {
        return ServerResponse.ok()
                .contentType(MediaType.APPLICATION_JSON)
                .body(productService.findAll(), ProductDto.class);
    }

    public Mono<ServerResponse> getProductById(ServerRequest request) {
        String id = request.pathVariable("id");
        return productService.findById(id)
                .flatMap(product -> ServerResponse.ok()
                        .contentType(MediaType.APPLICATION_JSON)
                        .bodyValue(product))
                .switchIfEmpty(ServerResponse.notFound().build());
    }

    public Mono<ServerResponse> createProduct(ServerRequest request) {
        return request.bodyToMono(ProductDto.class)
                .doOnNext(this::validate)
                .flatMap(productService::createProduct)
                .flatMap(product -> ServerResponse.created(
                        URI.create("/api/products/" + product.getId()))
                        .contentType(MediaType.APPLICATION_JSON)
                        .bodyValue(product));
    }

    public Mono<ServerResponse> updateProduct(ServerRequest request) {
        String id = request.pathVariable("id");
        return request.bodyToMono(ProductDto.class)
                .doOnNext(this::validate)
                .flatMap(dto -> productService.updateProduct(id, Mono.just(dto)))
                .flatMap(product -> ServerResponse.ok()
                        .contentType(MediaType.APPLICATION_JSON)
                        .bodyValue(product))
                .switchIfEmpty(ServerResponse.notFound().build());
    }

    public Mono<ServerResponse> deleteProduct(ServerRequest request) {
        String id = request.pathVariable("id");
        return productService.deleteProduct(id)
                .then(ServerResponse.noContent().build())
                .switchIfEmpty(ServerResponse.notFound().build());
    }

    public Mono<ServerResponse> searchProducts(ServerRequest request) {
        String keyword = request.queryParam("keyword").orElse(null);
        Double minPrice = request.queryParam("minPrice")
                .map(Double::parseDouble)
                .orElse(null);
        Double maxPrice = request.queryParam("maxPrice")
                .map(Double::parseDouble)
                .orElse(null);

        return ServerResponse.ok()
```

```
                .contentType(MediaType.APPLICATION_JSON)
                .body(productService.searchProducts(keyword, minPrice, maxPrice),
ProductDto.class);
    }

    private void validate(ProductDto productDto) {
        Set<ConstraintViolation<ProductDto>> violations =
validator.validate(productDto);
        if (!violations.isEmpty()) {
            Map<String, String> errors = violations.stream()
                    .collect(Collectors.toMap(
                            violation -> violation.getPropertyPath().toString(),
                            ConstraintViolation::getMessage
                    ));
            throw new ValidationException("Validation failed", errors);
        }
    }
}
```

WebSockets and Server-Sent Events

Real-time communication with WebSockets and SSE.

WebSocket Configuration

```
@Configuration
@EnableWebSocketMessageBroker
public class WebSocketConfig implements WebSocketMessageBrokerConfigurer {

    @Override
    public void configureMessageBroker(MessageBrokerRegistry registry) {
        // Enable a simple memory-based message broker
        registry.enableSimpleBroker("/topic");
        // Set prefix for client-to-server messages
        registry.setApplicationDestinationPrefixes("/app");
    }

    @Override
    public void registerStompEndpoints(StompEndpointRegistry registry) {
        // Register the "/ws" endpoint, enabling SockJS fallback options
        registry.addEndpoint("/ws")
                .setAllowedOrigins("*")
                .withSockJS();
    }
}
```

WebSocket Controller

```
@Controller
public class ProductWebSocketController {

    private final SimpMessagingTemplate messagingTemplate;
    private final ProductService productService;

    public ProductWebSocketController(SimpMessagingTemplate messagingTemplate,
                                      ProductService productService) {
        this.messagingTemplate = messagingTemplate;
        this.productService = productService;
    }

    @MessageMapping("/products.create")
    @SendTo("/topic/products")
    public ProductDto createProduct(ProductDto productDto) {
        return productService.createProduct(productDto);
    }

    @MessageMapping("/products.update")
    public void updateProduct(ProductUpdateRequest updateRequest) {
        ProductDto updatedProduct = productService.updateProduct(
                updateRequest.getProductId(), updateRequest.getProductDto());

        // Send to specific topic based on product ID
        messagingTemplate.convertAndSend(
                "/topic/products/" + updateRequest.getProductId(),
updatedProduct);

        // Send to general topic
        messagingTemplate.convertAndSend("/topic/products", updatedProduct);
    }

    // Data class for update requests
    @Data
    public static class ProductUpdateRequest {
        private Long productId;
        private ProductDto productDto;
    }
}
```

WebSocket Service Integration

```
@Service
public class ProductNotificationService {

    private final SimpMessagingTemplate messagingTemplate;

    public ProductNotificationService(SimpMessagingTemplate messagingTemplate) {
        this.messagingTemplate = messagingTemplate;
    }
```

```
    public void notifyProductCreated(ProductDto productDto) {
        messagingTemplate.convertAndSend("/topic/products", productDto);
    }

    public void notifyProductUpdated(ProductDto productDto) {
        messagingTemplate.convertAndSend("/topic/products/" + productDto.getId(),
productDto);
        messagingTemplate.convertAndSend("/topic/products", productDto);
    }

    public void notifyProductDeleted(Long productId) {
        messagingTemplate.convertAndSend("/topic/products/deleted", productId);
    }

    public void notifyLowStock(ProductStockDto stockDto) {
        messagingTemplate.convertAndSend("/topic/inventory/lowstock", stockDto);
    }

    @Data
    public static class ProductStockDto {
        private Long productId;
        private String productName;
        private Integer currentStock;
        private Integer minStockLevel;
    }
}
```

```
@Service
public class ProductServiceWithNotifications implements ProductService {

    private final ProductRepository productRepository;
    private final CategoryRepository categoryRepository;
    private final ProductMapper productMapper;
    private final ProductNotificationService notificationService;

    public ProductServiceWithNotifications(ProductRepository productRepository,
                                           CategoryRepository categoryRepository,
                                           ProductMapper productMapper,
                                           ProductNotificationService
notificationService) {
        this.productRepository = productRepository;
        this.categoryRepository = categoryRepository;
        this.productMapper = productMapper;
        this.notificationService = notificationService;
    }

    @Override
    @Transactional
    public ProductDto createProduct(ProductDto productDto) {
        Category category =
```

```
        categoryRepository.findById(productDto.getCategoryId())
                .orElseThrow(() -> new ResourceNotFoundException(
                        "Category not found with id: " +
productDto.getCategoryId()));

        Product product = productMapper.toEntity(productDto);
        product.setCategory(category);

        Product savedProduct = productRepository.save(product);
        ProductDto savedDto = productMapper.toDto(savedProduct);

        // Send notification
        notificationService.notifyProductCreated(savedDto);

        return savedDto;
    }

    @Override
    @Transactional
    public ProductDto updateProduct(Long id, ProductDto productDto) {
        return productRepository.findById(id)
                .map(existingProduct -> {
                    Category category =
categoryRepository.findById(productDto.getCategoryId())
                            .orElseThrow(() -> new ResourceNotFoundException(
                                    "Category not found with id: " +
productDto.getCategoryId()));

                    existingProduct.setName(productDto.getName());
                    existingProduct.setDescription(productDto.getDescription());
                    existingProduct.setPrice(productDto.getPrice());
                    existingProduct.setCategory(category);

                    Product updatedProduct =
productRepository.save(existingProduct);
                    ProductDto updatedDto = productMapper.toDto(updatedProduct);

                    // Send notification
                    notificationService.notifyProductUpdated(updatedDto);

                    return updatedDto;
                })
                .orElseThrow(() -> new ResourceNotFoundException("Product not
found with id: " + id));
    }

    @Override
    @Transactional
    public void deleteProduct(Long id) {
        Product product = productRepository.findById(id)
                .orElseThrow(() -> new ResourceNotFoundException("Product not
found with id: " + id));

        productRepository.delete(product);
```

```
        // Send notification
        notificationService.notifyProductDeleted(id);
    }

    // Other methods from ProductService interface...
}
```

WebSocket JavaScript Client

```
// Connect to WebSocket
const socket = new SockJS('/ws');
const stompClient = Stomp.over(socket);

stompClient.connect({}, function (frame) {
  console.log('Connected to WebSocket: ' + frame);

  // Subscribe to product updates
  stompClient.subscribe('/topic/products', function (message) {
    const product = JSON.parse(message.body);
    console.log('Received product update:', product);
    updateProductUI(product);
  });

  // Subscribe to specific product updates
  stompClient.subscribe('/topic/products/1', function (message) {
    const product = JSON.parse(message.body);
    console.log('Received specific product update:', product);
    updateSpecificProductUI(product);
  });

  // Subscribe to product deletions
  stompClient.subscribe('/topic/products/deleted', function (message) {
    const productId = JSON.parse(message.body);
    console.log('Product deleted:', productId);
    removeProductFromUI(productId);
  });

  // Subscribe to low stock notifications
  stompClient.subscribe('/topic/inventory/lowstock', function (message) {
    const stockInfo = JSON.parse(message.body);
    console.log('Low stock alert:', stockInfo);
    showLowStockAlert(stockInfo);
  });
});

// Send a create product request
function createProduct(product) {
  stompClient.send('/app/products.create', {}, JSON.stringify(product));
}
```

```
// Send an update product request
function updateProduct(productId, product) {
  const updateRequest = {
    productId: productId,
    productDto: product,
  };
  stompClient.send('/app/products.update', {}, JSON.stringify(updateRequest));
}

// UI update functions
function updateProductUI(product) {
  // Update product list
}

function updateSpecificProductUI(product) {
  // Update specific product details
}

function removeProductFromUI(productId) {
  // Remove product from UI
}

function showLowStockAlert(stockInfo) {
  // Show low stock alert
}
```

Server-Sent Events (SSE)

```
@RestController
@RequestMapping("/api/sse")
public class ServerSentEventController {

    private final ProductService productService;

    public ServerSentEventController(ProductService productService) {
        this.productService = productService;
    }

    @GetMapping(value = "/products", produces = MediaType.TEXT_EVENT_STREAM_VALUE)
    public Flux<ServerSentEvent<ProductDto>> streamProducts() {
        return Flux.interval(Duration.ofSeconds(1))
                .map(sequence -> ServerSentEvent.<ProductDto>builder()
                        .id(String.valueOf(sequence))
                        .event("product-list")
                        .data(productService.getRandomProduct())
                        .build());
    }

    @GetMapping(value = "/products/{id}", produces =
MediaType.TEXT_EVENT_STREAM_VALUE)
    public Flux<ServerSentEvent<ProductDto>> streamProduct(@PathVariable Long id)
```

```
{
        return Flux.interval(Duration.ofSeconds(5))
                .map(sequence -> {
                    ProductDto product = productService.findById(id);
                    return ServerSentEvent.<ProductDto>builder()
                            .id(String.valueOf(sequence))
                            .event("product-update")
                            .data(product)
                            .build();
                })
                .onErrorResume(e -> Flux.empty());
    }

    @GetMapping(value = "/inventory", produces =
MediaType.TEXT_EVENT_STREAM_VALUE)
    public Flux<ServerSentEvent<InventoryUpdate>> streamInventory() {
        return Flux.interval(Duration.ofSeconds(3))
                .map(sequence -> {
                    InventoryUpdate update = productService.getInventoryUpdate();
                    return ServerSentEvent.<InventoryUpdate>builder()
                            .id(String.valueOf(sequence))
                            .event("inventory-update")
                            .data(update)
                            .build();
                });
    }

    @Data
    public static class InventoryUpdate {
        private Long productId;
        private String productName;
        private Integer currentStock;
        private LocalDateTime timestamp;
    }
}
```

JavaScript client for SSE:

```
// Connect to SSE for product list
const productListSource = new EventSource('/api/sse/products');

productListSource.addEventListener('product-list', function (event) {
  const product = JSON.parse(event.data);
  console.log('Received product update via SSE:', product);
  updateProductListUI(product);
});

productListSource.onerror = function (error) {
  console.error('Error in product list SSE:', error);
  productListSource.close();
};
```

```
// Connect to SSE for specific product
const productSource = new EventSource('/api/sse/products/1');

productSource.addEventListener('product-update', function (event) {
  const product = JSON.parse(event.data);
  console.log('Received product update via SSE:', product);
  updateProductDetailUI(product);
});

productSource.onerror = function (error) {
  console.error('Error in product SSE:', error);
  productSource.close();
};

// Connect to SSE for inventory updates
const inventorySource = new EventSource('/api/sse/inventory');

inventorySource.addEventListener('inventory-update', function (event) {
  const inventoryUpdate = JSON.parse(event.data);
  console.log('Received inventory update via SSE:', inventoryUpdate);
  updateInventoryUI(inventoryUpdate);
});

inventorySource.onerror = function (error) {
  console.error('Error in inventory SSE:', error);
  inventorySource.close();
};

// UI update functions
function updateProductListUI(product) {
  // Update product list UI
}

function updateProductDetailUI(product) {
  // Update product detail UI
}

function updateInventoryUI(inventoryUpdate) {
  // Update inventory UI
}
```

Caching Strategies and Implementations

Implementing caching to improve application performance.

Caching Configuration

```
@Configuration
@EnableCaching
public class CacheConfig {
```

```
    @Bean
    public CacheManager cacheManager() {
        CaffeineCacheManager cacheManager = new CaffeineCacheManager();
        cacheManager.setCaffeine(Caffeine.newBuilder()
                .expireAfterWrite(60, TimeUnit.MINUTES)
                .initialCapacity(100)
                .maximumSize(1000));
        cacheManager.setCacheNames(Arrays.asList(
                "products", "categories", "product-search"));
        return cacheManager;
    }

    @Bean
    public CacheManager customCacheManager() {
        SimpleCacheManager cacheManager = new SimpleCacheManager();

        // Products cache with short TTL
        CaffeineCache productsCache = new CaffeineCache("products",
                Caffeine.newBuilder()
                        .expireAfterWrite(10, TimeUnit.MINUTES)
                        .maximumSize(500)
                        .recordStats()
                        .build());

        // Categories cache with longer TTL
        CaffeineCache categoriesCache = new CaffeineCache("categories",
                Caffeine.newBuilder()
                        .expireAfterWrite(30, TimeUnit.MINUTES)
                        .maximumSize(50)
                        .recordStats()
                        .build());

        // Search results cache with very short TTL
        CaffeineCache searchCache = new CaffeineCache("product-search",
                Caffeine.newBuilder()
                        .expireAfterWrite(5, TimeUnit.MINUTES)
                        .maximumSize(200)
                        .recordStats()
                        .build());

        cacheManager.setCaches(Arrays.asList(productsCache, categoriesCache,
searchCache));
        return cacheManager;
    }
}
```

Redis Caching Configuration

```
<dependency>
    <groupId>org.springframework.boot</groupId>
```

```
    <artifactId>spring-boot-starter-data-redis</artifactId>
</dependency>
```

```
@Configuration
@EnableCaching
public class RedisCacheConfig {

    @Bean
    public RedisConnectionFactory redisConnectionFactory() {
        LettuceConnectionFactory connectionFactory = new
LettuceConnectionFactory();
        connectionFactory.afterPropertiesSet();
        return connectionFactory;
    }

    @Bean
    public RedisCacheManager cacheManager(RedisConnectionFactory
connectionFactory) {
        // Default cache configuration
        RedisCacheConfiguration defaultConfig =
RedisCacheConfiguration.defaultCacheConfig()
                .entryTtl(Duration.ofMinutes(10))
                .serializeKeysWith(

RedisSerializationContext.SerializationPair.fromSerializer(new
StringRedisSerializer()))
                .serializeValuesWith(

RedisSerializationContext.SerializationPair.fromSerializer(
                                new GenericJackson2JsonRedisSerializer()));

        // Configure cache TTLs for different caches
        Map<String, RedisCacheConfiguration> cacheConfigurations = new HashMap<>
();

        // Products cache with medium TTL
        cacheConfigurations.put("products",
                defaultConfig.entryTtl(Duration.ofMinutes(10)));

        // Categories cache with longer TTL
        cacheConfigurations.put("categories",
                defaultConfig.entryTtl(Duration.ofHours(1)));

        // Search results cache with short TTL
        cacheConfigurations.put("product-search",
                defaultConfig.entryTtl(Duration.ofMinutes(1)));

        // Product statistics cache
        cacheConfigurations.put("product-stats",
                defaultConfig.entryTtl(Duration.ofHours(3)));
```

```
        return RedisCacheManager.builder(connectionFactory)
                .cacheDefaults(defaultConfig)
                .withInitialCacheConfigurations(cacheConfigurations)
                .build();
    }
}
```

Service with Caching

```
@Service
public class ProductServiceWithCaching implements ProductService {

    private final ProductRepository productRepository;
    private final CategoryRepository categoryRepository;
    private final ProductMapper productMapper;

    public ProductServiceWithCaching(ProductRepository productRepository,
                                     CategoryRepository categoryRepository,
                                     ProductMapper productMapper) {
        this.productRepository = productRepository;
        this.categoryRepository = categoryRepository;
        this.productMapper = productMapper;
    }

    @Override
    @Cacheable(value = "products")
    public List<ProductDto> findAll() {
        return productRepository.findAll().stream()
                .map(productMapper::toDto)
                .collect(Collectors.toList());
    }

    @Override
    @Cacheable(value = "products", key = "#id")
    public ProductDto findById(Long id) {
        return productRepository.findById(id)
                .map(productMapper::toDto)
                .orElseThrow(() -> new ResourceNotFoundException("Product not
found with id: " + id));
    }

    @Override
    @Cacheable(value = "products", key = "#categoryId")
    public List<ProductDto> findByCategoryId(Long categoryId) {
        return productRepository.findByCategoryId(categoryId).stream()
                .map(productMapper::toDto)
                .collect(Collectors.toList());
    }

    @Override
    @Cacheable(value = "product-search", key =
```

```
"T(java.util.Objects).hash(#keyword, #minPrice, #maxPrice)")
    public List<ProductDto> searchProducts(String keyword, BigDecimal minPrice,
BigDecimal maxPrice) {
        List<Product> results =
productRepository.findByNameContainingIgnoreCase(keyword);

        if (minPrice != null && maxPrice != null) {
            results = results.stream()
                    .filter(p -> p.getPrice().compareTo(minPrice) >= 0 &&
p.getPrice().compareTo(maxPrice) <= 0)
                    .collect(Collectors.toList());
        } else if (minPrice != null) {
            results = results.stream()
                    .filter(p -> p.getPrice().compareTo(minPrice) >= 0)
                    .collect(Collectors.toList());
        } else if (maxPrice != null) {
            results = results.stream()
                    .filter(p -> p.getPrice().compareTo(maxPrice) <= 0)
                    .collect(Collectors.toList());
        }

        return results.stream()
                .map(productMapper::toDto)
                .collect(Collectors.toList());
    }

    @Override
    @CachePut(value = "products", key = "#result.id")
    @Transactional
    public ProductDto createProduct(ProductDto productDto) {
        Category category =
categoryRepository.findById(productDto.getCategoryId())
                .orElseThrow(() -> new ResourceNotFoundException(
                        "Category not found with id: " +
productDto.getCategoryId()));

        Product product = productMapper.toEntity(productDto);
        product.setCategory(category);

        Product savedProduct = productRepository.save(product);
        return productMapper.toDto(savedProduct);
    }

    @Override
    @CachePut(value = "products", key = "#id")
    @CacheEvict(value = "product-search", allEntries = true)
    @Transactional
    public ProductDto updateProduct(Long id, ProductDto productDto) {
        return productRepository.findById(id)
                .map(existingProduct -> {
                    Category category =
categoryRepository.findById(productDto.getCategoryId())
                            .orElseThrow(() -> new ResourceNotFoundException(
                                    "Category not found with id: " +
```

```
productDto.getCategoryId()));

                    existingProduct.setName(productDto.getName());
                    existingProduct.setDescription(productDto.getDescription());
                    existingProduct.setPrice(productDto.getPrice());
                    existingProduct.setCategory(category);

                    Product updatedProduct =
productRepository.save(existingProduct);
                    return productMapper.toDto(updatedProduct);
                })
                .orElseThrow(() -> new ResourceNotFoundException("Product not
found with id: " + id));
    }

    @Override
    @CacheEvict(value = {"products", "product-search"}, allEntries = true)
    @Transactional
    public void deleteProduct(Long id) {
        Product product = productRepository.findById(id)
                .orElseThrow(() -> new ResourceNotFoundException("Product not
found with id: " + id));

        productRepository.delete(product);
    }

    @Override
    @Cacheable(value = "product-stats", key = "'admin-statistics'")
    public Map<String, Object> getAdminStatistics() {
        // Expensive operation to calculate statistics
        long totalProducts = productRepository.count();
        long totalCategories = categoryRepository.count();
        double averagePrice = productRepository.findAveragePrice();

        Map<String, Object> statistics = new HashMap<>();
        statistics.put("totalProducts", totalProducts);
        statistics.put("totalCategories", totalCategories);
        statistics.put("averagePrice", averagePrice);

        return statistics;
    }

    @Scheduled(fixedRate = 3600000) // Every hour
    @CacheEvict(value = "product-stats", allEntries = true)
    public void clearProductStatisticsCache() {
        // Clear product statistics cache to refresh data
    }

    @Scheduled(cron = "0 0 0 * * ?") // Midnight every day
    @CacheEvict(value = {"products", "product-search"}, allEntries = true)
    public void clearProductCaches() {
        // Clear product caches every day
    }
}
```

Custom Cache Key Generator

```
@Component
public class CustomCacheKeyGenerator implements KeyGenerator {

    @Override
    public Object generate(Object target, Method method, Object... params) {
        StringBuilder sb = new StringBuilder();
        sb.append(target.getClass().getSimpleName())
          .append(":")
          .append(method.getName());

        for (Object param : params) {
            sb.append(":")
              .append(getParamValue(param));
        }

        return sb.toString();
    }

    private String getParamValue(Object param) {
        if (param == null) {
            return "null";
        }

        if (param instanceof Collection) {
            return ((Collection<?>) param).stream()
                    .map(Object::toString)
                    .collect(Collectors.joining(",", "[", "]"));
        }

        return param.toString();
    }
}

@Cacheable(value = "products", keyGenerator = "customCacheKeyGenerator")
public List<ProductDto> findByNameAndCategory(String name, Long categoryId) {
    // Method implementation
}
```

Cache Monitoring and Metrics

```
@Configuration
public class CacheMetricsConfig {

    @Autowired
    private CacheManager cacheManager;
```

```
    @Autowired
    private MeterRegistry registry;

    @PostConstruct
    public void setupCacheMetrics() {
        if (cacheManager instanceof CaffeineCacheManager) {
            // Register Caffeine metrics
            CaffeineCacheManager caffeineManager = (CaffeineCacheManager)
caffeineManager = (CaffeineCacheManager) cacheManager;

            // Register metrics for each cache
            for (String cacheName : caffeineManager.getCacheNames()) {
                Cache cache = caffeineManager.getCache(cacheName);
                if (cache instanceof CaffeineCache) {
                    com.github.benmanes.caffeine.cache.Cache<Object, Object>
nativeCache =
                            ((CaffeineCache) cache).getNativeCache();

                    // Register size metric
                    Gauge.builder("cache.size", nativeCache, c ->
c.estimatedSize())
                            .tag("cache", cacheName)
                            .description("Number of entries in the cache")
                            .register(registry);

                    // Register hit ratio metric
                    Gauge.builder("cache.hit.ratio", nativeCache, c ->
c.stats().hitRate())
                            .tag("cache", cacheName)
                            .description("Cache hit ratio")
                            .register(registry);

                    // Register hit count metric
                    Gauge.builder("cache.hits", nativeCache, c ->
c.stats().hitCount())
                            .tag("cache", cacheName)
                            .description("Cache hit count")
                            .register(registry);

                    // Register miss count metric
                    Gauge.builder("cache.misses", nativeCache, c ->
c.stats().missCount())
                            .tag("cache", cacheName)
                            .description("Cache miss count")
                            .register(registry);

                    // Register evictions metric
                    Gauge.builder("cache.evictions", nativeCache, c ->
c.stats().evictionCount())
                            .tag("cache", cacheName)
                            .description("Cache eviction count")
                            .register(registry);
                }
            }
```

```
        }
    }

    @Bean
    public CacheMetricsRegistrar cacheMetricsRegistrar() {
        return new CacheMetricsRegistrar();
    }

    public static class CacheMetricsRegistrar {
        @Autowired
        private MeterRegistry registry;

        @Autowired
        private List<CacheManager> cacheManagers;

        @EventListener(ApplicationReadyEvent.class)
        public void registerCacheMetrics() {
            cacheManagers.forEach(cacheManager ->
                    cacheManager.getCacheNames().forEach(cacheName ->
                            registerCacheMetrics(cacheManager, cacheName)));
        }

        private void registerCacheMetrics(CacheManager cacheManager, String
cacheName) {
            // Register common cache metrics
            FunctionCounter.builder("cache.gets", cacheManager, cm ->
getCacheGets(cm, cacheName))
                    .tag("cache", cacheName)
                    .description("Number of times cache was accessed")
                    .register(registry);

            FunctionCounter.builder("cache.puts", cacheManager, cm ->
getCachePuts(cm, cacheName))
                    .tag("cache", cacheName)
                    .description("Number of times cache was updated")
                    .register(registry);

            FunctionCounter.builder("cache.evictions", cacheManager, cm ->
getCacheEvictions(cm, cacheName))
                    .tag("cache", cacheName)
                    .description("Number of evictions from cache")
                    .register(registry);
        }

        // These methods depend on the cache implementation and would need custom
implementation
        private long getCacheGets(CacheManager cacheManager, String cacheName) {
            // Implementation depends on cache manager type
            return 0;
        }

        private long getCachePuts(CacheManager cacheManager, String cacheName) {
            // Implementation depends on cache manager type
            return 0;
```

```
        }

        private long getCacheEvictions(CacheManager cacheManager, String
cacheName) {
            // Implementation depends on cache manager type
            return 0;
        }
    }
}
```

Conditional Caching

```
@Service
public class SmartCachingService {

    private final CacheManager cacheManager;

    public SmartCachingService(CacheManager cacheManager) {
        this.cacheManager = cacheManager;
    }

    @Cacheable(value = "products", condition = "#id > 0", unless = "#result ==
null")
    public ProductDto findById(Long id) {
        // Method implementation
    }

    @Cacheable(value = "product-search", condition = "#keyword != null &&
#keyword.length() > 2")
    public List<ProductDto> searchByKeyword(String keyword) {
        // Method implementation
    }

    @Cacheable(value = "premium-products", condition = "#price >= 1000")
    public List<ProductDto> findPremiumProducts(BigDecimal price) {
        // Method implementation
    }

    @CachePut(value = "products", key = "#product.id",
              condition = "#product.cacheEnabled == true",
              unless = "#result.price < 10")
    public ProductDto saveProduct(ProductDto product) {
        // Method implementation
    }
}
```

Distributed Caching with Hazelcast

```xml
<dependency>
    <groupId>com.hazelcast</groupId>
    <artifactId>hazelcast</artifactId>
</dependency>
<dependency>
    <groupId>com.hazelcast</groupId>
    <artifactId>hazelcast-spring</artifactId>
</dependency>
```

```java
@Configuration
@EnableCaching
public class HazelcastCacheConfig {

    @Bean
    public HazelcastInstance hazelcastInstance() {
        Config config = new Config();
        config.setInstanceName("hazelcast-instance")
              .addMapConfig(
                    new MapConfig()
                        .setName("products")
                        .setEvictionConfig(
                            new EvictionConfig()
                                .setEvictionPolicy(EvictionPolicy.LRU)
                                .setMaxSizePolicy(MaxSizePolicy.PER_NODE)
                                .setSize(1000))
                        .setTimeToLiveSeconds(3600))
              .addMapConfig(
                    new MapConfig()
                        .setName("categories")
                        .setTimeToLiveSeconds(7200))
              .addMapConfig(
                    new MapConfig()
                        .setName("product-search")
                        .setTimeToLiveSeconds(600));

        config.getNetworkConfig()
              .setPort(5701)
              .setPortAutoIncrement(true)
              .getJoin()
              .getMulticastConfig()
              .setEnabled(true)
              .setMulticastGroup("224.0.0.1")
              .setMulticastPort(5701);

        return Hazelcast.newHazelcastInstance(config);
    }

    @Bean
    public CacheManager cacheManager(HazelcastInstance hazelcastInstance) {
        return new HazelcastCacheManager(hazelcastInstance);
```

```
    }
}
```

Batch Processing with Spring Batch

Implementing batch processing jobs with Spring Batch for handling large datasets efficiently.

Setting Up Spring Batch

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

```
@Configuration
@EnableBatchProcessing
public class BatchConfig {

    @Autowired
    public JobBuilderFactory jobBuilderFactory;

    @Autowired
    public StepBuilderFactory stepBuilderFactory;
}
```

Creating a Simple Batch Job

```
@Configuration
@EnableBatchProcessing
public class ImportProductsJobConfig {

    private final JobBuilderFactory jobBuilderFactory;
    private final StepBuilderFactory stepBuilderFactory;
    private final ProductRepository productRepository;
    private final CategoryRepository categoryRepository;

    public ImportProductsJobConfig(JobBuilderFactory jobBuilderFactory,
                                   StepBuilderFactory stepBuilderFactory,
                                   ProductRepository productRepository,
                                   CategoryRepository categoryRepository) {
        this.jobBuilderFactory = jobBuilderFactory;
        this.stepBuilderFactory = stepBuilderFactory;
        this.productRepository = productRepository;
        this.categoryRepository = categoryRepository;
    }
```

```
    @Bean
    public FlatFileItemReader<ProductCsvDto> productItemReader(
            @Value("${batch.import.products.file}") Resource resource) {

        return new FlatFileItemReaderBuilder<ProductCsvDto>()
                .name("productItemReader")
                .resource(resource)
                .delimited()
                .names(new String[]{"name", "description", "price",
"categoryName"})
                .fieldSetMapper(new BeanWrapperFieldSetMapper<>() {{
                    setTargetType(ProductCsvDto.class);
                }})
                .linesToSkip(1) // Skip header line
                .build();
    }

    @Bean
    public ProductProcessor productProcessor() {
        return new ProductProcessor(categoryRepository);
    }

    @Bean
    public RepositoryItemWriter<Product> productItemWriter() {
        RepositoryItemWriter<Product> writer = new RepositoryItemWriter<>();
        writer.setRepository(productRepository);
        writer.setMethodName("save");
        return writer;
    }

    @Bean
    public Step importProductsStep(FlatFileItemReader<ProductCsvDto>
productItemReader,
                                   ProductProcessor productProcessor,
                                   RepositoryItemWriter<Product> productItemWriter)
{

        return stepBuilderFactory.get("importProductsStep")
                .<ProductCsvDto, Product>chunk(10)
                .reader(productItemReader)
                .processor(productProcessor)
                .writer(productItemWriter)
                .faultTolerant()
                .skipLimit(10)
                .skip(Exception.class)
                .listener(new ProductImportStepListener())
                .build();
    }

    @Bean
    public Job importProductsJob(JobCompletionNotificationListener listener,
                                 Step importProductsStep) {

        return jobBuilderFactory.get("importProductsJob")
```

```
                .incrementer(new RunIdIncrementer())
                .listener(listener)
                .flow(importProductsStep)
                .end()
                .build();
    }

    @Data
    public static class ProductCsvDto {
        private String name;
        private String description;
        private BigDecimal price;
        private String categoryName;
    }

    public static class ProductProcessor implements ItemProcessor<ProductCsvDto,
Product> {

        private final CategoryRepository categoryRepository;
        private final Map<String, Category> categoryCache = new HashMap<>();

        public ProductProcessor(CategoryRepository categoryRepository) {
            this.categoryRepository = categoryRepository;
        }

        @Override
        public Product process(ProductCsvDto item) throws Exception {
            // Log processing
            System.out.println("Processing product: " + item.getName());

            // Create the product entity
            Product product = new Product();
            product.setName(item.getName());
            product.setDescription(item.getDescription());
            product.setPrice(item.getPrice());

            // Find or create the category
            Category category = getOrCreateCategory(item.getCategoryName());
            product.setCategory(category);

            return product;
        }

        private Category getOrCreateCategory(String categoryName) {
            // Check cache first
            if (categoryCache.containsKey(categoryName)) {
                return categoryCache.get(categoryName);
            }

            // Find existing category
            Optional<Category> existingCategory =
categoryRepository.findByName(categoryName);
            if (existingCategory.isPresent()) {
                Category category = existingCategory.get();
```

```
                categoryCache.put(categoryName, category);
                return category;
            }

            // Create new category
            Category newCategory = new Category();
            newCategory.setName(categoryName);
            newCategory = categoryRepository.save(newCategory);
            categoryCache.put(categoryName, newCategory);

            return newCategory;
        }
    }

    public static class ProductImportStepListener implements StepExecutionListener
{

        private static final Logger logger =
LoggerFactory.getLogger(ProductImportStepListener.class);

        @Override
        public void beforeStep(StepExecution stepExecution) {
            logger.info("Starting product import step");
        }

        @Override
        public ExitStatus afterStep(StepExecution stepExecution) {
            logger.info("Completed product import step: {} items read, {} items
written, {} items skipped",
                    stepExecution.getReadCount(),
                    stepExecution.getWriteCount(),
                    stepExecution.getReadSkipCount() +
stepExecution.getProcessSkipCount() + stepExecution.getWriteSkipCount());

            if (stepExecution.getSkipCount() > 0) {
                return
ExitStatus.COMPLETED.addExitDescription(stepExecution.getSkipCount() + " items
skipped");
            }

            return ExitStatus.COMPLETED;
        }
    }

    @Component
    public static class JobCompletionNotificationListener extends
JobExecutionListenerSupport {

        private static final Logger logger =
LoggerFactory.getLogger(JobCompletionNotificationListener.class);

        private final ProductRepository productRepository;

        public JobCompletionNotificationListener(ProductRepository
```

```
productRepository) {
            this.productRepository = productRepository;
        }

        @Override
        public void afterJob(JobExecution jobExecution) {
            if (jobExecution.getStatus() == BatchStatus.COMPLETED) {
                logger.info("!!! JOB FINISHED! Time to verify the results");

                long count = productRepository.count();
                logger.info("Found {} products in the database", count);
            } else if (jobExecution.getStatus() == BatchStatus.FAILED) {
                logger.error("!!! JOB FAILED! with status: {}",
jobExecution.getStatus());
            }
        }
    }
}
```

Database to Database Job

```
@Configuration
@EnableBatchProcessing
public class ProductMigrationJobConfig {

    private final JobBuilderFactory jobBuilderFactory;
    private final StepBuilderFactory stepBuilderFactory;
    private final EntityManagerFactory entityManagerFactory;
    private final ProductMigrationService migrationService;

    public ProductMigrationJobConfig(JobBuilderFactory jobBuilderFactory,
                                     StepBuilderFactory stepBuilderFactory,
                                     EntityManagerFactory entityManagerFactory,
                                     ProductMigrationService migrationService) {
        this.jobBuilderFactory = jobBuilderFactory;
        this.stepBuilderFactory = stepBuilderFactory;
        this.entityManagerFactory = entityManagerFactory;
        this.migrationService = migrationService;
    }

    @Bean
    public JpaPagingItemReader<LegacyProduct> legacyProductItemReader() {
        return new JpaPagingItemReaderBuilder<LegacyProduct>()
                .name("legacyProductItemReader")
                .entityManagerFactory(entityManagerFactory)
                .queryString("SELECT p FROM LegacyProduct p")
                .pageSize(100)
                .build();
    }

    @Bean
```

```
    public ItemProcessor<LegacyProduct, Product> productMigrationProcessor() {
        return legacy -> {
            Product product = new Product();
            product.setName(legacy.getProductName());
            product.setDescription(legacy.getDescription());
            product.setPrice(legacy.getPrice());

            // Classify product and assign category
            Category category = migrationService.classifyProduct(legacy);
            product.setCategory(category);

            return product;
        };
    }

    @Bean
    public CompositeItemWriter<Product> productCompositeItemWriter(
            JpaItemWriter<Product> jpaItemWriter,
            ElasticsearchItemWriter<Product> esItemWriter) {

        return new CompositeItemWriterBuilder<Product>()
                .delegates(jpaItemWriter, esItemWriter)
                .build();
    }

    @Bean
    public JpaItemWriter<Product> jpaItemWriter() {
        JpaItemWriter<Product> writer = new JpaItemWriter<>();
        writer.setEntityManagerFactory(entityManagerFactory);
        return writer;
    }

    @Bean
    public ElasticsearchItemWriter<Product> elasticsearchItemWriter(
            ElasticsearchOperations elasticsearchOperations) {

        return new ElasticsearchItemWriterBuilder<Product>()
                .operations(elasticsearchOperations)
                .build();
    }

    @Bean
    public Step migrateProductsStep(JpaPagingItemReader<LegacyProduct> reader,
                                    ItemProcessor<LegacyProduct, Product> processor,
                                    CompositeItemWriter<Product> writer) {

        return stepBuilderFactory.get("migrateProductsStep")
                .<LegacyProduct, Product>chunk(50)
                .reader(reader)
                .processor(processor)
                .writer(writer)
                .faultTolerant()
                .retryLimit(3)
                .retry(OptimisticLockingFailureException.class)
```

```
                .skipLimit(10)
                .skip(Exception.class)
                .taskExecutor(taskExecutor())
                .build();
    }

    @Bean
    public TaskExecutor taskExecutor() {
        ThreadPoolTaskExecutor executor = new ThreadPoolTaskExecutor();
        executor.setCorePoolSize(4);
        executor.setMaxPoolSize(8);
        executor.setQueueCapacity(20);
        executor.setThreadNamePrefix("product-migration-");
        return executor;
    }

    @Bean
    public Job migrateProductsJob(Step migrateProductsStep) {
        return jobBuilderFactory.get("migrateProductsJob")
                .incrementer(new RunIdIncrementer())
                .start(migrateProductsStep)
                .listener(new ProductMigrationJobListener())
                .build();
    }

    public static class ProductMigrationJobListener extends
JobExecutionListenerSupport {

        private static final Logger logger =
LoggerFactory.getLogger(ProductMigrationJobListener.class);

        @Override
        public void beforeJob(JobExecution jobExecution) {
            logger.info("Starting product migration job at {}", new Date());
        }

        @Override
        public void afterJob(JobExecution jobExecution) {
            if (jobExecution.getStatus() == BatchStatus.COMPLETED) {
                logger.info("Product migration job completed successfully at {}",
new Date());

                // Log metrics
                long totalTime = jobExecution.getEndTime().getTime() -
jobExecution.getStartTime().getTime();
                long itemCount = 0;
                for (StepExecution stepExecution :
jobExecution.getStepExecutions()) {
                    itemCount += stepExecution.getWriteCount();
                }

                logger.info("Processed {} items in {} ms (avg: {} ms/item)",
                        itemCount, totalTime, itemCount > 0 ? totalTime /
itemCount : 0);
```

```
            } else {
                logger.error("Product migration job failed with status: {}",
jobExecution.getStatus());
            }
        }
    }
}
```

Batch Job with Partitioning

```
@Configuration
@EnableBatchProcessing
public class PartitionedProductProcessingJobConfig {

    private final JobBuilderFactory jobBuilderFactory;
    private final StepBuilderFactory stepBuilderFactory;
    private final ProductRepository productRepository;

    public PartitionedProductProcessingJobConfig(JobBuilderFactory
jobBuilderFactory,
                                                 StepBuilderFactory
stepBuilderFactory,
                                                 ProductRepository productRepository)
{
        this.jobBuilderFactory = jobBuilderFactory;
        this.stepBuilderFactory = stepBuilderFactory;
        this.productRepository = productRepository;
    }

    @Bean
    public ColumnRangePartitioner productIdRangePartitioner() {
        ColumnRangePartitioner partitioner = new ColumnRangePartitioner();
        partitioner.setColumn("id");
        partitioner.setTable("products");
        partitioner.setDataSource(dataSource);
        return partitioner;
    }

    @Bean
    public JdbcPagingItemReader<Product> productItemReader() {
        // This is just a template - actual readers will be created per partition
        return new JdbcPagingItemReaderBuilder<Product>()
                .name("productItemReader")
                .dataSource(dataSource)
                .queryProvider(queryProvider())
                .rowMapper(new ProductRowMapper())
                .pageSize(100)
                .build();
    }

    @Bean
```

```
    public SqlPagingQueryProviderFactoryBean queryProvider() {
        SqlPagingQueryProviderFactoryBean provider = new
SqlPagingQueryProviderFactoryBean();
        provider.setDataSource(dataSource);
        provider.setSelectClause("SELECT id, name, description, price,
category_id");
        provider.setFromClause("FROM products");
        provider.setWhereClause("WHERE id >= :minId AND id <= :maxId");
        provider.setSortKey("id");
        return provider;
    }

    @Bean
    public ProductEnrichmentProcessor productProcessor() {
        return new ProductEnrichmentProcessor();
    }

    @Bean
    public JdbcBatchItemWriter<Product> productItemWriter() {
        return new JdbcBatchItemWriterBuilder<Product>()
                .itemSqlParameterSourceProvider(new
BeanPropertyItemSqlParameterSourceProvider<>())
                .sql("UPDATE products SET description = :description, price =
:price WHERE id = :id")
                .dataSource(dataSource)
                .build();
    }

    @Bean
    public Step productProcessingStep() {
        return stepBuilderFactory.get("productProcessingStep")
                .<Product, Product>chunk(100)
                .reader(productItemReader())
                .processor(productProcessor())
                .writer(productItemWriter())
                .build();
    }

    @Bean
    public Step partitionedProductProcessingStep() {
        return stepBuilderFactory.get("partitionedProductProcessingStep")
                .partitioner("productProcessingStep", productIdRangePartitioner())
                .step(productProcessingStep())
                .gridSize(4)
                .taskExecutor(taskExecutor())
                .build();
    }

    @Bean
    public TaskExecutor taskExecutor() {
        ThreadPoolTaskExecutor executor = new ThreadPoolTaskExecutor();
        executor.setCorePoolSize(4);
        executor.setMaxPoolSize(8);
        executor.setQueueCapacity(20);
```

```
        executor.setThreadNamePrefix("product-processing-");
        return executor;
    }

    @Bean
    public Job partitionedProductProcessingJob(Step
partitionedProductProcessingStep) {
        return jobBuilderFactory.get("partitionedProductProcessingJob")
                .incrementer(new RunIdIncrementer())
                .start(partitionedProductProcessingStep)
                .build();
    }

    public static class ProductRowMapper implements RowMapper<Product> {
        @Override
        public Product mapRow(ResultSet rs, int rowNum) throws SQLException {
            Product product = new Product();
            product.setId(rs.getLong("id"));
            product.setName(rs.getString("name"));
            product.setDescription(rs.getString("description"));
            product.setPrice(rs.getBigDecimal("price"));

            // You would need to set the category separately or using a join

            return product;
        }
    }

    public static class ProductEnrichmentProcessor implements
ItemProcessor<Product, Product> {

        private final ApiClient apiClient;

        public ProductEnrichmentProcessor() {
            this.apiClient = new ApiClient();
        }

        @Override
        public Product process(Product product) throws Exception {
            // Enrich product with additional information from an external API
            String enrichedDescription =
apiClient.getProductDetails(product.getName());
            if (enrichedDescription != null && !enrichedDescription.isEmpty()) {
                product.setDescription(product.getDescription() + " " +
enrichedDescription);
            }

            // Calculate a special price discount based on some business logic
            if (product.getPrice().compareTo(new BigDecimal(100)) > 0) {
                // Apply 10% discount for products over $100
                BigDecimal discountedPrice = product.getPrice()
                        .multiply(new BigDecimal("0.9"))
                        .setScale(2, RoundingMode.HALF_UP);
                product.setPrice(discountedPrice);
```

```
            }

            return product;
        }
    }

    public static class ApiClient {
        public String getProductDetails(String productName) {
            // Simulate API call to get additional product details
            return "Additional details for " + productName;
        }
    }

    public static class ColumnRangePartitioner implements Partitioner {

        private JdbcTemplate jdbcTemplate;
        private String table;
        private String column;

        @Override
        public Map<String, ExecutionContext> partition(int gridSize) {
            int min = jdbcTemplate.queryForObject(
                    "SELECT MIN(" + column + ") FROM " + table, Integer.class);
            int max = jdbcTemplate.queryForObject(
                    "SELECT MAX(" + column + ") FROM " + table, Integer.class);
            int targetSize = (max - min) / gridSize + 1;

            Map<String, ExecutionContext> result = new HashMap<>();
            int number = 0;
            int start = min;
            int end = start + targetSize - 1;

            while (start <= max) {
                ExecutionContext value = new ExecutionContext();
                result.put("partition" + number, value);

                value.putInt("minId", start);
                value.putInt("maxId", end);

                start += targetSize;
                end += targetSize;

                number++;
            }

            return result;
        }

        public void setDataSource(DataSource dataSource) {
            this.jdbcTemplate = new JdbcTemplate(dataSource);
        }

        public void setTable(String table) {
            this.table = table;
```

```
        }

        public void setColumn(String column) {
            this.column = column;
        }
    }
}
```

Job Scheduling and Triggers

```
@Configuration
@EnableBatchProcessing
@EnableScheduling
public class BatchJobSchedulingConfig {

    private final JobLauncher jobLauncher;
    private final Job importProductsJob;
    private final Job migrateProductsJob;

    public BatchJobSchedulingConfig(JobLauncher jobLauncher,
                                    @Qualifier("importProductsJob") Job
importProductsJob,
                                    @Qualifier("migrateProductsJob") Job
migrateProductsJob) {
        this.jobLauncher = jobLauncher;
        this.importProductsJob = importProductsJob;
        this.migrateProductsJob = migrateProductsJob;
    }

    @Scheduled(cron = "0 0 1 * * ?") // Every day at 1 AM
    public void scheduleImportProductsJob() throws Exception {
        JobParameters params = new JobParametersBuilder()
                .addString("JobID", String.valueOf(System.currentTimeMillis()))
                .addString("fileName", "products_" + LocalDate.now() + ".csv")
                .toJobParameters();

        jobLauncher.run(importProductsJob, params);
    }

    @Scheduled(cron = "0 0 3 * * 0") // Every Sunday at 3 AM
    public void scheduleMigrateProductsJob() throws Exception {
        JobParameters params = new JobParametersBuilder()
                .addString("JobID", String.valueOf(System.currentTimeMillis()))
                .addDate("date", new Date())
                .toJobParameters();

        jobLauncher.run(migrateProductsJob, params);
    }

    @Bean
    public JobRegistryBeanPostProcessor jobRegistryBeanPostProcessor(JobRegistry
```

```
jobRegistry) {
        JobRegistryBeanPostProcessor postProcessor = new
JobRegistryBeanPostProcessor();
        postProcessor.setJobRegistry(jobRegistry);
        return postProcessor;
    }

    @Bean
    public JobRegistry jobRegistry() {
        return new MapJobRegistry();
    }

    @Bean
    public JobOperator jobOperator(JobRegistry jobRegistry, JobRepository
jobRepository) {
        SimpleJobOperator jobOperator = new SimpleJobOperator();
        jobOperator.setJobRegistry(jobRegistry);
        jobOperator.setJobRepository(jobRepository);
        jobOperator.setJobLauncher(jobLauncher);
        jobOperator.setJobExplorer(jobExplorer(jobRepository));
        return jobOperator;
    }

    @Bean
    public JobExplorer jobExplorer(JobRepository jobRepository) {
        return new SimpleJobExplorerFactoryBean() {{
            setJobRepository(jobRepository);
        }}.getObject();
    }
}
```

Batch Processing REST API

```
@RestController
@RequestMapping("/api/batch")
public class BatchJobController {

    private final JobLauncher jobLauncher;
    private final JobRegistry jobRegistry;
    private final JobOperator jobOperator;
    private final JobExplorer jobExplorer;

    public BatchJobController(JobLauncher jobLauncher,
                              JobRegistry jobRegistry,
                              JobOperator jobOperator,
                              JobExplorer jobExplorer) {
        this.jobLauncher = jobLauncher;
        this.jobRegistry = jobRegistry;
        this.jobOperator = jobOperator;
        this.jobExplorer = jobExplorer;
    }
```

```
    @GetMapping("/jobs")
    public List<String> listJobs() {
        return
jobRegistry.getJobNames().stream().sorted().collect(Collectors.toList());
    }

    @PostMapping("/jobs/{jobName}")
    public ResponseEntity<Map<String, String>> launchJob(@PathVariable String
jobName,
                                                         @RequestBody(required =
false) Map<String, String> jobParams) {

        try {
            Job job = jobRegistry.getJob(jobName);
            JobParametersBuilder jobParametersBuilder = new
JobParametersBuilder();

            // Add timestamp to make parameters unique
            jobParametersBuilder.addLong("timestamp", System.currentTimeMillis());

            // Add custom parameters if provided
            if (jobParams != null) {
                jobParams.forEach((key, value) -> {
                    if (value.matches("\\d+")) {
                        jobParametersBuilder.addLong(key, Long.parseLong(value));
                    } else if (value.matches("\\d{4}-\\d{2}-\\d{2}")) {
                        try {
                            Date date = new SimpleDateFormat("yyyy-MM-
dd").parse(value);
                            jobParametersBuilder.addDate(key, date);
                        } catch (ParseException e) {
                            jobParametersBuilder.addString(key, value);
                        }
                    } else {
                        jobParametersBuilder.addString(key, value);
                    }
                });
            }

            JobExecution jobExecution = jobLauncher.run(job,
jobParametersBuilder.toJobParameters());

            Map<String, String> response = new HashMap<>();
            response.put("jobExecutionId", String.valueOf(jobExecution.getId()));
            response.put("status", jobExecution.getStatus().toString());

            return ResponseEntity.ok(response);
        } catch (JobInstanceAlreadyExistsException e) {
            Map<String, String> response = new HashMap<>();
            response.put("error", "Job instance already exists");
            return ResponseEntity.badRequest().body(response);
        } catch (NoSuchJobException e) {
            return ResponseEntity.notFound().build();
```

```
        } catch (JobParametersInvalidException e) {
            Map<String, String> response = new HashMap<>();
            response.put("error", "Invalid job parameters: " + e.getMessage());
            return ResponseEntity.badRequest().body(response);
        } catch (Exception e) {
            Map<String, String> response = new HashMap<>();
            response.put("error", "Failed to launch job: " + e.getMessage());
            return
ResponseEntity.status(HttpStatus.INTERNAL_SERVER_ERROR).body(response);
        }
    }

    @GetMapping("/jobs/executions")
    public List<Map<String, Object>> getJobExecutions() {
        List<JobExecution> jobExecutions = new ArrayList<>();
        jobExplorer.getJobNames().forEach(jobName -> {
            for (JobInstance jobInstance :
jobExplorer.findJobInstancesByJobName(jobName, 0, 10)) {
                jobExecutions.addAll(jobExplorer.getJobExecutions(jobInstance));
            }
        });

        // Sort by start time, most recent first
        jobExecutions.sort(Comparator.comparing(JobExecution::getStartTime,
Comparator.nullsLast(Comparator.reverseOrder())));

        return jobExecutions.stream()
                .limit(50)  // Limit to most recent 50
                .map(this::mapJobExecutionToResponse)
                .collect(Collectors.toList());
    }

    @GetMapping("/jobs/executions/{executionId}")
    public ResponseEntity<Map<String, Object>> getJobExecution(@PathVariable Long
executionId) {
        JobExecution jobExecution = jobExplorer.getJobExecution(executionId);
        if (jobExecution == null) {
            return ResponseEntity.notFound().build();
        }

        return ResponseEntity.ok(mapJobExecutionToResponse(jobExecution));
    }

    @DeleteMapping("/jobs/executions/{executionId}")
    public ResponseEntity<Void> stopJobExecution(@PathVariable Long executionId) {
        try {
            jobOperator.stop(executionId);
            return ResponseEntity.noContent().build();
        } catch (NoSuchJobExecutionException | JobExecutionNotRunningException e)
{
            return ResponseEntity.notFound().build();
        }
    }
```

```
    @PostMapping("/jobs/executions/{executionId}/restart")
    public ResponseEntity<Map<String, String>> restartJobExecution(@PathVariable
Long executionId) {
        try {
            Long restartedExecutionId = jobOperator.restart(executionId);

            Map<String, String> response = new HashMap<>();
            response.put("restartedJobExecutionId",
String.valueOf(restartedExecutionId));

            return ResponseEntity.ok(response);
        } catch (JobInstanceAlreadyCompleteException | NoSuchJobExecutionException
|
                NoSuchJobException | JobRestartException |
JobParametersInvalidException e) {
            Map<String, String> response = new HashMap<>();
            response.put("error", "Failed to restart job: " + e.getMessage());
            return ResponseEntity.badRequest().body(response);
        }
    }

    private Map<String, Object> mapJobExecutionToResponse(JobExecution
jobExecution) {
        Map<String, Object> response = new HashMap<>();
        response.put("id", jobExecution.getId());
        response.put("jobName", jobExecution.getJobInstance().getJobName());
        response.put("startTime", jobExecution.getStartTime());
        response.put("endTime", jobExecution.getEndTime());
        response.put("status", jobExecution.getStatus().toString());
        response.put("exitCode", jobExecution.getExitStatus().getExitCode());
        response.put("exitDescription",
jobExecution.getExitStatus().getExitDescription());

        // Add parameters
        Map<String, Object> parameters = new HashMap<>();
        jobExecution.getJobParameters().getParameters().forEach((key, value) ->
                parameters.put(key, value.getValue()));
        response.put("parameters", parameters);

        // Add step executions
        List<Map<String, Object>> stepExecutions = new ArrayList<>();
        for (StepExecution stepExecution : jobExecution.getStepExecutions()) {
            Map<String, Object> step = new HashMap<>();
            step.put("id", stepExecution.getId());
            step.put("stepName", stepExecution.getStepName());
            step.put("status", stepExecution.getStatus().toString());
            step.put("readCount", stepExecution.getReadCount());
            step.put("writeCount", stepExecution.getWriteCount());
            step.put("skipCount", stepExecution.getSkipCount());
            step.put("startTime", stepExecution.getStartTime());
            step.put("endTime", stepExecution.getEndTime());
            stepExecutions.add(step);
        }
        response.put("stepExecutions", stepExecutions);
```

```
        return response;
    }
}
```

Scheduled Tasks and Async Processing

Implementing scheduled tasks and asynchronous processing for improved performance and resource utilization.

Scheduling Tasks

```
@Configuration
@EnableScheduling
public class SchedulingConfig {

    @Bean
    public TaskScheduler taskScheduler() {
        ThreadPoolTaskScheduler scheduler = new ThreadPoolTaskScheduler();
        scheduler.setPoolSize(5);
        scheduler.setThreadNamePrefix("scheduled-task-");
        scheduler.setErrorHandler(t -> {
            System.err.println("Error in scheduled task: " + t.getMessage());
            t.printStackTrace();
        });
        return scheduler;
    }
}
```

```
@Component
public class ProductScheduledTasks {

    private static final Logger logger =
LoggerFactory.getLogger(ProductScheduledTasks.class);

    private final ProductService productService;
    private final PriceService priceService;
    private final InventoryService inventoryService;
    private final NotificationService notificationService;

    public ProductScheduledTasks(ProductService productService,
                                 PriceService priceService,
                                 InventoryService inventoryService,
                                 NotificationService notificationService) {
        this.productService = productService;
        this.priceService = priceService;
        this.inventoryService = inventoryService;
        this.notificationService = notificationService;
    }
```

```
    @Scheduled(fixedRate = 3600000) // Every hour
    public void refreshProductPricing() {
        logger.info("Starting scheduled product pricing refresh at {}", new
Date());

        try {
            int updatedCount = priceService.updateProductPrices();
            logger.info("Completed product pricing refresh: {} products updated",
updatedCount);
        } catch (Exception e) {
            logger.error("Error during product pricing refresh", e);
        }
    }

    @Scheduled(cron = "0 0 2 * * ?") // Every day at 2 AM
    public void generateDailyProductReport() {
        logger.info("Starting daily product report generation at {}", new Date());

        try {
            String reportPath = productService.generateDailyReport();
            notificationService.sendReportNotification(reportPath);
            logger.info("Daily product report generated successfully: {}",
reportPath);
        } catch (Exception e) {
            logger.error("Error generating daily product report", e);
            notificationService.sendErrorNotification("Failed to generate daily
product report: " + e.getMessage());
        }
    }

    @Scheduled(fixedDelay = 900000) // Every 15 minutes
    public void checkLowInventory() {
        logger.info("Checking for low inventory products at {}", new Date());

        try {
            List<ProductInventoryDto> lowStockProducts =
inventoryService.findLowStockProducts();

            if (!lowStockProducts.isEmpty()) {
                logger.info("Found {} products with low inventory",
lowStockProducts.size());
                notificationService.sendLowInventoryAlert(lowStockProducts);
            } else {
                logger.info("No products with low inventory found");
            }
        } catch (Exception e) {
            logger.error("Error checking for low inventory products", e);
        }
    }

    @Scheduled(initialDelay = 60000, fixedRate = 3600000) // 1 minute after
startup, then hourly
    public void syncProductCatalog() {
```

```
        logger.info("Starting product catalog synchronization at {}", new Date());

        try {
            int addedCount = productService.syncExternalProductCatalog();
            logger.info("Product catalog synchronized: {} new products added",
addedCount);
        } catch (Exception e) {
            logger.error("Error synchronizing product catalog", e);
        }
    }

    // Scheduled rate based on property value
    @Scheduled(fixedRateString = "${app.product-cache.refresh-rate-ms:1800000}")
    public void refreshProductCache() {
        logger.info("Refreshing product cache at {}", new Date());

        try {
            productService.refreshCache();
            logger.info("Product cache refreshed successfully");
        } catch (Exception e) {
            logger.error("Error refreshing product cache", e);
        }
    }
}
```

Dynamic Task Scheduling

```
@Service
public class DynamicTaskScheduler {

    private final TaskScheduler taskScheduler;
    private final Map<String, ScheduledFuture<?>> scheduledTasks = new
ConcurrentHashMap<>();

    public DynamicTaskScheduler(TaskScheduler taskScheduler) {
        this.taskScheduler = taskScheduler;
    }

    public void scheduleAtFixedRate(String taskId, Runnable task, long fixedRate)
{
        ScheduledFuture<?> scheduledTask = taskScheduler.scheduleAtFixedRate(task,
fixedRate);
        ScheduledFuture<?> existingTask = scheduledTasks.put(taskId,
scheduledTask);

        if (existingTask != null) {
            existingTask.cancel(false);
        }
    }

    public void scheduleWithCronExpression(String taskId, Runnable task, String
```

```
cronExpression) {
        ScheduledFuture<?> scheduledTask = taskScheduler.schedule(
                task, new CronTrigger(cronExpression));
        ScheduledFuture<?> existingTask = scheduledTasks.put(taskId,
scheduledTask);

        if (existingTask != null) {
            existingTask.cancel(false);
        }
    }

    public boolean cancelTask(String taskId) {
        ScheduledFuture<?> scheduledTask = scheduledTasks.remove(taskId);

        if (scheduledTask != null) {
            return scheduledTask.cancel(false);
        }

        return false;
    }

    public Set<String> getScheduledTaskIds() {
        return scheduledTasks.keySet();
    }

    public boolean isTaskScheduled(String taskId) {
        ScheduledFuture<?> scheduledTask = scheduledTasks.get(taskId);
        return scheduledTask != null && !scheduledTask.isCancelled() &&
!scheduledTask.isDone();
    }
}
```

```
@RestController
@RequestMapping("/api/tasks")
public class ScheduledTaskController {

    private final DynamicTaskScheduler taskScheduler;
    private final ProductService productService;

    public ScheduledTaskController(DynamicTaskScheduler taskScheduler,
                                   ProductService productService) {
        this.taskScheduler = taskScheduler;
        this.productService = productService;
    }

    @GetMapping
    public Set<String> getScheduledTasks() {
        return taskScheduler.getScheduledTaskIds();
    }

    @PostMapping("/product-sync")
```

```
    public ResponseEntity<Map<String, String>> scheduleProductSync(
            @RequestParam String cronExpression) {

        taskScheduler.scheduleWithCronExpression(
                "product-sync",
                () -> productService.syncExternalProductCatalog(),
                cronExpression);

        Map<String, String> response = new HashMap<>();
        response.put("taskId", "product-sync");
        response.put("status", "scheduled");
        response.put("cronExpression", cronExpression);

        return ResponseEntity.ok(response);
    }

    @PostMapping("/cache-refresh")
    public ResponseEntity<Map<String, String>> scheduleProductCacheRefresh(
            @RequestParam long fixedRateMs) {

        taskScheduler.scheduleAtFixedRate(
                "cache-refresh",
                () -> productService.refreshCache(),
                fixedRateMs);

        Map<String, String> response = new HashMap<>();
        response.put("taskId", "cache-refresh");
        response.put("status", "scheduled");
        response.put("fixedRateMs", String.valueOf(fixedRateMs));

        return ResponseEntity.ok(response);
    }

    @DeleteMapping("/{taskId}")
    public ResponseEntity<Map<String, String>> cancelTask(@PathVariable String
taskId) {
        boolean cancelled = taskScheduler.cancelTask(taskId);

        if (cancelled) {
            Map<String, String> response = new HashMap<>();
            response.put("taskId", taskId);
            response.put("status", "cancelled");

            return ResponseEntity.ok(response);
        } else {
            return ResponseEntity.notFound().build();
        }
    }
}
```

Asynchronous Processing

```
@Configuration
@EnableAsync
public class AsyncConfig {

    @Bean
    public Executor asyncExecutor() {
        ThreadPoolTaskExecutor executor = new ThreadPoolTaskExecutor();
        executor.setCorePoolSize(5);
        executor.setMaxPoolSize(10);
        executor.setQueueCapacity(25);
        executor.setThreadNamePrefix("Async-");
        executor.setRejectedExecutionHandler(new
ThreadPoolExecutor.CallerRunsPolicy());
        executor.initialize();
        return executor;
    }
}
```

```
@Service
public class ProductServiceAsync {

    private static final Logger logger =
LoggerFactory.getLogger(ProductServiceAsync.class);

    private final ProductRepository productRepository;
    private final ImageService imageService;
    private final PriceService priceService;
    private final NotificationService notificationService;

    public ProductServiceAsync(ProductRepository productRepository,
                               ImageService imageService,
                               PriceService priceService,
                               NotificationService notificationService) {
        this.productRepository = productRepository;
        this.imageService = imageService;
        this.priceService = priceService;
        this.notificationService = notificationService;
    }

    @Async
    public CompletableFuture<ProductDto> createProductAsync(ProductDto productDto,
                                                             MultipartFile imageFile)
{
        logger.info("Processing product creation asynchronously: {}",
productDto.getName());

        try {
            // First save the product
            Product product = mapToEntity(productDto);
            Product savedProduct = productRepository.save(product);
```

```
            // Process image in parallel if provided
            CompletableFuture<String> imageUrlFuture =
                    (imageFile != null && !imageFile.isEmpty())
                            ?
imageService.uploadProductImageAsync(savedProduct.getId(), imageFile)
                            : CompletableFuture.completedFuture(null);

            // Calculate competitive price in parallel
            CompletableFuture<BigDecimal> priceFuture =

priceService.calculateCompetitivePriceAsync(productDto.getName(),
productDto.getPrice());

            // Wait for both operations to complete
            CompletableFuture.allOf(imageUrlFuture, priceFuture).join();

            // Update product with image and price
            String imageUrl = imageUrlFuture.get();
            BigDecimal competitivePrice = priceFuture.get();

            // Only update if necessary
            boolean needsUpdate = false;

            if (imageUrl != null) {
                savedProduct.setImageUrl(imageUrl);
                needsUpdate = true;
            }

            if (competitivePrice != null &&
                    !competitivePrice.equals(savedProduct.getPrice())) {
                savedProduct.setPriceHistory(
                        savedProduct.getPriceHistory() + "," +
savedProduct.getPrice());
                savedProduct.setPrice(competitivePrice);
                needsUpdate = true;
            }

            if (needsUpdate) {
                savedProduct = productRepository.save(savedProduct);
            }

            // Notify interested parties
            notificationService.notifyProductCreated(savedProduct);

            return CompletableFuture.completedFuture(mapToDto(savedProduct));
        } catch (Exception e) {
            logger.error("Error during async product creation", e);
            throw new ProductProcessingException("Failed to process product
asynchronously", e);
        }
    }

    @Async
    public CompletableFuture<List<ProductDto>> findAllAsync() {
```

```
        logger.info("Fetching all products asynchronously");

        List<ProductDto> products = productRepository.findAll().stream()
                .map(this::mapToDto)
                .collect(Collectors.toList());

        return CompletableFuture.completedFuture(products);
    }

    @Async
    public CompletableFuture<List<ProductDto>> searchProductsAsync(
            String keyword, BigDecimal minPrice, BigDecimal maxPrice) {

        logger.info("Searching products asynchronously with criteria: keyword={},
minPrice={}, maxPrice={}",
                keyword, minPrice, maxPrice);

        // Create a list to store query results from different repositories
        List<CompletableFuture<List<Product>>> futures = new ArrayList<>();

        // Search by keyword
        if (keyword != null && !keyword.trim().isEmpty()) {
            futures.add(CompletableFuture.supplyAsync(() ->
                    productRepository.findByNameContainingIgnoreCase(keyword)));
        }

        // Search by price range
        if (minPrice != null && maxPrice != null) {
            futures.add(CompletableFuture.supplyAsync(() ->
                    productRepository.findByPriceBetween(minPrice, maxPrice)));
        } else if (minPrice != null) {
            futures.add(CompletableFuture.supplyAsync(() ->
                    productRepository.findByPriceGreaterThanEqual(minPrice)));
        } else if (maxPrice != null) {
            futures.add(CompletableFuture.supplyAsync(() ->
                    productRepository.findByPriceLessThanEqual(maxPrice)));
        }

        // If no search criteria, get all products
        if (futures.isEmpty()) {
            futures.add(CompletableFuture.supplyAsync(() ->
                    productRepository.findAll()));
        }

        // Wait for all search operations to complete
        CompletableFuture<Void> allFutures = CompletableFuture.allOf(
                futures.toArray(new CompletableFuture[0]));

        // Combine the results
        return allFutures.thenApply(v -> {
            // If multiple search criteria, find the intersection
            if (futures.size() > 1) {
                Set<Long> commonIds = new HashSet<>();
                boolean firstSet = true;
```

```
                for (CompletableFuture<List<Product>> future : futures) {
                    Set<Long> ids = future.join().stream()
                            .map(Product::getId)
                            .collect(Collectors.toSet());

                    if (firstSet) {
                        commonIds.addAll(ids);
                        firstSet = false;
                    } else {
                        commonIds.retainAll(ids);
                    }
                }

                // Fetch products by common IDs
                List<Product> products = productRepository.findAllById(commonIds);
                return products.stream()
                        .map(this::mapToDto)
                        .collect(Collectors.toList());
            } else {
                // Only one search criteria, use its results directly
                return futures.get(0).join().stream()
                        .map(this::mapToDto)
                        .collect(Collectors.toList());
            }
        });
    }

    @Async
    public CompletableFuture<Void> bulkUpdatePricesAsync(Map<Long, BigDecimal>
priceUpdates) {
        logger.info("Starting bulk price update for {} products",
priceUpdates.size());

        List<CompletableFuture<Void>> updateFutures = new ArrayList<>();

        for (Map.Entry<Long, BigDecimal> entry : priceUpdates.entrySet()) {
            Long productId = entry.getKey();
            BigDecimal newPrice = entry.getValue();

            CompletableFuture<Void> updateFuture = CompletableFuture.runAsync(() -
> {
                try {
                    logger.debug("Updating price for product ID {}: {}",
productId, newPrice);
                    productRepository.findById(productId).ifPresent(product -> {
                        product.setPrice(newPrice);
                        productRepository.save(product);
                    });
                } catch (Exception e) {
                    logger.error("Error updating price for product ID " +
productId, e);
                    throw new CompletionException(e);
                }
```

```
            });

            updateFutures.add(updateFuture);
        }

        // Wait for all updates to complete
        CompletableFuture<Void> allUpdates = CompletableFuture.allOf(
                updateFutures.toArray(new CompletableFuture[0]));

        return allUpdates.thenRun(() ->
                logger.info("Completed bulk price update for {} products",
priceUpdates.size()));
    }

    // Helper methods
    private Product mapToEntity(ProductDto dto) {
        // Implementation
        return new Product();
    }

    private ProductDto mapToDto(Product entity) {
        // Implementation
        return new ProductDto();
    }
}
```

```
@RestController
@RequestMapping("/api/products/async")
public class AsyncProductController {

    private final ProductServiceAsync productServiceAsync;

    public AsyncProductController(ProductServiceAsync productServiceAsync) {
        this.productServiceAsync = productServiceAsync;
    }

    @GetMapping
    public CompletableFuture<ResponseEntity<List<ProductDto>>>
getAllProductsAsync() {
        return productServiceAsync.findAllAsync()
                .thenApply(ResponseEntity::ok);
    }

    @GetMapping("/search")
    public CompletableFuture<ResponseEntity<List<ProductDto>>>
searchProductsAsync(
            @RequestParam(required = false) String keyword,
            @RequestParam(required = false) BigDecimal minPrice,
            @RequestParam(required = false) BigDecimal maxPrice) {

        return productServiceAsync.searchProductsAsync(keyword, minPrice,
```

```
maxPrice)
                .thenApply(ResponseEntity::ok);
    }

    @PostMapping(consumes = MediaType.MULTIPART_FORM_DATA_VALUE)
    public CompletableFuture<ResponseEntity<ProductDto>> createProductAsync(
            @RequestPart("product") ProductDto productDto,
            @RequestPart(value = "image", required = false) MultipartFile
imageFile) {

        return productServiceAsync.createProductAsync(productDto, imageFile)
                .thenApply(savedProduct -> {
                    URI location =
ServletUriComponentsBuilder.fromCurrentRequest()
                            .path("/{id}")
                            .buildAndExpand(savedProduct.getId())
                            .toUri();

                    return ResponseEntity.created(location).body(savedProduct);
                });
    }

    @PutMapping("/bulk-price-update")
    public CompletableFuture<ResponseEntity<Void>> bulkUpdatePricesAsync(
            @RequestBody Map<Long, BigDecimal> priceUpdates) {

        return productServiceAsync.bulkUpdatePricesAsync(priceUpdates)
                .thenApply(v -> ResponseEntity.noContent().build());
    }

    @ExceptionHandler(ProductProcessingException.class)
    public ResponseEntity<ErrorResponse>
handleProductProcessingException(ProductProcessingException ex) {
        ErrorResponse errorResponse = new ErrorResponse();
        errorResponse.setStatus(HttpStatus.INTERNAL_SERVER_ERROR.value());
        errorResponse.setMessage(ex.getMessage());
        errorResponse.setTimestamp(LocalDateTime.now());

        return
ResponseEntity.status(HttpStatus.INTERNAL_SERVER_ERROR).body(errorResponse);
    }

    @Data
    public static class ErrorResponse {
        private int status;
        private String message;
        private LocalDateTime timestamp;
    }

    public static class ProductProcessingException extends RuntimeException {
        public ProductProcessingException(String message, Throwable cause) {
            super(message, cause);
        }
```

```
    }
}
```

Event-Driven Processing

```
@Configuration
public class AsyncEventsConfig {

    @Bean(name = "applicationEventMulticaster")
    public ApplicationEventMulticaster simpleApplicationEventMulticaster() {
        SimpleApplicationEventMulticaster eventMulticaster = new
SimpleApplicationEventMulticaster();
        eventMulticaster.setTaskExecutor(new SimpleAsyncTaskExecutor());
        return eventMulticaster;
    }
}
```

```
public class ProductCreatedEvent extends ApplicationEvent {

    private final ProductDto product;

    public ProductCreatedEvent(Object source, ProductDto product) {
        super(source);
        this.product = product;
    }

    public ProductDto getProduct() {
        return product;
    }
}

public class ProductUpdatedEvent extends ApplicationEvent {

    private final ProductDto product;
    private final ProductDto oldProduct;

    public ProductUpdatedEvent(Object source, ProductDto product, ProductDto
oldProduct) {
        super(source);
        this.product = product;
        this.oldProduct = oldProduct;
    }

    public ProductDto getProduct() {
        return product;
    }

    public ProductDto getOldProduct() {
        return oldProduct;
```

```
    }
}

public class ProductDeletedEvent extends ApplicationEvent {

    private final Long productId;

    public ProductDeletedEvent(Object source, Long productId) {
        super(source);
        this.productId = productId;
    }

    public Long getProductId() {
        return productId;
    }
}
```

```
@Service
public class EventDrivenProductService implements ProductService {

    private static final Logger logger =
LoggerFactory.getLogger(EventDrivenProductService.class);

    private final ProductRepository productRepository;
    private final CategoryRepository categoryRepository;
    private final ApplicationEventPublisher eventPublisher;

    public EventDrivenProductService(ProductRepository productRepository,
                                     CategoryRepository categoryRepository,
                                     ApplicationEventPublisher eventPublisher) {
        this.productRepository = productRepository;
        this.categoryRepository = categoryRepository;
        this.eventPublisher = eventPublisher;
    }

    @Override
    @Transactional
    public ProductDto createProduct(ProductDto productDto) {
        Category category =
categoryRepository.findById(productDto.getCategoryId())
                .orElseThrow(() -> new ResourceNotFoundException(
                        "Category not found with id: " +
productDto.getCategoryId()));

        Product product = mapToEntity(productDto);
        product.setCategory(category);

        Product savedProduct = productRepository.save(product);
        ProductDto savedDto = mapToDto(savedProduct);

        // Publish event
```

```
        eventPublisher.publishEvent(new ProductCreatedEvent(this, savedDto));

        return savedDto;
    }

    @Override
    @Transactional
    public ProductDto updateProduct(Long id, ProductDto productDto) {
        return productRepository.findById(id)
                .map(existingProduct -> {
                    // Store old state for event
                    ProductDto oldProductDto = mapToDto(existingProduct);

                    Category category =
categoryRepository.findById(productDto.getCategoryId())
                            .orElseThrow(() -> new ResourceNotFoundException(
                                    "Category not found with id: " +
productDto.getCategoryId()));

                    existingProduct.setName(productDto.getName());
                    existingProduct.setDescription(productDto.getDescription());
                    existingProduct.setPrice(productDto.getPrice());
                    existingProduct.setCategory(category);

                    Product updatedProduct =
productRepository.save(existingProduct);
                    ProductDto updatedDto = mapToDto(updatedProduct);

                    // Publish event
                    eventPublisher.publishEvent(new ProductUpdatedEvent(this,
updatedDto, oldProductDto));

                    return updatedDto;
                })
                .orElseThrow(() -> new ResourceNotFoundException("Product not
found with id: " + id));
    }

    @Override
    @Transactional
    public void deleteProduct(Long id) {
        Product product = productRepository.findById(id)
                .orElseThrow(() -> new ResourceNotFoundException("Product not
found with id: " + id));

        productRepository.delete(product);

        // Publish event
        eventPublisher.publishEvent(new ProductDeletedEvent(this, id));
    }

    // Other ProductService methods...

    // Helper methods
```

```
    private Product mapToEntity(ProductDto dto) {
        // Implementation
        return new Product();
    }

    private ProductDto mapToDto(Product entity) {
        // Implementation
        return new ProductDto();
    }
}
```

```
@Component
public class ProductEventListeners {

    private static final Logger logger =
LoggerFactory.getLogger(ProductEventListeners.class);

    private final SearchIndexService searchIndexService;
    private final CacheService cacheService;
    private final InventoryService inventoryService;
    private final NotificationService notificationService;

    public ProductEventListeners(SearchIndexService searchIndexService,
                                 CacheService cacheService,
                                 InventoryService inventoryService,
                                 NotificationService notificationService) {
        this.searchIndexService = searchIndexService;
        this.cacheService = cacheService;
        this.inventoryService = inventoryService;
        this.notificationService = notificationService;
    }

    @EventListener
    public void handleProductCreatedEvent(ProductCreatedEvent event) {
        ProductDto product = event.getProduct();
        logger.info("Handling product created event for product: {}",
product.getName());

        try {
            // Add to search index
            searchIndexService.indexProduct(product);

            // Initialize inventory
            inventoryService.initializeInventory(product.getId(), 0);

            // Invalidate related caches
            cacheService.evictCachesByCategory(product.getCategoryId());

            // Send notifications
            notificationService.notifyProductCreated(product);
        } catch (Exception e) {
```

```
            logger.error("Error handling product created event", e);
        }
    }

    @EventListener
    public void handleProductUpdatedEvent(ProductUpdatedEvent event) {
        ProductDto product = event.getProduct();
        ProductDto oldProduct = event.getOldProduct();
        logger.info("Handling product updated event for product: {}",
product.getName());

        try {
            // Update search index
            searchIndexService.updateProductIndex(product);

            // Invalidate related caches
            cacheService.evictProductCache(product.getId());

            if (!product.getCategoryId().equals(oldProduct.getCategoryId())) {
                // Category changed, update category caches
                cacheService.evictCachesByCategory(oldProduct.getCategoryId());
                cacheService.evictCachesByCategory(product.getCategoryId());
            }

            // Check for price changes
            if (!product.getPrice().equals(oldProduct.getPrice())) {
                notificationService.notifyPriceChanged(product,
oldProduct.getPrice());
            }
        } catch (Exception e) {
            logger.error("Error handling product updated event", e);
        }
    }

    @EventListener
    public void handleProductDeletedEvent(ProductDeletedEvent event) {
        Long productId = event.getProductId();
        logger.info("Handling product deleted event for product ID: {}",
productId);

        try {
            // Remove from search index
            searchIndexService.removeProductFromIndex(productId);

            // Remove inventory
            inventoryService.removeInventory(productId);

            // Invalidate related caches
            cacheService.evictProductCache(productId);
            cacheService.evictAllProductCollectionCaches();

            // Send notifications
            notificationService.notifyProductDeleted(productId);
        } catch (Exception e) {
```

```
            logger.error("Error handling product deleted event", e);
        }
    }
}
```

GraphQL with Spring Boot

Implementing GraphQL APIs for flexible and efficient data querying.

Setting Up GraphQL

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

```
# src/main/resources/graphql/schema.graphqls
type Query {
  products(limit: Int, offset: Int): [Product]
  product(id: ID!): Product
  categories: [Category]
  category(id: ID!): Category
  searchProducts(keyword: String, minPrice: Float, maxPrice: Float): [Product]
}

type Mutation {
  createProduct(input: ProductInput!): Product
  updateProduct(id: ID!, input: ProductInput!): Product
  deleteProduct(id: ID!): Boolean
  createCategory(input: CategoryInput!): Category
}

type Subscription {
  productCreated: Product
  productUpdated: Product
  productDeleted: ID
}

type Product {
  id: ID!
  name: String!
  description: String
  price: Float!
  imageUrl: String
  category: Category!
  createdAt: String
  updatedAt: String
  inventory: Inventory
  reviews: [Review]
```

```
}

type Category {
  id: ID!
  name: String!
  description: String
  products: [Product]
}

type Inventory {
  productId: ID!
  quantity: Int!
  status: InventoryStatus!
  lastUpdated: String
}

enum InventoryStatus {
  IN_STOCK
  LOW_STOCK
  OUT_OF_STOCK
}

type Review {
  id: ID!
  productId: ID!
  rating: Int!
  comment: String
  reviewer: String
  createdAt: String
}

input ProductInput {
  name: String!
  description: String
  price: Float!
  categoryId: ID!
  imageUrl: String
}

input CategoryInput {
  name: String!
  description: String
}
```

GraphQL Controllers (Resolvers)

```
@Controller
public class ProductGraphQLController {

    private final ProductService productService;
    private final CategoryService categoryService;
```

```
    private final InventoryService inventoryService;
    private final ReviewService reviewService;

    public ProductGraphQLController(ProductService productService,
                                    CategoryService categoryService,
                                    InventoryService inventoryService,
                                    ReviewService reviewService) {
        this.productService = productService;
        this.categoryService = categoryService;
        this.inventoryService = inventoryService;
        this.reviewService = reviewService;
    }

    @QueryMapping
    public List<ProductDto> products(@Argument Integer limit, @Argument Integer
offset) {
        if (limit != null && offset != null) {
            return productService.findAllPaginated(offset, limit);
        }
        return productService.findAll();
    }

    @QueryMapping
    public ProductDto product(@Argument String id) {
        return productService.findById(Long.valueOf(id));
    }

    @QueryMapping
    public List<ProductDto> searchProducts(
            @Argument String keyword,
            @Argument BigDecimal minPrice,
            @Argument BigDecimal maxPrice) {

        return productService.searchProducts(keyword, minPrice, maxPrice);
    }

    @MutationMapping
    public ProductDto createProduct(@Argument ProductInput input) {
        ProductDto productDto = new ProductDto();
        productDto.setName(input.getName());
        productDto.setDescription(input.getDescription());
        productDto.setPrice(input.getPrice());
        productDto.setCategoryId(Long.valueOf(input.getCategoryId()));
        productDto.setImageUrl(input.getImageUrl());

        return productService.createProduct(productDto);
    }

    @MutationMapping
    public ProductDto updateProduct(@Argument String id, @Argument ProductInput
input) {
        ProductDto productDto = new ProductDto();
        productDto.setName(input.getName());
        productDto.setDescription(input.getDescription());
```

```
        productDto.setPrice(input.getPrice());
        productDto.setCategoryId(Long.valueOf(input.getCategoryId()));
        productDto.setImageUrl(input.getImageUrl());

        return productService.updateProduct(Long.valueOf(id), productDto);
    }

    @MutationMapping
    public boolean deleteProduct(@Argument String id) {
        try {
            productService.deleteProduct(Long.valueOf(id));
            return true;
        } catch (Exception e) {
            return false;
        }
    }

    @SchemaMapping(typeName = "Product", field = "category")
    public CategoryDto getCategory(ProductDto product) {
        return categoryService.findById(product.getCategoryId());
    }

    @SchemaMapping(typeName = "Product", field = "inventory")
    public InventoryDto getInventory(ProductDto product) {
        return inventoryService.findByProductId(product.getId());
    }

    @SchemaMapping(typeName = "Product", field = "reviews")
    public List<ReviewDto> getReviews(ProductDto product) {
        return reviewService.findByProductId(product.getId());
    }

    @Data
    public static class ProductInput {
        private String name;
        private String description;
        private BigDecimal price;
        private String categoryId;
        private String imageUrl;
    }
}
```

```
@Controller
public class CategoryGraphQLController {

    private final CategoryService categoryService;
    private final ProductService productService;

    public CategoryGraphQLController(CategoryService categoryService,
                                     ProductService productService) {
        this.categoryService = categoryService;
```

```
        this.productService = productService;
    }

    @QueryMapping
    public List<CategoryDto> categories() {
        return categoryService.findAll();
    }

    @QueryMapping
    public CategoryDto category(@Argument String id) {
        return categoryService.findById(Long.valueOf(id));
    }

    @MutationMapping
    public CategoryDto createCategory(@Argument CategoryInput input) {
        CategoryDto categoryDto = new CategoryDto();
        categoryDto.setName(input.getName());
        categoryDto.setDescription(input.getDescription());

        return categoryService.createCategory(categoryDto);
    }

    @SchemaMapping(typeName = "Category", field = "products")
    public List<ProductDto> getProducts(CategoryDto category) {
        return productService.findByCategoryId(category.getId());
    }

    @Data
    public static class CategoryInput {
        private String name;
        private String description;
    }
}
```

GraphQL Exception Handling

```
@Component
public class GraphQLExceptionHandler implements DataFetcherExceptionResolver {

    private static final Logger logger =
LoggerFactory.getLogger(GraphQLExceptionHandler.class);

    @Override
    public List<GraphQLError> resolveException(Throwable exception,
            DataFetchingEnvironment environment) {

        if (exception instanceof ResourceNotFoundException) {
            return Collections.singletonList(
                    GraphqlErrorBuilder.newError(environment)
                            .message(exception.getMessage())
                            .errorType(ErrorType.NOT_FOUND)
```

```
                                .build());
        } else if (exception instanceof ValidationException) {
            ValidationException validationException = (ValidationException)
exception;
            return Collections.singletonList(
                    GraphqlErrorBuilder.newError(environment)
                            .message("Validation error")
                            .errorType(ErrorType.BAD_REQUEST)
                            .extensions(Map.of("errors",
validationException.getErrors()))
                            .build());
        } else if (exception instanceof AccessDeniedException) {
            return Collections.singletonList(
                    GraphqlErrorBuilder.newError(environment)
                            .message("Access denied")
                            .errorType(ErrorType.FORBIDDEN)
                            .build());
        }

        logger.error("Unhandled exception during GraphQL execution", exception);

        return Collections.singletonList(
                GraphqlErrorBuilder.newError(environment)
                        .message("Internal server error")
                        .errorType(ErrorType.INTERNAL_ERROR)
                        .build());
    }
}
```

GraphQL Subscriptions

```
@Controller
public class ProductSubscriptionController {

    private final Sinks.Many<ProductDto> productCreatedSink =
Sinks.many().multicast().onBackpressureBuffer();
    private final Sinks.Many<ProductDto> productUpdatedSink =
Sinks.many().multicast().onBackpressureBuffer();
    private final Sinks.Many<String> productDeletedSink =
Sinks.many().multicast().onBackpressureBuffer();

    @SubscriptionMapping
    public Flux<ProductDto> productCreated() {
        return productCreatedSink.asFlux();
    }

    @SubscriptionMapping
    public Flux<ProductDto> productUpdated() {
        return productUpdatedSink.asFlux();
    }
```

```
    @SubscriptionMapping
    public Flux<String> productDeleted() {
        return productDeletedSink.asFlux();
    }

    @EventListener
    public void handleProductCreatedEvent(ProductCreatedEvent event) {
        productCreatedSink.tryEmitNext(event.getProduct());
    }

    @EventListener
    public void handleProductUpdatedEvent(ProductUpdatedEvent event) {
        productUpdatedSink.tryEmitNext(event.getProduct());
    }

    @EventListener
    public void handleProductDeletedEvent(ProductDeletedEvent event) {
        productDeletedSink.tryEmitNext(event.getProductId().toString());
    }
}
```

GraphQL Configuration

```
@Configuration
public class GraphQLConfig {

    @Bean
    public GraphQlSourceBuilderCustomizer sourceBuilderCustomizer() {
        return (builder) -> builder
                .configureGraphQl(graphQlBuilder -> graphQlBuilder
                        .instrumentation(new
DataLoaderDispatcherInstrumentation())
                        .queryExecutionStrategy(new AsyncExecutionStrategy())
                        .mutationExecutionStrategy(new
AsyncSerialExecutionStrategy()));
    }

    @Bean
    public WebSocketService webSocketService() {
        return new
DefaultWebSocketService(WebSocketHandlerAdapter.builder().build());
    }

    @Bean
    public WebSocketGraphQlHandler webSocketGraphQlHandler(
            GraphQlService graphQlService, WebSocketService webSocketService) {
        return WebSocketGraphQlHandler.builder(graphQlService,
webSocketService).build();
    }

    @Bean
```

```
    public HandlerMapping graphQlWebSocketEndpoint(WebSocketGraphQlHandler
webSocketGraphQlHandler) {
        return new WebSocketHandlerMapping("/graphql", webSocketGraphQlHandler);
    }
}
```

DataLoader for Batch Loading

```
@Component
public class GraphQLDataLoaders {

    private final ProductService productService;
    private final CategoryService categoryService;
    private final InventoryService inventoryService;
    private final ReviewService reviewService;

    public GraphQLDataLoaders(ProductService productService,
                              CategoryService categoryService,
                              InventoryService inventoryService,
                              ReviewService reviewService) {
        this.productService = productService;
        this.categoryService = categoryService;
        this.inventoryService = inventoryService;
        this.reviewService = reviewService;
    }

    @Bean
    public DataLoader<Long, CategoryDto> categoryDataLoader() {
        return DataLoader.newDataLoader(categoryIds -> {
            List<CategoryDto> categories =
categoryService.findAllById(categoryIds);
            Map<Long, CategoryDto> categoryMap = categories.stream()
                    .collect(Collectors.toMap(CategoryDto::getId,
Function.identity()));

            return CompletableFuture.supplyAsync(() ->
                    categoryIds.stream()
                            .map(id -> categoryMap.getOrDefault(id, null))
                            .collect(Collectors.toList()));
        });
    }

    @Bean
    public DataLoader<Long, InventoryDto> inventoryDataLoader() {
        return DataLoader.newDataLoader(productIds -> {
            List<InventoryDto> inventories =
inventoryService.findByProductIds(productIds);
            Map<Long, InventoryDto> inventoryMap = inventories.stream()
                    .collect(Collectors.toMap(InventoryDto::getProductId,
Function.identity()));
```

```
            return CompletableFuture.supplyAsync(() ->
                    productIds.stream()
                            .map(id -> inventoryMap.getOrDefault(id, null))
                            .collect(Collectors.toList()));
        });
    }

    @Bean
    public DataLoader<Long, List<ReviewDto>> reviewsDataLoader() {
        return DataLoader.newDataLoader(productIds -> {
            Map<Long, List<ReviewDto>> reviewsMap =
reviewService.findByProductIds(productIds).stream()
                    .collect(Collectors.groupingBy(ReviewDto::getProductId));

            return CompletableFuture.supplyAsync(() ->
                    productIds.stream()
                            .map(id -> reviewsMap.getOrDefault(id,
Collections.emptyList()))
                            .collect(Collectors.toList()));
        });
    }

    @Bean
    public DataLoader<Long, List<ProductDto>> productsByCategoryDataLoader() {
        return DataLoader.newDataLoader(categoryIds -> {
            Map<Long, List<ProductDto>> productsMap =
productService.findByCategoryIds(categoryIds).stream()
                    .collect(Collectors.groupingBy(ProductDto::getCategoryId));

            return CompletableFuture.supplyAsync(() ->
                    categoryIds.stream()
                            .map(id -> productsMap.getOrDefault(id,
Collections.emptyList()))
                            .collect(Collectors.toList()));
        });
    }
}
```

```
@Controller
public class OptimizedProductGraphQLController {

    private final ProductService productService;

    public OptimizedProductGraphQLController(ProductService productService) {
        this.productService = productService;
    }

    @QueryMapping
    public List<ProductDto> products(@Argument Integer limit, @Argument Integer
offset) {
        if (limit != null && offset != null) {
```

```
            return productService.findAllPaginated(offset, limit);
        }
        return productService.findAll();
    }

    @SchemaMapping(typeName = "Product", field = "category")
    public CompletableFuture<CategoryDto> getCategory(ProductDto product,
                                                      DataLoader<Long, CategoryDto>
categoryDataLoader) {
        return categoryDataLoader.load(product.getCategoryId());
    }

    @SchemaMapping(typeName = "Product", field = "inventory")
    public CompletableFuture<InventoryDto> getInventory(ProductDto product,
                                                        DataLoader<Long,
InventoryDto> inventoryDataLoader) {
        return inventoryDataLoader.load(product.getId());
    }

    @SchemaMapping(typeName = "Product", field = "reviews")
    public CompletableFuture<List<ReviewDto>> getReviews(ProductDto product,
                                                         DataLoader<Long,
List<ReviewDto>> reviewsDataLoader) {
        return reviewsDataLoader.load(product.getId());
    }

    @SchemaMapping(typeName = "Category", field = "products")
    public CompletableFuture<List<ProductDto>> getProducts(CategoryDto category,
                                                           DataLoader<Long,
List<ProductDto>> productsByCategoryDataLoader) {
        return productsByCategoryDataLoader.load(category.getId());
    }
}
```

Custom Starter Creation

Creating reusable Spring Boot starters for common functionality.

Structure of a Custom Starter

```
my-starter/
├── pom.xml
├── my-starter-autoconfigure/
│   ├── pom.xml
│   └── src/
│       └── main/
│           ├── java/
│           │   └── com/example/starter/
│           │       ├── config/
│           │       │   ├── MyAutoConfiguration.java
│           │       │   └── MyProperties.java
```

```
│           │       ├── service/
│           │       │   ├── MyService.java
│           │       │   └── MyServiceImpl.java
│           │       └── MyEnableAutoConfiguration.java
│           └── resources/
│               └── META-INF/
│                   └── spring.factories
└── my-starter/
    └── pom.xml
```

Autoconfiguration Module

```xml
<!-- my-starter-autoconfigure/pom.xml -->
<project>
    <modelVersion>4.0.0</modelVersion>

    <groupId>com.example</groupId>
    <artifactId>my-starter-autoconfigure</artifactId>
    <version>1.0.0</version>
    <packaging>jar</packaging>

    <properties>
        <java.version>11</java.version>
        <spring-boot.version>2.7.0</spring-boot.version>
    </properties>

    <dependencies>
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter</artifactId>
        </dependency>
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-autoconfigure</artifactId>
        </dependency>
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-configuration-processor</artifactId>
            <optional>true</optional>
        </dependency>
    </dependencies>

    <dependencyManagement>
        <dependencies>
            <dependency>
                <groupId>org.springframework.boot</groupId>
                <artifactId>spring-boot-dependencies</artifactId>
                <version>${spring-boot.version}</version>
                <type>pom</type>
                <scope>import</scope>
            </dependency>
```

```
        </dependencies>
    </dependencyManagement>
</project>
```

```
// my-starter-
autoconfigure/src/main/java/com/example/starter/config/MyProperties.java
@ConfigurationProperties(prefix = "my.service")
@Data
public class MyProperties {

    /**
     * Whether the service is enabled.
     */
    private boolean enabled = true;

    /**
     * The service name.
     */
    private String name = "Default Service";

    /**
     * The service timeout in milliseconds.
     */
    private int timeoutMs = 1000;

    /**
     * The maximum number of retries.
     */
    private int maxRetries = 3;
}
```

```
// my-starter-
autoconfigure/src/main/java/com/example/starter/service/MyService.java
public interface MyService {

    /**
     * Process the given data.
     *
     * @param data the data to process
     * @return the processed result
     */
    String process(String data);

    /**
     * Get service information.
     *
     * @return service information as a map
     */
```

```
    Map<String, Object> getInfo();
}
```

```
// my-starter-
autoconfigure/src/main/java/com/example/starter/service/MyServiceImpl.java
public class MyServiceImpl implements MyService {

    private final String name;
    private final int timeoutMs;
    private final int maxRetries;

    public MyServiceImpl(String name, int timeoutMs, int maxRetries) {
        this.name = name;
        this.timeoutMs = timeoutMs;
        this.maxRetries = maxRetries;
    }

    @Override
    public String process(String data) {
        // Implement service logic
        int retryCount = 0;
        while (retryCount < maxRetries) {
            try {
                // Simulate processing
                if (data == null || data.isEmpty()) {
                    throw new IllegalArgumentException("Data cannot be empty");
                }

                // Process the data
                return "[" + name + "] Processed: " + data;
            } catch (Exception e) {
                retryCount++;
                if (retryCount >= maxRetries) {
                    throw new RuntimeException("Failed to process after " +
maxRetries + " retries", e);
                }

                // Wait before retry
                try {
                    Thread.sleep(timeoutMs / maxRetries);
                } catch (InterruptedException ex) {
                    Thread.currentThread().interrupt();
                    throw new RuntimeException("Processing interrupted", ex);
                }
            }
        }

        return null; // Should not reach here
    }

    @Override
```

```
    public Map<String, Object> getInfo() {
        Map<String, Object> info = new HashMap<>();
        info.put("name", name);
        info.put("timeoutMs", timeoutMs);
        info.put("maxRetries", maxRetries);
        return info;
    }
}
```

```
// my-starter-
autoconfigure/src/main/java/com/example/starter/config/MyAutoConfiguration.java
@Configuration
@ConditionalOnClass(MyService.class)
@EnableConfigurationProperties(MyProperties.class)
public class MyAutoConfiguration {

    private final MyProperties properties;

    public MyAutoConfiguration(MyProperties properties) {
        this.properties = properties;
    }

    @Bean
    @ConditionalOnMissingBean
    @ConditionalOnProperty(prefix = "my.service", name = "enabled", havingValue =
"true", matchIfMissing = true)
    public MyService myService() {
        return new MyServiceImpl(
                properties.getName(),
                properties.getTimeoutMs(),
                properties.getMaxRetries());
    }

    @Bean
    @ConditionalOnMissingBean
    public MyServiceHealthIndicator myServiceHealthIndicator(MyService myService)
{
        return new MyServiceHealthIndicator(myService);
    }

    @Bean
    @ConditionalOnWebApplication
    public MyServiceController myServiceController(MyService myService) {
        return new MyServiceController(myService);
    }

    @Configuration
    @ConditionalOnBean(MetricsRegistry.class)
    public static class MyServiceMetricsConfiguration {

        @Bean
```

```
        public MyServiceMetrics myServiceMetrics(MyService myService,
MetricsRegistry metricsRegistry) {
            return new MyServiceMetrics(myService, metricsRegistry);
        }
    }

    public static class MyServiceHealthIndicator implements HealthIndicator {

        private final MyService myService;

        public MyServiceHealthIndicator(MyService myService) {
            this.myService = myService;
        }

        @Override
        public Health health() {
            try {
                Map<String, Object> info = myService.getInfo();
                return Health.up()
                        .withDetails(info)
                        .build();
            } catch (Exception e) {
                return Health.down()
                        .withException(e)
                        .build();
            }
        }
    }

    @RestController
    @RequestMapping("/api/my-service")
    public static class MyServiceController {

        private final MyService myService;

        public MyServiceController(MyService myService) {
            this.myService = myService;
        }

        @GetMapping("/info")
        public Map<String, Object> getInfo() {
            return myService.getInfo();
        }

        @PostMapping("/process")
        public Map<String, String> process(@RequestBody String data) {
            String result = myService.process(data);
            Map<String, String> response = new HashMap<>();
            response.put("result", result);
            return response;
        }
    }

    public static class MyServiceMetrics {
```

```
        private final MyService myService;
        private final MetricsRegistry metricsRegistry;

        public MyServiceMetrics(MyService myService, MetricsRegistry
metricsRegistry) {
            this.myService = myService;
            this.metricsRegistry = metricsRegistry;
            registerMetrics();
        }

        private void registerMetrics() {
            // Register service metrics
            metricsRegistry.gauge("my.service.timeout", myService, service ->
                    ((Number) service.getInfo().get("timeoutMs")).doubleValue());

            metricsRegistry.gauge("my.service.max.retries", myService, service ->
                    ((Number) service.getInfo().get("maxRetries")).doubleValue());
        }
    }

    // Mock interface for metrics registry
    public interface MetricsRegistry {
        <T> void gauge(String name, T obj, ToDoubleFunction<T> valueFunction);
    }
}
```

```
# my-starter-autoconfigure/src/main/resources/META-INF/spring.factories
org.springframework.boot.autoconfigure.EnableAutoConfiguration=\
  com.example.starter.config.MyAutoConfiguration
```

Starter Module

```
<!-- my-starter/pom.xml -->
<project>
    <modelVersion>4.0.0</modelVersion>

    <groupId>com.example</groupId>
    <artifactId>my-starter</artifactId>
    <version>1.0.0</version>
    <packaging>jar</packaging>

    <properties>
        <java.version>11</java.version>
    </properties>

    <dependencies>
        <dependency>
            <groupId>com.example</groupId>
```

```
            <artifactId>my-starter-autoconfigure</artifactId>
            <version>${project.version}</version>
        </dependency>
    </dependencies>
</project>
```

Root POM

```
<!-- my-starter/pom.xml (root) -->
<project>
    <modelVersion>4.0.0</modelVersion>

    <groupId>com.example</groupId>
    <artifactId>my-starter-parent</artifactId>
    <version>1.0.0</version>
    <packaging>pom</packaging>

    <modules>
        <module>my-starter-autoconfigure</module>
        <module>my-starter</module>
    </modules>
</project>
```

Using the Custom Starter

```
<!-- Application using the starter -->
<dependency>
    <groupId>com.example</groupId>
    <artifactId>my-starter</artifactId>
    <version>1.0.0</version>
</dependency>
```

```
# application.properties
my.service.enabled=true
my.service.name=Custom Service
my.service.timeout-ms=2000
my.service.max-retries=5
```

```
@RestController
@RequestMapping("/custom")
public class CustomController {

    private final MyService myService;
```

```
    public CustomController(MyService myService) {
        this.myService = myService;
    }

    @GetMapping("/process")
    public String processData(@RequestParam String data) {
        return myService.process(data);
    }
}
```

Advanced Starter Features

```
// Conditional Auto-Configuration
@Configuration
@ConditionalOnClass({ DataSource.class, MyService.class })
@ConditionalOnBean(DataSource.class)
@EnableConfigurationProperties(MyProperties.class)
public class MyDatabaseAutoConfiguration {

    private final MyProperties properties;
    private final DataSource dataSource;

    public MyDatabaseAutoConfiguration(MyProperties properties, DataSource
dataSource) {
        this.properties = properties;
        this.dataSource = dataSource;
    }

    @Bean
    @ConditionalOnMissingBean
    public MyDatabaseService myDatabaseService() {
        return new MyDatabaseServiceImpl(dataSource, properties.getName());
    }
}
```

```
// Test Auto-Configuration
@Configuration
@AutoConfigureAfter(MyAutoConfiguration.class)
@ConditionalOnClass(MockMvc.class)
@ConditionalOnProperty(prefix = "my.service", name = "test-mode", havingValue =
"true")
public class MyTestAutoConfiguration {

    private final MyProperties properties;

    public MyTestAutoConfiguration(MyProperties properties) {
        this.properties = properties;
    }
```

```
    @Bean
    @ConditionalOnMissingBean
    public MyService myMockService() {
        // Return a mock implementation for testing
        return new MyServiceMock(properties.getName());
    }

    @Bean
    public MyServiceTestController myServiceTestController(MyService myService) {
        return new MyServiceTestController(myService);
    }

    @RestController
    @RequestMapping("/test/my-service")
    public static class MyServiceTestController {

        private final MyService myService;

        public MyServiceTestController(MyService myService) {
            this.myService = myService;
        }

        @GetMapping("/test")
        public Map<String, String> test() {
            Map<String, String> response = new HashMap<>();
            response.put("result", myService.process("test-data"));
            response.put("mode", "test");
            return response;
        }
    }

    public static class MyServiceMock implements MyService {

        private final String name;

        public MyServiceMock(String name) {
            this.name = name;
        }

        @Override
        public String process(String data) {
            return "[MOCK: " + name + "] Processed: " + data;
        }

        @Override
        public Map<String, Object> getInfo() {
            Map<String, Object> info = new HashMap<>();
            info.put("name", name);
            info.put("mock", true);
            return info;
        }
    }
}
```

```
# my-starter-autoconfigure/src/main/resources/META-INF/spring.factories
org.springframework.boot.autoconfigure.EnableAutoConfiguration=\
  com.example.starter.config.MyAutoConfiguration,\
  com.example.starter.config.MyDatabaseAutoConfiguration,\
  com.example.starter.config.MyTestAutoConfiguration
```

Custom Annotations

```
// my-starter-autoconfigure/src/main/java/com/example/starter/EnableMyService.java
@Target(ElementType.TYPE)
@Retention(RetentionPolicy.RUNTIME)
@Documented
@Import(MyAutoConfiguration.class)
public @interface EnableMyService {
}
```

```
// Application using explicit opt-in
@SpringBootApplication
@EnableMyService
public class MyApplication {

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

Documentation

```
# My Service Starter

A Spring Boot starter that provides a simple service for processing data with
configurable retry and timeout behavior.

## Features

- Simple service interface with process and info methods
- Configurable name, timeout, and retry settings
- Auto-configured health indicators
- Optional REST controller for service access
- Test mode support for easier testing

## Installation

```xml
```
```

```
<dependency>
    <groupId>com.example</groupId>
    <artifactId>my-starter</artifactId>
    <version>1.0.0</version>
</dependency>
```

Configuration Properties

Property	Description	Default
`my.service.enabled`	Whether the service is enabled	`true`
`my.service.name`	The service name	`Default Service`
`my.service.timeout-ms`	The service timeout in milliseconds	`1000`
`my.service.max-retries`	Maximum number of retries	`3`
`my.service.test-mode`	Whether to use the test mock implementation	`false`

Usage Examples

Basic Usage

```
@RestController
@RequestMapping("/api")
public class MyController {

    private final MyService myService;

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

    @PostMapping("/process")
    public String processData(@RequestBody String data) {
        return myService.process(data);
    }

    @GetMapping("/info")
    public Map<String, Object> getInfo() {
        return myService.getInfo();
    }
}
```

Custom Configuration

```
# application.properties
my.service.name=Custom Service
my.service.timeout-ms=2000
my.service.max-retries=5
```

Test Mode

For integration tests, you can enable the test mode:

```
# application-test.properties
my.service.test-mode=true
```

Advanced Usage

Creating a Custom Implementation

```
@Configuration
public class CustomServiceConfig {

    @Bean
    @Primary
    public MyService customMyService(MyProperties properties) {
        return new CustomMyServiceImpl(properties);
    }

    private static class CustomMyServiceImpl implements MyService {
        // Custom implementation
    }
}
```

```


## Reference Section

### Spring Boot Interview Questions and Answers

#### Foundation Questions

**Q1: What is Spring Boot, and how does it differ from the traditional Spring Framework?**

Spring Boot is an extension of the Spring Framework that simplifies the process of building production-ready applications. It differs from traditional Spring in several key ways:

1. **Auto-configuration**: Spring Boot automatically configures your application based on the dependencies you've added to your project, eliminating the need for
```

extensive XML or Java configuration.

2. **Standalone applications**: Spring Boot allows you to create self-contained applications that can be run without deploying to an external server.

3. **Embedded servers**: It includes embedded servlet containers like Tomcat, Jetty, or Undertow, eliminating the need for external server configuration.

4. **Opinionated defaults**: Spring Boot provides sensible default configurations for various components, allowing developers to get started quickly.

5. **Production-ready features**: It includes built-in support for metrics, health checks, and other production-oriented features through Spring Boot Actuator.

The traditional Spring Framework, while powerful, required more manual configuration through XML files or Java-based configuration classes, and applications typically needed to be deployed to external servers.

Q2: Explain the concept of auto-configuration in Spring Boot.

Auto-configuration is one of the core features of Spring Boot that automatically configures your Spring application based on the dependencies present on the classpath.

- Spring Boot detects the libraries available in your application and automatically creates and configures beans based on recommended practices.
- These auto-configurations are conditionally applied using `@Conditional` annotations that check for the presence of specific classes, beans, or properties.
- Auto-configurations are defined in `META-INF/spring.factories` files with the key `org.springframework.boot.autoconfigure.EnableAutoConfiguration`.
- For example, if H2 database is on the classpath, Spring Boot automatically configures an in-memory database connection without requiring explicit configuration.
- Developers can override auto-configured beans by defining their own beans of the same type.
- Auto-configuration can be disabled either for specific configurations using `@EnableAutoConfiguration(exclude={...})` or through properties like `spring.autoconfigure.exclude`.

Q3: What are the Spring Boot Starters? Give examples of commonly used starters.

Spring Boot Starters are dependency descriptors that bundle related dependencies for specific functionalities, simplifying dependency management for developers.

Common starters include:

1. **spring-boot-starter-web**: For building web applications, including RESTful applications with Spring MVC. It includes embedded Tomcat and JSON support.

2. **spring-boot-starter-data-jpa**: For using Spring Data JPA with Hibernate to access relational databases.

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

````
application.

4. **spring-boot-starter-test**: Includes testing libraries like JUnit, Mockito,
and Spring Test for comprehensive testing.

5. **spring-boot-starter-actuator**: Adds production-ready features for monitoring
and managing your application.

6. **spring-boot-starter-thymeleaf**: For using the Thymeleaf templating engine
for server-side HTML rendering.

7. **spring-boot-starter-jdbc**: For accessing relational databases using JDBC.

8. **spring-boot-starter-data-mongodb**: For working with MongoDB document
databases.

9. **spring-boot-starter-cache**: For enabling Spring's caching support.

10. **spring-boot-starter-validation**: For using validation capabilities with
Hibernate Validator.

**Q4: How can you exclude auto-configuration classes in Spring Boot?**

In Spring Boot, there are several ways to exclude specific auto-configuration
classes:

1. **Using @SpringBootApplication annotation**:
   ```java
   @SpringBootApplication(exclude = {DataSourceAutoConfiguration.class})
   public class Application {
       public static void main(String[] args) {
           SpringApplication.run(Application.class, args);
       }
   }
````

2. **Using @EnableAutoConfiguration annotation**:

```java
@EnableAutoConfiguration(exclude = {DataSourceAutoConfiguration.class})
@ComponentScan
@Configuration
public class Application {
    public static void main(String[] args) {
        SpringApplication.run(Application.class, args);
    }
}
```

3. **Using properties in application.properties/application.yml**:
```

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

OR

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

4. **When excluding auto-configurations that don't exist on the classpath**, use the `excludeName` property:

```
@SpringBootApplication(excludeName =
{"org.springframework.boot.autoconfigure.jdbc.DataSourceAutoConfiguration"})
```

Q5: Explain Spring Boot Actuator and its benefits.

Spring Boot Actuator is a sub-project of Spring Boot that adds several production-ready features to your application, helping with monitoring and management.

Benefits:

1. **Health Checks**: Built-in health indicators to check the state of your application and its dependencies.
2. **Metrics Collection**: Integration with monitoring systems to collect and export metrics about JVM, HTTP requests, database usage, etc.
3. **Endpoint Exposure**: Multiple endpoints for application information, such as:
   - `/health`: Shows application health information
   - `/info`: Displays arbitrary application information
   - `/metrics`: Shows metrics information
   - `/env`: Exposes environment properties
   - `/mappings`: Displays all HTTP request mappings
   - `/loggers`: Shows and modifies the configuration of loggers in the application
4. **Security Integration**: Secure access to sensitive endpoints with Spring Security.
5. **Customizability**: Ability to create custom endpoints and health indicators.
6. **JMX Integration**: Exposes endpoints via JMX for management tools.
7. **Distributed Tracing**: Integration with distributed tracing systems.

Example configuration in application.properties:

```
# Enable all endpoints
management.endpoints.web.exposure.include=*

# Enable only specific endpoints
management.endpoints.web.exposure.include=health,info,metrics

# Custom path for actuator endpoints
management.endpoints.web.base-path=/management

# Show full health details when authenticated
management.endpoint.health.show-details=when-authorized
```

Intermediate Questions

Q6: How can you configure external properties in Spring Boot?

Spring Boot provides several ways to configure external properties:

1. **application.properties/application.yml**: The most common approach is to place these files in the following locations:
   - `./config/` subdirectory of the current directory
   - Current directory
   - `config` package in the classpath
   - Root of the classpath
2. **Profile-specific files**: For environment-specific configurations:
   - `application-{profile}.properties` or `application-{profile}.yml`
   - Activated via `spring.profiles.active` property
3. **Command-line arguments**:

   ```
   java -jar myapp.jar --server.port=8080 --spring.profiles.active=prod
   ```

4. **Environment variables**:

   ```
   export SERVER_PORT=8080
   export SPRING_PROFILES_ACTIVE=prod
   ```

   Spring Boot automatically converts environment variables to camel case property names.
5. **External configuration files**:

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

6. **JNDI attributes** in `java:comp/env`
7. **@Value annotation** to inject properties:

```
@Value("${server.port}")
private int serverPort;
```

8. **@ConfigurationProperties** for type-safe configuration:

```
@ConfigurationProperties(prefix = "app")
public class AppProperties {
    private String name;
    private int maxConnections;
    // getters and setters
}
```

The order of precedence (from highest to lowest) is:

- Command-line arguments
- SPRING_APPLICATION_JSON properties
- Properties from JNDI
- Java System properties
- OS environment variables
- Profile-specific properties
- Application properties
- Default properties

Q7: What is Spring Boot DevTools and what features does it provide?

Spring Boot DevTools is a set of tools that aims to improve developer productivity. It is automatically disabled when running a packaged application.

Key features include:

1. **Automatic Restart**: When files on the classpath change, the application automatically restarts. It uses two classloaders:
    - One for your application code (frequently changing)
    - One for dependencies (rarely changing) This approach makes restarts faster than a cold start.
2. **LiveReload**: Automatic browser refresh when resources change.
3. **Property Defaults**: Development-friendly property defaults (like disabling template caching).
4. **Remote Development**: Support for remote application updates during development.
5. **Global Settings**: Store configuration in a `.spring-boot-devtools.properties` file in your home directory to apply settings to all projects.

6. **Remote Debugging**: Simplified remote debugging configuration.

To add DevTools to your project:

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-devtools</artifactId>
    <optional>true</optional>
</dependency>
```

Q8: How do you implement custom health indicators in Spring Boot Actuator?

Custom health indicators in Spring Boot Actuator allow you to report the health status of specific components or dependencies of your application:

1. **Implement HealthIndicator interface**:

```
@Component
public class CustomHealthIndicator implements HealthIndicator {

    @Override
    public Health health() {
        // Check health of the component
        boolean isHealthy = checkIfComponentIsHealthy();

        if (isHealthy) {
            return Health.up()
                    .withDetail("description", "Component is working
correctly")
                    .build();
        } else {
            return Health.down()
                    .withDetail("error", "Component is not responding")
                    .build();
        }
    }

    private boolean checkIfComponentIsHealthy() {
        // Actual health check logic here
        return true;
    }
}
```

2. **Using AbstractHealthIndicator**:

```
@Component
public class CustomHealthIndicator extends AbstractHealthIndicator {
```

```
    @Override
    protected void doHealthCheck(Builder builder) throws Exception {
        try {
            // Perform health check
            builder.up()
                   .withDetail("version", "1.0.0")
                   .withDetail("status", "Running");
        } catch (Exception e) {
            builder.down()
                   .withDetail("error", e.getMessage())
                   .withException(e);
        }
    }
}
```

3. **Configure health endpoint**:

```
# Show detailed health information
management.endpoint.health.show-details=always

# Group health indicators
management.endpoint.health.group.custom.include=customHealth,diskSpace
```

4. **Access health endpoint**:
   - `/actuator/health` - General health status
   - `/actuator/health/custom` - Specific health group
   - `/actuator/health/customHealth` - Specific indicator

Health statuses include UP, DOWN, OUT_OF_SERVICE, and UNKNOWN, with custom details providing additional context.

Q9: How do you secure a Spring Boot application?

Securing a Spring Boot application typically involves:

1. **Adding Spring Security dependency**:

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

2. **Creating a security configuration class**:

```
@Configuration
@EnableWebSecurity
```

```
public class SecurityConfig extends WebSecurityConfigurerAdapter {

    @Override
    protected void configure(HttpSecurity http) throws Exception {
        http
            .authorizeRequests()
                .antMatchers("/", "/public/**").permitAll()
                .antMatchers("/admin/**").hasRole("ADMIN")
                .anyRequest().authenticated()
            .and()
            .formLogin()
                .loginPage("/login")
                .permitAll()
            .and()
            .logout()
                .permitAll();
    }

    @Override
    protected void configure(AuthenticationManagerBuilder auth) throws
Exception {
        auth.inMemoryAuthentication()

.withUser("user").password(passwordEncoder().encode("password")).roles("USER
")
            .and()

.withUser("admin").password(passwordEncoder().encode("admin")).roles("USER",
"ADMIN");
    }

    @Bean
    public PasswordEncoder passwordEncoder() {
        return new BCryptPasswordEncoder();
    }
}
```

3. **Using database authentication**:

```
@Autowired
private UserDetailsService userDetailsService;

@Override
protected void configure(AuthenticationManagerBuilder auth) throws Exception
{
    auth.userDetailsService(userDetailsService)
        .passwordEncoder(passwordEncoder());
}
```

4. **Method-level security**:

```
@EnableGlobalMethodSecurity(prePostEnabled = true, securedEnabled = true,
jsr250Enabled = true)
public class SecurityConfig extends WebSecurityConfigurerAdapter {
    // configuration
}

@RestController
public class UserController {
    @PreAuthorize("hasRole('ADMIN')")
    @GetMapping("/users")
    public List<User> getAllUsers() {
        // Only accessible to admins
    }
}
```

5. **OAuth2/JWT authentication**:

```
@Configuration
@EnableResourceServer
public class ResourceServerConfig extends ResourceServerConfigurerAdapter {
    @Override
    public void configure(HttpSecurity http) throws Exception {
        http.authorizeRequests()
            .antMatchers("/api/**").authenticated();
    }
}
```

6. **CSRF protection** (enabled by default):

```
<form action="/process" method="post">
  <input
    type="hidden"
    name="${_csrf.parameterName}"
    value="${_csrf.token}"
  />
  <!-- form fields -->
</form>
```

7. **CORS configuration**:

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

```
                .allowedMethods("GET", "POST", "PUT", "DELETE")
                .allowCredentials(true);
    }
}
```

8. **Security headers**:

```
http.headers()
    .contentSecurityPolicy("default-src 'self'")
    .and()
    .referrerPolicy(ReferrerPolicy.ORIGIN)
    .and()
    .frameOptions().deny();
```

9. **Rate limiting** (with Spring Security and a custom filter):

```
@Component
public class RateLimitingFilter extends OncePerRequestFilter {
    // Implement rate limiting logic
}
```

Q10: How do you handle exceptions in a Spring Boot REST application?

Spring Boot offers several approaches for handling exceptions in REST applications:

1. **@ExceptionHandler** at the controller level:

```
@RestController
public class ProductController {
    // Controller methods

    @ExceptionHandler(ResourceNotFoundException.class)
    public ResponseEntity<ErrorResponse> handleResourceNotFoundException(
            ResourceNotFoundException ex, WebRequest request) {

        ErrorResponse error = new ErrorResponse(
            LocalDateTime.now(),
            HttpStatus.NOT_FOUND.value(),
            ex.getMessage(),
            request.getDescription(false)
        );

        return new ResponseEntity<>(error, HttpStatus.NOT_FOUND);
    }
}
```

2. **@ControllerAdvice/@RestControllerAdvice** for global exception handling:

```
@RestControllerAdvice
public class GlobalExceptionHandler {

    @ExceptionHandler(ResourceNotFoundException.class)
    public ResponseEntity<ErrorResponse> handleResourceNotFoundException(
            ResourceNotFoundException ex, WebRequest request) {

        ErrorResponse error = new ErrorResponse(
            LocalDateTime.now(),
            HttpStatus.NOT_FOUND.value(),
            ex.getMessage(),
            request.getDescription(false)
        );

        return new ResponseEntity<>(error, HttpStatus.NOT_FOUND);
    }

    @ExceptionHandler(ValidationException.class)
    public ResponseEntity<ErrorResponse> handleValidationException(
            ValidationException ex, WebRequest request) {
        // Handle validation exceptions
        return new ResponseEntity<>(error, HttpStatus.BAD_REQUEST);
    }

    @ExceptionHandler(Exception.class)
    public ResponseEntity<ErrorResponse> handleGlobalException(
            Exception ex, WebRequest request) {
        // Handle all other exceptions
        return new ResponseEntity<>(error,
HttpStatus.INTERNAL_SERVER_ERROR);
    }
}
```

3. **Custom error response class**:

```
@Data
public class ErrorResponse {
    private LocalDateTime timestamp;
    private int status;
    private String message;
    private String path;
    private Map<String, String> errors;

    // Constructors and methods
}
```

4. **@Valid with MethodArgumentNotValidException**:

```
@RestControllerAdvice
public class GlobalExceptionHandler {

    @ExceptionHandler(MethodArgumentNotValidException.class)
    public ResponseEntity<ErrorResponse> handleValidationExceptions(
            MethodArgumentNotValidException ex) {

        Map<String, String> errors = new HashMap<>();
        ex.getBindingResult().getFieldErrors().forEach(error ->
            errors.put(error.getField(), error.getDefaultMessage()));

        ErrorResponse errorResponse = new ErrorResponse();
        errorResponse.setTimestamp(LocalDateTime.now());
        errorResponse.setStatus(HttpStatus.BAD_REQUEST.value());
        errorResponse.setMessage("Validation failed");
        errorResponse.setErrors(errors);

        return new ResponseEntity<>(errorResponse, HttpStatus.BAD_REQUEST);
    }
}
```

5. **ResponseStatusException** for simple cases:

```
@GetMapping("/products/{id}")
public Product getProductById(@PathVariable Long id) {
    return productRepository.findById(id)
        .orElseThrow(() -> new ResponseStatusException(
            HttpStatus.NOT_FOUND, "Product not found with id: " + id
        ));
}
```

6. **Custom exception hierarchy**:

```
@ResponseStatus(HttpStatus.NOT_FOUND)
public class ResourceNotFoundException extends RuntimeException {
    public ResourceNotFoundException(String message) {
        super(message);
    }
}

@ResponseStatus(HttpStatus.BAD_REQUEST)
public class ValidationException extends RuntimeException {
    private Map<String, String> errors;

    public ValidationException(String message, Map<String, String> errors) {
        super(message);
        this.errors = errors;
    }
```

```
    public Map<String, String> getErrors() {
        return errors;
    }
}
```

Advanced Questions

Q11: Explain the Spring Boot Actuator endpoints and how to secure them.

Spring Boot Actuator provides various endpoints for monitoring and managing applications:

Key endpoints:

- `/health`: Application health information
- `/info`: Application information
- `/metrics`: Application metrics
- `/env`: Environment properties
- `/configprops`: Configuration properties
- `/mappings`: Request mapping information
- `/beans`: Application bean list
- `/threaddump`: Thread dump
- `/loggers`: Logger configuration
- `/shutdown`: Triggers application shutdown (disabled by default)
- `/prometheus`: Exports metrics in Prometheus format

Configuring endpoints:

```
# Enable/disable all endpoints
management.endpoints.enabled-by-default=false

# Enable specific endpoints
management.endpoint.health.enabled=true
management.endpoint.info.enabled=true

# Expose endpoints over HTTP
management.endpoints.web.exposure.include=health,info,metrics
management.endpoints.web.exposure.exclude=env,beans

# Custom base path
management.endpoints.web.base-path=/management

# Health details visibility
management.endpoint.health.show-details=when-authorized
```

Securing Actuator endpoints:

1. **Spring Security configuration**:

```
@Configuration
public class ActuatorSecurityConfig extends WebSecurityConfigurerAdapter {

    @Override
    protected void configure(HttpSecurity http) throws Exception {
        http
            .requestMatcher(EndpointRequest.toAnyEndpoint())
            .authorizeRequests()
                .requestMatchers(EndpointRequest.to("health",
"info")).permitAll()

.requestMatchers(EndpointRequest.toAnyEndpoint()).hasRole("ACTUATOR_ADMIN")
            .and()
            .httpBasic();
    }
}
```

2. **Using properties**:

```
# Require authentication for all endpoints except health and info
management.endpoints.web.exposure.include=*
management.endpoint.health.show-details=when-authorized

# Spring Security credentials (basic approach, use proper authentication in
production)
spring.security.user.name=admin
spring.security.user.password=admin_password
spring.security.user.roles=ACTUATOR_ADMIN
```

3. **Custom endpoint security**:

```
@Component
public class CustomEndpointSecurity extends
EndpointRequest.EndpointRequestMatcher {
    @Override
    protected boolean matchesInternal(HttpServletRequest request) {
        // Custom security logic
        return super.matchesInternal(request);
    }
}
```

4. **Configuring CORS for endpoints**:

```
management.endpoints.web.cors.allowed-origins=https://example.com
management.endpoints.web.cors.allowed-methods=GET,POST
```

Best practices:

- Only expose necessary endpoints
- Use HTTPS for all actuator endpoints
- Place actuator endpoints behind a gateway
- Use role-based access control
- Apply rate limiting
- Consider running actuator on a separate port
- Monitor access to sensitive endpoints

Q12: How can you deploy a Spring Boot application in different environments?

Deploying Spring Boot applications across environments involves several strategies for configuration and deployment:

1. Profile-based Configuration:

```
# application.properties (default/common properties)
spring.application.name=myapp
logging.level.root=INFO

# application-dev.properties
server.port=8080
spring.datasource.url=jdbc:h2:mem:devdb

# application-prod.properties
server.port=80
spring.datasource.url=jdbc:mysql://productiondb:3306/myapp
spring.jpa.hibernate.ddl-auto=validate
```

Activating profiles:

```
// Programmatically
SpringApplication app = new SpringApplication(MyApp.class);
app.setAdditionalProfiles("prod");
app.run(args);

// Command line
java -jar myapp.jar --spring.profiles.active=prod

// Environment variable
export SPRING_PROFILES_ACTIVE=prod
```

2. Externalized Configuration:

- Placing properties files in external locations:

```
java -jar myapp.jar --
spring.config.location=file:/etc/myapp/application.properties
```

- Using environment variables:

```
export SERVER_PORT=80
export SPRING_DATASOURCE_URL=jdbc:mysql://productiondb:3306/myapp
```

- Using command-line arguments:

```
java -jar myapp.jar --server.port=80
```

3. Deployment Options:

- **JAR Deployment**:

```
# Build the JAR
./mvnw package

# Run the JAR
java -jar target/myapp-0.0.1-SNAPSHOT.jar
```

- **WAR Deployment** to external servers:

```
<packaging>war</packaging>
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-tomcat</artifactId>
    <scope>provided</scope>
</dependency>
```

```
@SpringBootApplication
public class Application extends SpringBootServletInitializer {
    @Override
    protected SpringApplicationBuilder configure(SpringApplicationBuilder
application) {
        return application.sources(Application.class);
    }
}
```

- **Docker Containerization**:

```
FROM openjdk:11-jdk-slim
VOLUME /tmp
ARG JAR_FILE=target/*.jar
COPY ${JAR_FILE} app.jar
ENTRYPOINT ["java","-jar","/app.jar"]
```

- **Kubernetes Deployment**:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: myapp
spec:
  replicas: 3
  selector:
    matchLabels:
      app: myapp
  template:
    metadata:
      labels:
        app: myapp
    spec:
      containers:
        - name: myapp
          image: myapp:latest
          ports:
            - containerPort: 8080
          env:
            - name: SPRING_PROFILES_ACTIVE
              value: 'prod'
```

- **Cloud Platforms** (AWS, Azure, GCP):
  - AWS Elastic Beanstalk
  - Azure Spring Cloud
  - Google App Engine
  - Cloud Foundry
  - Heroku

4. Configuration Management:

- **Spring Cloud Config** for centralized configuration:

```
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-config</artifactId>
</dependency>
```

```
spring.cloud.config.uri=https://config-server.example.com
spring.cloud.config.name=myapp
spring.cloud.config.profile=prod
```

- **Kubernetes ConfigMaps and Secrets**:

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: myapp-config
data:
  application.properties: |
    server.port=8080
    logging.level.root=INFO
```

5. Monitoring and Management:

- Enable Actuator in all environments:

```
management.endpoints.web.exposure.include=health,info,metrics
```

- Use monitoring solutions:
  - Prometheus + Grafana
  - ELK Stack (Elasticsearch, Logstash, Kibana)
  - Dynatrace, New Relic, etc.
- Configure logging appropriately:

```
# Dev environment
logging.level.com.example=DEBUG

# Production environment
logging.level.com.example=WARN
logging.file.name=/var/log/myapp/application.log
```

Best practices:

- Use CI/CD pipelines for automated deployment
- Implement blue-green or canary deployments
- Keep environment-specific details out of code
- Use proper secret management
- Test configuration in staging environments before production
- Implement health checks and readiness probes

- Use orchestration tools like Kubernetes for scaling and management

Q13: Explain the Spring Boot testing annotations and their uses.

Spring Boot provides extensive support for testing with various annotations to simplify test configuration:

1. Core Testing Annotations:

- **@SpringBootTest**: Creates an application context for integration tests

```
@SpringBootTest
class ApplicationTests {
    @Test
    void contextLoads() {
        // Tests that the application context loads successfully
    }
}
```

- **@WebMvcTest**: Tests Spring MVC controllers without starting a full HTTP server

```
@WebMvcTest(UserController.class)
class UserControllerTest {
    @Autowired
    private MockMvc mockMvc;

    @MockBean
    private UserService userService;

    @Test
    void shouldGetUser() throws Exception {
        when(userService.getUser(1L)).thenReturn(new User(1L, "John"));

        mockMvc.perform(get("/api/users/1"))
               .andExpect(status().isOk())
               .andExpect(jsonPath("$.name").value("John"));
    }
}
```

- **@DataJpaTest**: Tests JPA components with an in-memory database

```
@DataJpaTest
class UserRepositoryTest {
    @Autowired
    private UserRepository userRepository;

    @Test
    void shouldFindByEmail() {
        userRepository.save(new User("test@example.com", "password"));
```

```
        Optional<User> found =
userRepository.findByEmail("test@example.com");
        assertTrue(found.isPresent());
        assertEquals("test@example.com", found.get().getEmail());
    }
}
```

- **@RestClientTest**: Tests REST clients

```
@RestClientTest(RemoteUserService.class)
class RemoteUserServiceTest {
    @Autowired
    private RemoteUserService userService;

    @Autowired
    private MockRestServiceServer server;

    @Test
    void shouldRetrieveUserFromRemoteService() {
        server.expect(requestTo("/api/users/1"))
              .andRespond(withSuccess("{\"id\":1,\"name\":\"John\"}",
MediaType.APPLICATION_JSON));

        User user = userService.getUser(1L);
        assertEquals("John", user.getName());
    }
}
```

- **@WebFluxTest**: Tests WebFlux controllers

```
@WebFluxTest(UserController.class)
class UserControllerTest {
    @Autowired
    private WebTestClient webClient;

    @MockBean
    private UserService userService;

    @Test
    void shouldGetUser() {
        when(userService.getUser(1L)).thenReturn(Mono.just(new User(1L,
"John")));

        webClient.get().uri("/api/users/1")
                .exchange()
                .expectStatus().isOk()
                .expectBody()
                .jsonPath("$.name").isEqualTo("John");
```

```
    }
}
```

- **@JdbcTest**: Tests JDBC components

```
@JdbcTest
@AutoConfigureTestDatabase(replace = AutoConfigureTestDatabase.Replace.NONE)
class JdbcUserRepositoryTest {
    @Autowired
    private JdbcTemplate jdbcTemplate;

    @Autowired
    private JdbcUserRepository userRepository;

    @Test
    void shouldFindUserById() {
        jdbcTemplate.update("INSERT INTO users (id, name) VALUES (?, ?)", 1,
"John");

        User user = userRepository.findById(1L);
        assertEquals("John", user.getName());
    }
}
```

2. Configuration and Mocking Annotations:

- **@MockBean**: Adds mock objects to the Spring context

```
@SpringBootTest
class UserServiceTest {
    @MockBean
    private UserRepository userRepository;

    @Autowired
    private UserService userService;

    @Test
    void shouldCreateUser() {
        User user = new User("test@example.com", "password");
        when(userRepository.save(any(User.class))).thenReturn(user);

        User created = userService.createUser("test@example.com",
"password");
        assertEquals("test@example.com", created.getEmail());

        verify(userRepository).save(any(User.class));
    }
}
```

- **@SpyBean**: Adds spy objects to the Spring context

```
@SpringBootTest
class EmailServiceTest {
    @SpyBean
    private EmailService emailService;

    @Test
    void shouldSendWelcomeEmail() {
        User user = new User("test@example.com", "John");
        emailService.sendWelcomeEmail(user);

        verify(emailService).sendEmail(eq("test@example.com"),
contains("Welcome"));
    }
}
```

- **@TestConfiguration**: Defines additional beans for testing

```
@SpringBootTest
class ApplicationTests {

    @TestConfiguration
    static class TestConfig {
        @Bean
        public TestDataGenerator testDataGenerator() {
            return new TestDataGenerator();
        }
    }

    @Autowired
    private TestDataGenerator testDataGenerator;

    @Test
    void shouldUseTestData() {
        User user = testDataGenerator.generateUser();
        assertNotNull(user);
    }
}
```

- **@Import**: Imports additional configurations

```
@WebMvcTest
@Import(SecurityConfig.class)
class SecuredControllerTest {
    // Test secured endpoints
}
```

3. Property and Profile Annotations:

- **@TestPropertySource**: Configures test properties

```
@SpringBootTest
@TestPropertySource(properties = {
    "spring.datasource.url=jdbc:h2:mem:testdb",
    "spring.jpa.hibernate.ddl-auto=create-drop"
})
class DatabaseTest {
    // Test with specific properties
}
```

- **@ActiveProfiles**: Sets active profiles for testing

```
@SpringBootTest
@ActiveProfiles("test")
class ProfileSpecificTest {
    // Test with test profile active
}
```

4. Web Testing Annotations:

- **@AutoConfigureMockMvc**: Configures MockMvc

```
@SpringBootTest
@AutoConfigureMockMvc
class WebApplicationTest {
    @Autowired
    private MockMvc mockMvc;

    @Test
    void shouldReturnDefaultMessage() throws Exception {
        mockMvc.perform(get("/hello"))
               .andExpect(status().isOk())
               .andExpect(content().string(containsString("Hello, World")));
    }
}
```

- **@AutoConfigureWebTestClient**: Configures WebTestClient

```
@SpringBootTest(webEnvironment = SpringBootTest.WebEnvironment.RANDOM_PORT)
@AutoConfigureWebTestClient
class WebFluxApplicationTest {
    @Autowired
    private WebTestClient webClient;
```

```
    @Test
    void shouldReturnUsers() {
        webClient.get().uri("/api/users")
                .exchange()
                .expectStatus().isOk()
                .expectBodyList(User.class);
    }
}
```

5. Slice Testing Annotations:

- **@JsonTest**: Tests JSON serialization/deserialization

```
@JsonTest
class UserJsonTest {
    @Autowired
    private JacksonTester<User> json;

    @Test
    void shouldSerializeUser() throws Exception {
        User user = new User(1L, "John", "john@example.com");

        JsonContent<User> result = json.write(user);

        assertThat(result).extractingJsonPathStringValue("$.name")
                          .isEqualTo("John");
        assertThat(result).extractingJsonPathStringValue("$.email")
                          .isEqualTo("john@example.com");
    }
}
```

- **@DataMongoTest**: Tests MongoDB components

```
@DataMongoTest
class MongoUserRepositoryTest {
    @Autowired
    private MongoUserRepository userRepository;

    @Test
    void shouldSaveAndRetrieveUser() {
        userRepository.save(new User("John", "john@example.com"));

        User foundUser = userRepository.findByEmail("john@example.com");
        assertEquals("John", foundUser.getName());
    }
}
```

Best practices for Spring Boot testing:

- Use slice tests where appropriate to keep tests fast
- Prefer focused tests with clear assertions
- Use @DirtiesContext wisely as it impacts performance
- Setup test data using @Before or @BeforeEach methods
- Use TestContainers for integration tests with real databases
- Implement custom test configurations for complex scenarios
- Take advantage of Spring Boot's automatic property binding
- Test different environments with profiles
- Document test assumptions with clear method names and comments

Q14: How would you implement distributed tracing in a Spring Boot microservices architecture?

Implementing distributed tracing in Spring Boot microservices involves several components working together to track requests as they flow through multiple services:

1. Add Required Dependencies:

```
<!-- Spring Cloud Sleuth for distributed tracing -->
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-sleuth</artifactId>
</dependency>

<!-- Zipkin integration for trace collection -->
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-sleuth-zipkin</artifactId>
</dependency>

<!-- Optional: Brave OpenTracing integration -->
<dependency>
    <groupId>io.opentracing.brave</groupId>
    <artifactId>brave-opentracing</artifactId>
</dependency>
```

2. Configure Tracing in application.properties/yml:

```
# Application name (used as service name in traces)
spring.application.name=order-service

# Zipkin server connection details
spring.zipkin.base-url=http://zipkin-server:9411
spring.zipkin.service.name=${spring.application.name}

# Sampling configuration
spring.sleuth.sampler.probability=1.0  # 1.0 = 100% of requests traced (use lower
in production)
```

```
# Propagate trace headers across all communications
spring.sleuth.propagation.type=B3,W3C

# Optional: Configure integration with specific components
spring.sleuth.integration.enabled=true
spring.sleuth.web.enabled=true
spring.sleuth.async.enabled=true
spring.sleuth.scheduled.enabled=true
spring.sleuth.messaging.enabled=true
```

3. Set Up a Zipkin Server:

- Using Docker:

```
docker run -d -p 9411:9411 openzipkin/zipkin
```

- Using Kubernetes:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: zipkin
spec:
  replicas: 1
  selector:
    matchLabels:
      app: zipkin
  template:
    metadata:
      labels:
        app: zipkin
    spec:
      containers:
        - name: zipkin
          image: openzipkin/zipkin
          ports:
            - containerPort: 9411
---
apiVersion: v1
kind: Service
metadata:
  name: zipkin
spec:
  ports:
    - port: 9411
      targetPort: 9411
  selector:
    app: zipkin
```

4. Enhanced Tracing in Code:

- Add custom spans for important operations:

```
@Service
public class OrderService {

    private final Tracer tracer;

    public OrderService(Tracer tracer) {
        this.tracer = tracer;
    }

    public Order processOrder(Order order) {
        // Start a new span for order processing
        Span orderProcessingSpan = tracer.buildSpan("process-order").start();

        try (Scope scope = tracer.activateSpan(orderProcessingSpan)) {
            // Add metadata to the span
            orderProcessingSpan.setTag("orderId", order.getId());
            orderProcessingSpan.setTag("customerId", order.getCustomerId());

            // Process the order
            validateOrder(order);
            calculateTotal(order);
            saveOrder(order);

            return order;
        } catch (Exception e) {
            // Log error to span
            orderProcessingSpan.setTag("error", true);
            orderProcessingSpan.log(Map.of(
                "event", "error",
                "error.object", e,
                "message", e.getMessage()
            ));
            throw e;
        } finally {
            // Close the span
            orderProcessingSpan.finish();
        }
    }

    private void validateOrder(Order order) {
        Span validationSpan = tracer.buildSpan("validate-order")
            .asChildOf(tracer.activeSpan())
            .start();
        try (Scope scope = tracer.activateSpan(validationSpan)) {
            // Validation logic
            Thread.sleep(50); // Simulate work
        } catch (Exception e) {
            validationSpan.setTag("error", true);
```

```
            throw e;
        } finally {
            validationSpan.finish();
        }
    }

    // Additional methods with similar tracing
}
```

- Use Spring's TraceableExecutorService for async operations:

```
@Configuration
public class AsyncConfig {

    private final BeanFactory beanFactory;

    public AsyncConfig(BeanFactory beanFactory) {
        this.beanFactory = beanFactory;
    }

    @Bean
    public Executor asyncExecutor() {
        ThreadPoolTaskExecutor executor = new ThreadPoolTaskExecutor();
        executor.setCorePoolSize(5);
        executor.setMaxPoolSize(10);
        executor.setQueueCapacity(25);
        executor.setThreadNamePrefix("Async-");
        executor.initialize();

        return new LazyTraceExecutor(beanFactory, executor);
    }
}
```

5. Trace Context Propagation:

- For RestTemplate:

```
@Bean
public RestTemplate restTemplate() {
    return new RestTemplate(); // Spring Cloud Sleuth automatically handles trace
propagation
}
```

- For WebClient:

```
@Bean
public WebClient webClient() {
```

```
    return WebClient.builder().build(); // Spring Cloud Sleuth automatically
handles trace propagation
}
```

- For Kafka:

```
@Bean
public ProducerFactory<String, String> producerFactory() {
    Map<String, Object> props = new HashMap<>();
    props.put(ProducerConfig.BOOTSTRAP_SERVERS_CONFIG, "kafka:9092");
    props.put(ProducerConfig.KEY_SERIALIZER_CLASS_CONFIG, StringSerializer.class);
    props.put(ProducerConfig.VALUE_SERIALIZER_CLASS_CONFIG,
StringSerializer.class);
    return new DefaultKafkaProducerFactory<>(props);
    // Spring Cloud Sleuth automatically handles trace propagation
}
```

6. MDC Logging Integration:

```
<dependency>
    <groupId>net.logstash.logback</groupId>
    <artifactId>logstash-logback-encoder</artifactId>
    <version>7.0.1</version>
</dependency>
```

```
<!-- logback-spring.xml -->
<configuration>
    <appender name="CONSOLE" class="ch.qos.logback.core.ConsoleAppender">
        <encoder class="net.logstash.logback.encoder.LogstashEncoder">
            <!-- This will include trace IDs in logs -->
        </encoder>
    </appender>

    <root level="INFO">
        <appender-ref ref="CONSOLE" />
    </root>
</configuration>
```

7. OpenTelemetry Integration (Modern Approach):

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

```
    <groupId>io.micrometer</groupId>
    <artifactId>micrometer-tracing-bridge-otel</artifactId>
</dependency>
<dependency>
    <groupId>io.opentelemetry</groupId>
    <artifactId>opentelemetry-exporter-zipkin</artifactId>
</dependency>
```

```
# Configure OpenTelemetry
management.tracing.sampling.probability=1.0
management.zipkin.tracing.endpoint=http://zipkin-server:9411/api/v2/spans
```

8. Monitoring and Visualization:

- Connect Zipkin to persistent storage:

```
# Zipkin with Elasticsearch
version: '3'
services:
  elasticsearch:
    image: docker.elastic.co/elasticsearch/elasticsearch:7.10.0
    environment:
      - discovery.type=single-node
    ports:
      - 9200:9200

  zipkin:
    image: openzipkin/zipkin
    environment:
      - STORAGE_TYPE=elasticsearch
      - ES_HOSTS=elasticsearch:9200
    ports:
      - 9411:9411
    depends_on:
      - elasticsearch
```

- Integrate with Grafana:

```
grafana:
  image: grafana/grafana
  ports:
    - 3000:3000
  volumes:
    - ./grafana/provisioning:/etc/grafana/provisioning
  depends_on:
    - zipkin
```

9. Best Practices:

- **Optimize sampling rates** for production: Use lower rates (e.g., 0.1) for high-traffic services

```
spring.sleuth.sampler.probability=0.1
```

- **Filter sensitive data** from being traced:

```
@Component
public class SensitiveDataFilter extends GenericFilterBean {
    @Override
    public void doFilter(ServletRequest request, ServletResponse response,
FilterChain chain)
            throws IOException, ServletException {
        // Remove sensitive headers or query parameters
        HttpServletRequest httpRequest = (HttpServletRequest) request;

        // Create wrapper that filters sensitive data
        HttpServletRequestWrapper wrappedRequest = new
HttpServletRequestWrapper(httpRequest) {
            @Override
            public String getHeader(String name) {
                if ("Authorization".equalsIgnoreCase(name)) {
                    return "REDACTED";
                }
                return super.getHeader(name);
            }
        };

        chain.doFilter(wrappedRequest, response);
    }
}
```

- **Use semantic naming conventions** for traces and spans

```
// Follow pattern: service-name/operation-name
Span span = tracer.buildSpan("order-service/payment-processing").start();
```

- **Add business context** to spans:

```
span.setTag("customer.type", customer.getType());
span.setTag("order.value", order.getTotalAmount());
span.setTag("payment.method", payment.getMethod());
```

- **Implement trace-based debugging flags**:

```
if (tracer.activeSpan() != null &&
    Boolean.TRUE.equals(tracer.activeSpan().getBaggageItem("debug"))) {
    // Perform extra debugging
    log.debug("Detailed debugging for order: {}", order);
}
```

Q15: How would you optimize the performance of a Spring Boot application?

Optimizing Spring Boot applications involves various strategies targeting different aspects of the application:

1. JVM Optimization:

- **Heap size tuning**:

```
java -Xms2g -Xmx2g -jar myapp.jar
```

- **Garbage collector selection**:

```
# Use G1GC (recommended for modern applications)
java -XX:+UseG1GC -jar myapp.jar

# For low-latency applications
java -XX:+UseZGC -jar myapp.jar
```

- **Monitor and optimize GC behavior**:

```
java -XX:+PrintGCDetails -XX:+PrintGCDateStamps -Xloggc:/path/to/gc.log -jar
myapp.jar
```

- **Class data sharing**:

```
# Create a shared archive
java -Xshare:dump -XX:SharedArchiveFile=app-classes.jsa -jar myapp.jar

# Use the shared archive
java -Xshare:on -XX:SharedArchiveFile=app-classes.jsa -jar myapp.jar
```

2. Spring Boot Configuration:

- **Enable lazy initialization** to reduce startup time:

```
spring.main.lazy-initialization=true
```

- **Optimize BeanDefinitionLoader**:

```
spring.main.banner-mode=off
```

- **Use profiles for custom configurations**:

```
spring.profiles.active=prod
```

- **Set sensible connection pool sizes**:

```
spring.datasource.hikari.maximum-pool-size=10
spring.datasource.hikari.minimum-idle=5
spring.datasource.hikari.idle-timeout=600000
```

3. Database Optimization:

- **Indexing strategically**:

```
CREATE INDEX idx_customer_email ON customers(email);
```

- **Optimize queries** with proper JPA and Hibernate settings:

```
spring.jpa.properties.hibernate.jdbc.batch_size=30
spring.jpa.properties.hibernate.order_inserts=true
spring.jpa.properties.hibernate.order_updates=true
spring.jpa.properties.hibernate.batch_versioned_data=true

# Disable unnecessary features
spring.jpa.open-in-view=false
spring.jpa.properties.hibernate.jdbc.lob.non_contextual_creation=true
```

- **Use projections** for specific data needs:

```
public interface OrderSummary {
    Long getId();
    BigDecimal getTotal();
    String getStatus();
}

@Repository
```

```
public interface OrderRepository extends JpaRepository<Order, Long> {
    List<OrderSummary> findByCustomerId(Long customerId);
}
```

- **Native queries** for complex operations:

```
@Query(value = "SELECT * FROM orders o JOIN order_items i ON o.id =
i.order_id " +
               "WHERE i.product_id = :productId AND o.created_at > :date",
       nativeQuery = true)
List<Order> findOrdersWithProduct(@Param("productId") Long productId,
                                  @Param("date") LocalDateTime date);
```

4. Caching Strategies:

- **Application-level caching**:

```
@Configuration
@EnableCaching
public class CachingConfig {
    @Bean
    public CacheManager cacheManager() {
        CaffeineCacheManager cacheManager = new CaffeineCacheManager();
        cacheManager.setCaffeine(Caffeine.newBuilder()
                .expireAfterWrite(10, TimeUnit.MINUTES)
                .maximumSize(500));
        return cacheManager;
    }
}
```

- **Method-level caching**:

```
@Service
public class ProductService {
    @Cacheable(value = "products", key = "#id")
    public Product getProduct(Long id) {
        // Expensive operation to fetch product
    }

    @CacheEvict(value = "products", key = "#product.id")
    public void updateProduct(Product product) {
        // Update logic
    }

    @CacheEvict(value = "products", allEntries = true)
    public void refreshCache() {
        // Clear entire cache
```

```
    }
}
```

- **Use HTTP caching** for REST APIs:

```
@GetMapping("/products/{id}")
public ResponseEntity<Product> getProduct(@PathVariable Long id) {
    Product product = productService.getProduct(id);

    // Cache for 1 hour using ETag
    return ResponseEntity.ok()
            .cacheControl(CacheControl.maxAge(1, TimeUnit.HOURS))
            .eTag(Integer.toString(product.getVersion()))
            .body(product);
}
```

5. Network and Web Optimization:

- **Enable compression**:

```
server.compression.enabled=true
server.compression.min-response-size=1024
server.compression.mime-
types=text/html,text/xml,text/plain,text/css,application/javascript,applicat
ion/json
```

- **Configure connection parameters**:

```
server.tomcat.max-threads=200
server.tomcat.max-connections=8192
server.tomcat.accept-count=100
server.tomcat.connection-timeout=5000
```

- **Use asynchronous processing** for I/O-bound operations:

```
@GetMapping("/reports")
public Callable<ResponseEntity<Report>> generateReport() {
    return () -> {
        Report report = reportService.generateLargeReport();
        return ResponseEntity.ok(report);
    };
}
```

6. Resource Handling:

- **Optimize static resource handling**:

```
spring.web.resources.chain.cache=true
spring.web.resources.chain.strategy.content.enabled=true
spring.web.resources.chain.strategy.fixed.enabled=true
spring.web.resources.chain.strategy.fixed.paths=/**
spring.web.resources.chain.strategy.fixed.version=v12
```

- **Configure resource cache period**:

```
spring.web.resources.cache.period=3600
```

- **File upload configuration**:

```
spring.servlet.multipart.max-file-size=10MB
spring.servlet.multipart.max-request-size=10MB
spring.servlet.multipart.file-size-threshold=2KB
```

7. Logging Optimization:

- **Configure appropriate log levels**:

```
logging.level.root=WARN
logging.level.org.springframework.web=INFO
logging.level.com.example.myapp=DEBUG
```

- **Use asynchronous appenders**:

```
<appender name="ASYNC" class="ch.qos.logback.classic.AsyncAppender">
    <appender-ref ref="FILE"/>
    <queueSize>512</queueSize>
    <discardingThreshold>0</discardingThreshold>
</appender>
```

- **Log rotation**:

```
logging.file.name=myapp.log
logging.file.max-size=10MB
logging.file.max-history=10
```

8. Monitoring and Profiling:

- **Enable Micrometer metrics**:

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-actuator</artifactId>
</dependency>
<dependency>
    <groupId>io.micrometer</groupId>
    <artifactId>micrometer-registry-prometheus</artifactId>
</dependency>
```

- **Configure Actuator endpoints**:

```
management.endpoints.web.exposure.include=health,info,metrics,prometheus
management.metrics.export.prometheus.enabled=true
```

- **Use Flight Recorder for profiling**:

```
java -XX:+FlightRecorder -
XX:StartFlightRecording=duration=60s,filename=recording.jfr -jar myapp.jar
```

9. Code-Level Optimizations:

- **Avoid N+1 query problems**:

```
// Bad approach
List<Order> orders = orderRepository.findAll();
for (Order order : orders) {
    Customer customer =
customerRepository.findById(order.getCustomerId()).orElse(null);
    // Process order with customer
}

// Good approach
@Query("SELECT o FROM Order o JOIN FETCH o.customer")
List<Order> findAllWithCustomer();
```

- **Use pagination** for large datasets:

```
@GetMapping("/products")
public Page<Product> getProducts(
        @RequestParam(defaultValue = "0") int page,
        @RequestParam(defaultValue = "20") int size) {
```

```
    return productRepository.findAll(PageRequest.of(page, size));
}
```

- **Optimize serialization/deserialization**:

```
@JsonInclude(JsonInclude.Include.NON_NULL)
public class ProductDto {
    // Fields...
}
```

- **Use non-blocking I/O** for high concurrency:

```
@GetMapping("/products/{id}")
public Mono<Product> getProduct(@PathVariable String id) {
    return productRepository.findById(id);
}
```

10. Advanced Techniques:

- **Implement database sharding**:

```
@Configuration
public class ShardingDataSourceConfig {
    @Bean
    @Primary
    public DataSource shardingDataSource() {
        Map<String, DataSource> dataSourceMap = new HashMap<>();
        dataSourceMap.put("shard1", shard1DataSource());
        dataSourceMap.put("shard2", shard2DataSource());

        // Configure routing strategy
        return new ShardingDataSource(dataSourceMap, shardingStrategy());
    }
}
```

- **Consider reactive programming** for I/O-bound applications:

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

    @Bean
    public RouterFunction<ServerResponse> routes(ProductHandler handler) {
```

```
        return RouterFunctions.route()
                .GET("/products", handler::getAllProducts)
                .GET("/products/{id}", handler::getProduct)
                .POST("/products", handler::createProduct)
                .build();
    }
}
```

- **Implement CQRS pattern** for read/write segregation:

```
@Service
public class ProductQueryService {
    private final ProductReadRepository readRepository;

    // Methods optimized for reading
}

@Service
public class ProductCommandService {
    private final ProductWriteRepository writeRepository;
    private final EventPublisher eventPublisher;

    // Methods for write operations
}
```

Performance testing and validation:

- Use JMeter or Gatling for load testing
- Monitor memory usage with tools like VisualVM
- Analyze thread dumps with tools like FastThread
- Enable flight recordings with JDK Mission Control
- Implement APM solutions like New Relic or Dynatrace
- Use Spring Boot Actuator metrics with Prometheus and Grafana

Dependency Reference for Common Requirements

Requirement	Dependency	Description
Web Applications	`spring-boot-starter-web`	Build web applications with Spring MVC and embedded Tomcat
Reactive Web	`spring-boot-starter-webflux`	Build reactive web applications with Spring WebFlux
Data Access	`spring-boot-starter-data-jpa`	Access relational databases with Spring Data JPA and Hibernate
	`spring-boot-starter-data-mongodb`	Access MongoDB document databases
	`spring-boot-starter-data-redis`	Work with Redis key-value store

Requirement	Dependency	Description
	`spring-boot-starter-data-elasticsearch`	Search with Elasticsearch
	`spring-boot-starter-jdbc`	Low-level database access with JDBC
Security	`spring-boot-starter-security`	Add security to your application
	`spring-boot-starter-oauth2-client`	OAuth2 client support
	`spring-boot-starter-oauth2-resource-server`	OAuth2 resource server support
Testing	`spring-boot-starter-test`	Comprehensive testing utilities
Monitoring	`spring-boot-starter-actuator`	Add production-ready features for monitoring and management
	`micrometer-registry-prometheus`	Export metrics to Prometheus
Templating	`spring-boot-starter-thymeleaf`	Thymeleaf templating engine
	`spring-boot-starter-freemarker`	FreeMarker templating engine
	`spring-boot-starter-mustache`	Mustache templating engine
Messaging	`spring-boot-starter-amqp`	Message processing with RabbitMQ
	`spring-boot-starter-activemq`	Message processing with ActiveMQ
	`spring-boot-starter-integration`	Spring Integration for enterprise integration patterns
	`spring-kafka`	Kafka messaging
Validation	`spring-boot-starter-validation`	Bean validation with Hibernate Validator
Caching	`spring-boot-starter-cache`	Spring's caching abstraction
	`cache-api`	JSR-107 (JCache) support
Cloud	`spring-cloud-starter`	Basic Spring Cloud support
	`spring-cloud-starter-config`	Spring Cloud Config client
	`spring-cloud-starter-netflix-eureka-client`	Service discovery with Eureka
	`spring-cloud-starter-netflix-zuul`	API gateway with Zuul
	`spring-cloud-starter-sleuth`	Distributed tracing
	`spring-cloud-starter-zipkin`	Zipkin tracing

Requirement	Dependency	Description
	`spring-cloud-starter-circuitbreaker-resilience4j`	Circuit breaking with Resilience4j
Batch Processing	`spring-boot-starter-batch`	Batch processing
Scheduling	`spring-boot-starter-quartz`	Quartz scheduler
WebSockets	`spring-boot-starter-websocket`	WebSocket support
Mail	`spring-boot-starter-mail`	Email sending
Documentation	`springdoc-openapi-ui`	OpenAPI documentation
Development Tools	`spring-boot-devtools`	Development-time tools and auto-restart
GraphQL	`spring-boot-starter-graphql`	GraphQL with Spring Boot
Observability	`micrometer-tracing-bridge-brave`	Distributed tracing with Brave

Migration Guide from Traditional Spring to Spring Boot

Understanding the Differences

Traditional Spring	Spring Boot
Manual configuration through XML or Java configuration classes	Auto-configuration based on classpath and properties
External server deployment (WAR files)	Self-contained applications (JAR files) with embedded servers
Manual dependency management	Starter dependencies that bundle compatible libraries
Manual configuration of connection pools, transaction managers, etc.	Sensible defaults with minimal configuration
Manual setup of common patterns and functionality	Production-ready features built-in

Step-by-Step Migration Process

1. Assess Your Current Application:

- Identify Spring Framework version
- Document existing XML configurations
- List Java-based configuration classes
- Catalog external dependencies
- Identify custom infrastructure code

2. Set Up a Spring Boot Project:

- Create a new Spring Boot project:

```
<parent>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-parent</artifactId>
    <version>2.7.0</version>
</parent>
```

- Add core dependencies:

```
<dependencies>
    <!-- Replace individual Spring dependencies with starters -->
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-web</artifactId>
    </dependency>
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-data-jpa</artifactId>
    </dependency>
    <!-- Other starters as needed -->
</dependencies>
```

- Create the main application class:

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

3. Migrate Configuration:

- Replace XML configurations with application.properties/yaml:

```
# Database configuration (previously in XML)
spring.datasource.url=jdbc:mysql://localhost:3306/myapp
spring.datasource.username=root
spring.datasource.password=password

# JPA configuration
spring.jpa.hibernate.ddl-auto=update
spring.jpa.properties.hibernate.dialect=org.hibernate.dialect.MySQL8Dialect
```

```
# Server configuration
server.port=8080
```

- Keep custom @Configuration classes but simplify them:

```
// Before (traditional Spring)
@Configuration
public class DatabaseConfig {
    @Bean
    public DataSource dataSource() {
        DriverManagerDataSource dataSource = new DriverManagerDataSource();
        dataSource.setDriverClassName("com.mysql.cj.jdbc.Driver");
        dataSource.setUrl("jdbc:mysql://localhost:3306/myapp");
        dataSource.setUsername("root");
        dataSource.setPassword("password");
        return dataSource;
    }

    @Bean
    public LocalContainerEntityManagerFactoryBean entityManagerFactory() {
        // Complex configuration...
    }

    @Bean
    public PlatformTransactionManager transactionManager() {
        // Transaction manager config...
    }
}

// After (Spring Boot)
@Configuration
public class CustomDatabaseConfig {
    // Only override what needs customization
    @Bean
    public DataSourceProperties dataSourceProperties() {
        return new DataSourceProperties();
    }
}
```

- For XML configurations that must be retained, import them:

```
@SpringBootApplication
@ImportResource("classpath:legacy-context.xml")
public class Application {
    public static void main(String[] args) {
        SpringApplication.run(Application.class, args);
    }
}
```

4. Migrate Components:

- Keep existing @Component, @Service, @Repository, and @Controller classes
- Update Spring MVC controllers to use simpler annotations:

```
// Before
@Controller
@RequestMapping("/users")
public class UserController {
    @RequestMapping(value = "/{id}", method = RequestMethod.GET)
    public ModelAndView getUser(@PathVariable Long id) {
        // Implementation
    }
}

// After
@RestController
@RequestMapping("/users")
public class UserController {
    @GetMapping("/{id}")
    public User getUser(@PathVariable Long id) {
        // Implementation
    }
}
```

- Simplify exception handling:

```
@RestControllerAdvice
public class GlobalExceptionHandler {
    @ExceptionHandler(ResourceNotFoundException.class)
    public ResponseEntity<ErrorResponse>
handleResourceNotFound(ResourceNotFoundException ex) {
        return ResponseEntity.status(HttpStatus.NOT_FOUND)
                .body(new ErrorResponse(ex.getMessage()));
    }
}
```

5. Update Data Access Layer:

- Replace custom repository implementations with Spring Data interfaces:

```
// Before
@Repository
public class JdbcUserRepository implements UserRepository {
    private JdbcTemplate jdbcTemplate;

    public User findById(Long id) {
        // Complex JDBC code
```

```
    }
}

// After
public interface UserRepository extends JpaRepository<User, Long> {
    // Spring Data handles implementation
    User findByEmail(String email);
}
```

- Update entity classes if needed:

```
// Before
@Entity
@Table(name = "users")
public class User {
    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    private Long id;
    // Other fields and methods
}

// After - often no changes needed
@Entity
@Table(name = "users")
public class User {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;
    // Other fields and methods
}
```

6. Migrate Testing:

- Update test dependencies:

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

- Simplify test classes:

```
// Before
@RunWith(SpringJUnit4ClassRunner.class)
@ContextConfiguration(classes = {TestConfig.class})
public class UserServiceTest {
```

```
    @Autowired
    private UserService userService;

    @Test
    public void testFindById() {
        // Test implementation
    }
}

// After
@SpringBootTest
public class UserServiceTest {
    @Autowired
    private UserService userService;

    @Test
    void testFindById() {
        // Test implementation
    }
}
```

- Use sliced tests for focused testing:

```
@WebMvcTest(UserController.class)
public class UserControllerTest {
    @Autowired
    private MockMvc mockMvc;

    @MockBean
    private UserService userService;

    @Test
    void getUserById() throws Exception {
        // Test implementation
    }
}
```

7. Update Build & Deployment:

- Add Spring Boot Maven Plugin:

```
<build>
    <plugins>
        <plugin>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-maven-plugin</artifactId>
        </plugin>
    </plugins>
</build>
```

- Convert from WAR to JAR if possible:

```
# Traditional Spring
mvn clean package  # Produces a WAR

# Spring Boot
mvn clean package  # Produces an executable JAR
java -jar target/myapp-0.0.1-SNAPSHOT.jar
```

- If WAR deployment is still needed:

```
<packaging>war</packaging>

<dependencies>
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-tomcat</artifactId>
        <scope>provided</scope>
    </dependency>
</dependencies>
```

```
@SpringBootApplication
public class Application extends SpringBootServletInitializer {
    @Override
    protected SpringApplicationBuilder configure(SpringApplicationBuilder
application) {
        return application.sources(Application.class);
    }

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

8. Add Production-Ready Features:

- Enable Actuator:

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

```
management.endpoints.web.exposure.include=health,info,metrics
```

- Configure logging:

```
logging.level.root=WARN
logging.level.com.example=INFO
logging.file.name=myapp.log
```

- Implement health checks:

```
@Component
public class DatabaseHealthIndicator implements HealthIndicator {
    private final DataSource dataSource;

    public DatabaseHealthIndicator(DataSource dataSource) {
        this.dataSource = dataSource;
    }

    @Override
    public Health health() {
        try (Connection conn = dataSource.getConnection()) {
            // Perform health check
            return Health.up().build();
        } catch (Exception e) {
            return Health.down().withException(e).build();
        }
    }
}
```

9. Test and Refine:

- Run the application with different profiles:

```
java -jar myapp.jar --spring.profiles.active=dev
```

- Monitor startup logs for auto-configuration report
- Address any issues in test environment
- Review actuator endpoints to identify potential issues
- Gradually refactor complex components to leverage Spring Boot features

10. Advanced Migration:

- Replace legacy security configurations:

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

- Implement externalized configuration:

```
@ConfigurationProperties(prefix = "app")
@Data
public class ApplicationProperties {
    private String name;
    private Notification notification = new Notification();

    @Data
    public static class Notification {
        private boolean enabled = true;
        private String endpoint;
    }
}
```

- Consider migrating to Spring Cloud for microservices:

```
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-netflix-eureka-client</artifactId>
</dependency>
```

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

Troubleshooting Guide for Common Issues

Application Startup Issues

Problem: Application fails to start with "Port already in use" error

Solution:

- Check if another process is using the port:

```
# Linux/Mac
lsof -i :8080

# Windows
netstat -ano | findstr 8080
```

- Configure a different port:

```
server.port=8081
```

- Programmatically use a random port:

```
server.port=0
```

Problem: Bean creation failure

Solution:

- Check dependency conflicts:

```
mvn dependency:tree
```

- Verify auto-configuration conditions:

```
# Show auto-configuration report
debug=true
```

- Check circular dependencies:

```
// Use constructor injection instead of field injection
@Service
public class UserService {
```

```
    private final EmailService emailService;

    public UserService(EmailService emailService) {
        this.emailService = emailService;
    }
}
```

Problem: "No qualifying bean of type" error

Solution:

- Check component scanning:

```
@SpringBootApplication(scanBasePackages = "com.example")
```

- Verify bean definition:

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

- Enable debug logging:

```
logging.level.org.springframework=DEBUG
```

Database Issues

Problem: "Cannot get a connection" error

Solution:

- Check database connectivity:

```
spring.datasource.url=jdbc:mysql://localhost:3306/mydb
spring.datasource.username=root
spring.datasource.password=password
```

- Verify database is running
- Check connection pool settings:

```
spring.datasource.hikari.maximum-pool-size=10
spring.datasource.hikari.connection-timeout=30000
```

Problem: Hibernate schema issues

Solution:

- Control schema generation:

```
spring.jpa.hibernate.ddl-auto=validate
```

- Use database migration tools:

```
<dependency>
    <groupId>org.flywaydb</groupId>
    <artifactId>flyway-core</artifactId>
</dependency>
```

```
src/main/resources/db/migration/V1__init.sql
```

- Enable SQL logging:

```
spring.jpa.show-sql=true
spring.jpa.properties.hibernate.format_sql=true
```

Problem: "LazyInitializationException"

Solution:

- Fetch eager when needed:

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

- Use fetch joins in queries:

```
@Query("SELECT o FROM Order o JOIN FETCH o.items WHERE o.id = :id")
Order findByIdWithItems(@Param("id") Long id);
```

- Consider disabling Open Session in View (recommended):

```
spring.jpa.open-in-view=false
```

Performance Issues

Problem: Slow application startup

Solution:

- Use lazy initialization:

```
spring.main.lazy-initialization=true
```

- Limit component scanning:

```
@SpringBootApplication(scanBasePackages = "com.example.core")
```

- Profile startup:

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

Problem: High memory usage

Solution:

- Tune JVM settings:

```
java -Xms1g -Xmx1g -XX:+UseG1GC -jar myapp.jar
```

- Check for memory leaks using tools like VisualVM
- Monitor heap dumps:

```
java -XX:+HeapDumpOnOutOfMemoryError -XX:HeapDumpPath=/tmp/heapdump.hprof -
jar myapp.jar
```

- Use lightweight dependencies:

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

```
    <exclusions>
        <exclusion>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-tomcat</artifactId>
        </exclusion>
    </exclusions>
</dependency>
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-undertow</artifactId>
</dependency>
```

Problem: Slow response times

Solution:

- Implement caching:

```
@Cacheable("products")
public Product findById(Long id) {
    // Implementation
}
```

- Use asynchronous processing:

```
@Async
public CompletableFuture<List<Product>> findAllAsync() {
    List<Product> products = productRepository.findAll();
    return CompletableFuture.completedFuture(products);
}
```

- Profile database queries:

```
logging.level.org.hibernate.SQL=DEBUG
logging.level.org.hibernate.type.descriptor.sql.BasicBinder=TRACE
```

Security Issues

Problem: CSRF protection preventing form submission

Solution:

- Include CSRF token in forms:

```
```

```
<form method="post" action="/submit">
  <input type="hidden" name="${_csrf.parameterName}" value="${_csrf.token}"
/>
  <!-- Other form fields -->
</form>
```

- Disable CSRF for specific endpoints (API endpoints with tokens):

```
@Configuration
public class SecurityConfig extends WebSecurityConfigurerAdapter {
    @Override
    protected void configure(HttpSecurity http) throws Exception {
        http
            .csrf()
                .ignoringAntMatchers("/api/**")
            .and()
            // Other configuration
    }
}
```

Problem: CORS issues

Solution:

- Enable CORS globally:

```
@Configuration
public class WebConfig implements WebMvcConfigurer {
    @Override
    public void addCorsMappings(CorsRegistry registry) {
        registry.addMapping("/api/**")
                .allowedOrigins("https://example.com")
                .allowedMethods("GET", "POST", "PUT", "DELETE")
                .allowedHeaders("*")
                .allowCredentials(true);
    }
}
```

- Enable CORS for specific controllers:

```
@RestController
@RequestMapping("/api/products")
@CrossOrigin(origins = "https://example.com")
public class ProductController {
    // Controller methods
}
```

Problem: Unauthorized access despite security configuration

Solution:

- Check security debug logs:

```
logging.level.org.springframework.security=DEBUG
```

- Verify security order of operations:

```
@Override
protected void configure(HttpSecurity http) throws Exception {
    // Most specific rules first
    http.authorizeRequests()
        .antMatchers("/admin/**").hasRole("ADMIN")
        .antMatchers("/user/**").hasRole("USER")
        .antMatchers("/public/**").permitAll()
        // Most general rules last
        .anyRequest().authenticated();
}
```

- Check role prefixes (Spring Security adds "ROLE_" prefix):

```
@PreAuthorize("hasRole('ADMIN')") // Looks for ROLE_ADMIN
public void adminMethod() {
    // Implementation
}
```

Configuration Issues

Problem: Properties not being applied

Solution:

- Check property names and case:

```
# Correct
spring.datasource.url=jdbc:mysql://localhost:3306/mydb

# Incorrect
spring.datasource.URL=jdbc:mysql://localhost:3306/mydb
```

- Verify property source order:

```
java -jar myapp.jar --debug | grep "PropertySource"
```

- Use property binding debugging:

```
logging.level.org.springframework.boot.context.config=TRACE
```

Problem: Profiles not activating correctly

Solution:

- Verify profile activation:

```
java -jar myapp.jar --spring.profiles.active=prod
```

- Check profile-specific properties:

```
application-prod.properties
```

- Use programmatic activation:

```
SpringApplication app = new SpringApplication(MyApp.class);
app.setAdditionalProfiles("prod");
app.run(args);
```

- Verify with actuator:

```
GET /actuator/env
```

Problem: Auto-configuration not working as expected

Solution:

- Enable auto-configuration report:

```
debug=true
```

- Check for missing dependencies:

```

```

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

- Ensure you're not excluding necessary auto-configurations:

```
@EnableAutoConfiguration(exclude = {DataSourceAutoConfiguration.class})
```

Testing Issues

Problem: Tests failing to load context

Solution:

- Use appropriate test slices:

```
@WebMvcTest(UserController.class)
public class UserControllerTest {
    // Test methods
}
```

- Mock required beans:

```
@MockBean
private UserService userService;
```

- Verify configuration:

```
@SpringBootTest(properties = "spring.profiles.active=test")
public class ApplicationTests {
    // Test methods
}
```

Problem: Tests interacting with production systems

Solution:

- Use test properties:

```
# application-test.properties
spring.datasource.url=jdbc:h2:mem:testdb
```

- Configure test containers:

```
@SpringBootTest
@Testcontainers
public class IntegrationTests {
    @Container
    static PostgreSQLContainer<?> postgres = new PostgreSQLContainer<>
("postgres:13");

    @DynamicPropertySource
    static void databaseProperties(DynamicPropertyRegistry registry) {
        registry.add("spring.datasource.url", postgres::getJdbcUrl);
        registry.add("spring.datasource.username", postgres::getUsername);
        registry.add("spring.datasource.password", postgres::getPassword);
    }
}
```

- Mock external services:

```
@Test
void testExternalService() {
    mockServer.expect(requestTo("https://api.example.com/data"))
        .andRespond(withSuccess("{\"result\":\"success\"}",
MediaType.APPLICATION_JSON));

    // Test code
}
```

Problem: Transactional tests not rolling back

Solution:

- Add @Transactional to test class/method:

```
@SpringBootTest
@Transactional
public class UserServiceIntegrationTest {
    // Test methods will automatically roll back
}
```

- Check if using incompatible features:

```
@Transactional
@Sql("/test-data.sql") // This will run before transaction starts
public void testUserService() {
```

```
    // Test code
}
```

- Consider manual cleanup:

```
@AfterEach
void cleanup() {
    userRepository.deleteAll();
}
```

Deployment Issues

Problem: Application not finding configuration in production

Solution:

- Externalize configuration:

```
java -jar myapp.jar --
spring.config.location=file:/etc/myapp/application.properties
```

- Use environment variables:

```
export SPRING_DATASOURCE_URL=jdbc:mysql://prod-db:3306/mydb
export SPRING_DATASOURCE_USERNAME=app_user
export SPRING_DATASOURCE_PASSWORD=secure_password
```

- Use Spring Cloud Config:

```
spring.cloud.config.uri=https://config-server.example.com
spring.cloud.config.name=myapp
spring.cloud.config.profile=prod
```

Problem: JVM memory issues in containers

Solution:

- Set container-aware memory settings:

```
java -XX:+UseContainerSupport -XX:MaxRAMPercentage=75.0 -jar myapp.jar
```

- Use Docker memory limits:

```
docker run -m 512m myapp:latest
```

- Monitor container metrics with actuator:

```
management.metrics.export.prometheus.enabled=true
```

Problem: Health checks failing in containers

Solution:

- Implement proper health checks:

```
@Component
public class DatabaseHealthIndicator implements HealthIndicator {
    @Override
    public Health health() {
        // Implementation
    }
}
```

- Configure appropriate timeouts:

```
management.endpoint.health.show-details=always
management.health.db.enabled=true
management.health.diskspace.enabled=true
```

- Use container-specific health checks:

```
# Docker Compose
healthcheck:
  test: ['CMD', 'curl', '-f', 'http://localhost:8080/actuator/health']
  interval: 30s
  timeout: 10s
  retries: 3
```

Recommended Resources for Continued Learning

Official Documentation

- Spring Boot Reference Documentation
- Spring Framework Documentation
- Spring Projects

- Spring Guides
- Spring Blog

Books

- "Spring Boot in Action" by Craig Walls
- "Spring Microservices in Action" by John Carnell
- "Cloud Native Java" by Josh Long and Kenny Bastani
- "Testing Spring Boot: Applications with JUnit, Mockito and Spring Boot Test" by Michael Simons
- "Spring Security in Action" by Laurentiu Spilca

Online Courses

- Spring Framework 5: Beginner to Guru
- Building Microservices with Spring Boot and Spring Cloud
- Testing Spring Boot: Beginner to Guru
- REST with Spring

Blogs and Websites

- Baeldung Spring Tutorials
- Reflectoring Blog
- Spring Boot Tutorial by Mkyong
- Spring Framework Guru
- DZone Spring Zone
- Spring Development Team at Medium

YouTube Channels

- Spring Developer
- JavaBrains
- Amigoscode
- Java Techie
- In28Minutes

GitHub Repositories

- Spring Boot Samples
- Spring PetClinic
- Spring Boot Microservices Example
- Spring Boot Shopping Cart

Community and Support

- Stack Overflow - Spring Boot
- Spring Community Forums
- Spring Boot Gitter
- Spring Boot Slack

Podcasts

- Spring Boot Podcast
- A Bootiful Podcast
- Java Off Heap
- Conversations about Software Engineering

Conferences and Events

- SpringOne
- Spring I/O
- Devoxx
- JFokus
- Java Day

Conclusion

This comprehensive guide has taken you from the fundamentals of Spring Boot to advanced topics and best practices. By now, you should have a solid understanding of how Spring Boot simplifies Java application development through its convention-over-configuration approach, auto-configuration, and production-ready features.

We've covered:

1. **Spring Boot Fundamentals**: How Spring Boot evolved from the traditional Spring Framework, its core principles, project structure, and auto-configuration mechanism.
2. **Building REST APIs**: Creating controllers, handling requests and responses, validation, exception handling, and documenting your APIs.
3. **Data Access**: Working with various databases using Spring Data, implementing repositories, and managing transactions.
4. **Security**: Implementing authentication and authorization, protecting endpoints, and securing your application against common threats.
5. **Microservices**: Building distributed systems with Spring Cloud, implementing service discovery, API gateway, circuit breakers, and distributed configuration.
6. **Testing**: Writing unit, integration, and end-to-end tests, using test slices, and implementing test containers for realistic testing.
7. **Production Deployment**: Packaging applications, containerization, Kubernetes deployment, monitoring, and performance tuning.
8. **Advanced Topics**: Reactive programming, WebSockets, caching strategies, batch processing, scheduled tasks, GraphQL, and creating custom starters.

Spring Boot continues to evolve, with each new release bringing improvements and new features. The best way to stay current is to regularly check the official Spring documentation, follow the Spring blog, and participate in the community through forums and social media.

Remember that Spring Boot is designed to help you focus on business logic rather than boilerplate configuration. As you build more applications, you'll discover additional ways to leverage Spring Boot's capabilities to create robust, maintainable, and production-ready applications.

Happy coding with Spring Boot!

Appendix: Spring Boot Version History and Migration

Spring Boot Version History

Version	Release Date	Major Features	Spring Framework Version
1.0.0	April 2014	Initial release, auto-configuration, embedded servers	4.0
1.1.0	June 2014	JTA support, improved templating	4.0.5
1.2.0	March 2015	Metrics, health checks, enhanced logging	4.1.3
1.3.0	December 2015	Fully executable JARs, dev tools	4.2.3
1.4.0	July 2016	Improved testing support, Couchbase support	4.3.2
1.5.0	January 2017	Kafka support, Cloud Foundry discovery	4.3.6
2.0.0	March 2018	Java 8 baseline, reactive programming, Kotlin support	5.0.4
2.1.0	October 2018	JDK 11 support, actuator endpoints, improved logging	5.1.2
2.2.0	October 2019	Java 13 support, immutable configuration properties	5.2.0
2.3.0	May 2020	Java 14 support, Graceful shutdown, Validation starter	5.2.6
2.4.0	November 2020	Java 15 support, Docker compose support	5.3.1
2.5.0	May 2021	Java 16 support, Gradle 7, Improved startup	5.3.7
2.6.0	November 2021	Java 17 support, New AOT engine, Circular references detection	5.3.13
2.7.0	May 2022	Java 18 support, Micrometer observation API	5.3.20
3.0.0	November 2022	Java 17 baseline, Jakarta EE 10, GraalVM native, AOT compilation	6.0.0
3.1.0	May 2023	GraalVM improvements, CRaC support	6.0.9

Version	Release Date	Major Features	Spring Framework Version
3.2.0	November 2023	Java 21 support, Enhanced observability, Virtual threads	6.1.0

Migration Between Major Versions

Migrating from Spring Boot 1.x to 2.x

Key Changes:

1. Java 8+ is required
2. Third-party library upgrades (Tomcat 8.5+, Hibernate 5.2+)
3. Spring Framework 5.0
4. Configuration property binding changes
5. Actuator endpoint changes

Migration Steps:

1. Update Java version to 8+
2. Update Spring Boot version
3. Update third-party dependencies
4. Migrate configuration properties
5. Update Actuator endpoints
6. Test thoroughly

Migrating from Spring Boot 2.x to 3.x

Key Changes:

1. Java 17+ is required
2. Jakarta EE instead of Java EE
3. Third-party library upgrades
4. Spring Framework 6.0
5. AOT compilation for GraalVM native images
6. Observability changes

Migration Steps:

1. Update Java version to 17+
2. Migrate from Java EE to Jakarta EE packages

```
// Before
import javax.servlet.http.HttpServletRequest;

// After
import jakarta.servlet.http.HttpServletRequest;
```

3. Update Spring Boot version
4. Update third-party dependencies
5. Test with the Spring Boot Migrator tool
6. Validate auto-configuration changes
7. Test thoroughly

Java Version Compatibility

Spring Boot Version	Minimum Java Version	Maximum Java Version
1.0.x - 1.5.x	Java 6	Java 8
2.0.x - 2.3.x	Java 8	Java 14
2.4.x - 2.5.x	Java 8	Java 16
2.6.x - 2.7.x	Java 8	Java 18
3.0.x - 3.1.x	Java 17	Java 20
3.2.x+	Java 17	Java 21

Useful Migration Tools

1. **Spring Boot Migrator**: A tool that helps automate migration between Spring Boot versions

```
java -jar spring-boot-migrator.jar
```

2. **OpenRewrite**: Automated source code refactoring

```
<plugin>
    <groupId>org.openrewrite.maven</groupId>
    <artifactId>rewrite-maven-plugin</artifactId>
    <version>4.38.0</version>
    <configuration>
        <activeRecipes>

<recipe>org.openrewrite.java.spring.boot3.SpringBoot2To3Migration</recipe>
        </activeRecipes>
    </configuration>
</plugin>
```

3. **Dependency Analysis Tools**:

```
mvn dependency:analyze
mvn versions:display-dependency-updates
```

4. **Spring Boot Actuator Compatibility Verifier**:

```
@Bean
public ApplicationStartupAware compatibilityVerifier() {
    return new CompatibilityVerifier();
}
```

Continuous Migration Tips

1. Keep dependencies updated regularly
2. Follow deprecation warnings
3. Read release notes for each version
4. Use feature flags for gradual migration
5. Maintain comprehensive test coverage
6. Consider using the Spring Boot Starter BOM for dependency management
7. Follow Spring projects on GitHub for early access to changes
8. Participate in milestone and RC testing

By following these guidelines and understanding the changes between versions, you can ensure smoother migrations between Spring Boot versions and continue to leverage the latest features and improvements.