Microservice Design Patterns in Spring Boot

Complete Guide and Reference

Table of Contents

Introduction

Microservices architecture involves decomposing applications into small, independent services that communicate over well-defined APIs. Spring Boot provides excellent support for implementing various microservice patterns through its ecosystem of projects, particularly Spring Cloud.

This guide covers the essential design patterns that are crucial for building robust, scalable microservice architectures using Spring Boot.

---

1. Service Discovery Pattern

Purpose

Enables services to find and communicate with each other dynamically without hardcoding network locations. This is essential in cloud environments where service instances can start, stop, or move frequently.

Implementation in Spring Boot

Netflix Eureka

- **Eureka Server**: Acts as a service registry
- **Eureka Client**: Services register themselves and discover others

```java
// Eureka Server Configuration
@SpringBootApplication
@EnableEurekaServer
public class EurekaServerApplication {
    public static void main(String[] args) {
        SpringApplication.run(EurekaServerApplication.class, args);
    }
}

// Eureka Client Configuration
@SpringBootApplication
@EnableEurekaClient
public class UserServiceApplication {
    public static void main(String[] args) {
        SpringApplication.run(UserServiceApplication.class, args);
    }
}
```

Spring Cloud Consul

Alternative service discovery mechanism with additional features like health checking and key-value store.

Key Components

- **Service Registry**: Central database of available services
- **Service Registration**: Process where services register themselves
- **Service Discovery**: Mechanism for services to find each other
- **Health Monitoring**: Continuous health checks of registered services

Benefits

- Dynamic scaling capabilities
- Fault tolerance through automatic service deregistration

- Load balancing across multiple service instances
- Zero-downtime deployments

Configuration Example

```yaml
# application.yml for Eureka Client
eureka:
  client:
    service-url:
      defaultZone: http://localhost:8761/eureka/
  instance:
    prefer-ip-address: true
    lease-renewal-interval-in-seconds: 30
```

---

2. API Gateway Pattern

Purpose

Provides a single entry point for all client requests, acting as a reverse proxy that routes requests to appropriate microservices while handling cross-cutting concerns.

Implementation Options

Spring Cloud Gateway

Modern, reactive gateway built on Spring WebFlux:

```java
```

```
@Configuration
public class GatewayConfig {

    @Bean
    public RouteLocator customRouteLocator(RouteLocatorBuilder builder) {
        return builder.routes()
            .route("user-service", r -> r.path("/users/**")
                .uri("lb://user-service"))
            .route("order-service", r -> r.path("/orders/**")
                .uri("lb://order-service"))
            .build();
    }
}
```

Netflix Zuul (Legacy)

Proxy-based gateway, now in maintenance mode but still widely used.

Key Features

- **Request Routing**: Direct requests to appropriate services
- **Load Balancing**: Distribute requests across service instances
- **Authentication & Authorization**: Centralized security handling
- **Rate Limiting**: Prevent abuse and ensure fair usage
- **Request/Response Transformation**: Modify requests/responses as needed
- **Monitoring & Analytics**: Centralized logging and metrics collection

Benefits

- Simplified client interactions
- Centralized cross-cutting concerns
- Protocol translation (HTTP to WebSocket, etc.)
- Reduced client complexity
- Better security posture

Configuration Example

```
yaml
```

```
# Gateway routing configuration
spring:
  cloud:
    gateway:
      routes:
        - id: user-service
          uri: lb://user-service
          predicates:
            - Path=/api/users/**
          filters:
            - StripPrefix=2
            - AddRequestHeader=X-Request-Source, gateway
```

---

3. Circuit Breaker Pattern

Purpose

Prevents cascading failures by monitoring service calls and "opening the circuit" when failure rates exceed configured thresholds, providing graceful degradation.

Implementation Options

Resilience4j (Recommended)

Modern resilience library with Spring Boot integration:

java

```
@Service
public class UserService {

  @CircuitBreaker(name = "user-service", fallbackMethod = "fallbackUser")
  @Retry(name = "user-service")
  @TimeLimiter(name = "user-service")
  public CompletableFuture<User> getUserById(Long id) {
    return CompletableFuture.supplyAsync(() -> {
      // External service call
      return userClient.getUser(id);
    });
  }

  public CompletableFuture<User> fallbackUser(Long id, Exception ex) {
    return CompletableFuture.completedFuture(new User(id, "Default User"));
  }
}
```

Netflix Hystrix (Legacy)

Original circuit breaker implementation, now in maintenance mode.

Circuit States

1. **Closed**: Normal operation, requests flow through
2. **Open**: Failure threshold exceeded, requests fail fast with fallback
3. **Half-Open**: Testing phase to check if service has recovered

Configuration

```
yaml
```

```
# Resilience4j configuration
resilience4j:
  circuitbreaker:
    instances:
      user-service:
        failure-rate-threshold: 50
        wait-duration-in-open-state: 30s
        sliding-window-size: 10
        minimum-number-of-calls: 5
  retry:
    instances:
      user-service:
        max-attempts: 3
        wait-duration: 1s
```

Benefits

- Improved system resilience
- Faster failure detection and response
- Prevents resource exhaustion
- Graceful degradation of functionality
- Better user experience during outages

---

4. Configuration Management Pattern

Purpose

Centralizes configuration management across multiple microservices, enabling dynamic updates without service restarts and environment-specific configurations.

Implementation Options

Spring Cloud Config

Git-backed configuration server:

```
java
```

```
// Config Server
@SpringBootApplication
@EnableConfigServer
public class ConfigServerApplication {
    public static void main(String[] args) {
        SpringApplication.run(ConfigServerApplication.class, args);
    }
}

// Config Client
@RestController
@RefreshScope
public class ConfigController {

    @Value("${app.message:Default Message}")
    private String message;

    @GetMapping("/message")
    public String getMessage() {
        return message;
    }
}
```

Consul Config

Key-value store for configuration with real-time updates.

Kubernetes ConfigMaps

Native Kubernetes configuration management.

Key Features

- **Environment-specific configurations**: Dev, staging, production profiles
- **Dynamic configuration updates**: Runtime configuration changes
- **Configuration versioning**: Git-based version control
- **Encryption support**: Sensitive data protection
- **Profile-based configuration**: Spring profiles integration

Benefits

- Centralized configuration management

- Reduced configuration drift
- Easier environment promotion
- Better security for sensitive configurations
- Audit trail for configuration changes

Configuration Example

yaml

```
# bootstrap.yml for Config Client
spring:
  application:
    name: user-service
  cloud:
    config:
      uri: http://config-server:8888
      profile: development
      label: master
```

---

5. Event-Driven Architecture Patterns

Event Sourcing Pattern

Purpose

Stores all changes to application state as a sequence of events, providing complete audit trail and enabling temporal queries.

Implementation

java

```
@Entity
public class EventStore {
    private String aggregateId;
    private String eventType;
    private String eventData;
    private LocalDateTime timestamp;
    private Long version;
}

@Service
public class OrderEventSourcingService {

    public void processOrder(OrderCreatedEvent event) {
        // Store event
        eventStore.save(event);

        // Publish event
        eventPublisher.publishEvent(event);
    }

    public Order reconstructOrder(String orderId) {
        List<Event> events = eventStore.findByAggregateId(orderId);
        return events.stream()
            .collect(Order::new, Order::apply, Order::merge);
    }
}
```

CQRS (Command Query Responsibility Segregation)

Purpose

Separates read and write operations into different models, optimizing each for their specific use case.

Implementation

```
java
```

```
// Command Side
@Service
public class OrderCommandService {

    public void createOrder(CreateOrderCommand command) {
        Order order = new Order(command);
        orderRepository.save(order);

        // Publish event for read side
        eventPublisher.publishEvent(new OrderCreatedEvent(order));
    }
}

// Query Side
@Service
public class OrderQueryService {

    public OrderView getOrder(String orderId) {
        return orderViewRepository.findById(orderId);
    }

    @EventHandler
    public void on(OrderCreatedEvent event) {
        OrderView view = new OrderView(event);
        orderViewRepository.save(view);
    }
}
```

Message-Driven Communication

Spring Cloud Stream

java

```java
@EnableBinding(Processor.class)
public class OrderProcessor {

    @StreamListener(Processor.INPUT)
    @SendTo(Processor.OUTPUT)
    public OrderProcessedEvent processOrder(OrderCreatedEvent event) {
        // Process order logic
        return new OrderProcessedEvent(event.getOrderId());
    }
}
```

Benefits

- Loose coupling between services
- Scalability through asynchronous processing
- Audit trail and compliance
- Temporal queries and point-in-time recovery
- Better performance through optimized read/write models

---

6. Data Management Patterns

Database per Service Pattern

Purpose

Each microservice owns and manages its own data, ensuring loose coupling and independent evolution.

Implementation Considerations

- **Service-specific data stores**: Choose optimal database type per service (SQL, NoSQL, Graph, etc.)
- **Data ownership**: Clear boundaries of data responsibility
- **API-based data access**: No direct database access between services

```java
```

```
// User Service with its own database
@Entity
@Table(name = "users")
public class User {
    @Id
    private Long id;
    private String username;
    private String email;
}

// Order Service with its own database
@Entity
@Table(name = "orders")
public class Order {
    @Id
    private Long id;
    private Long userId; // Reference, not foreign key
    private BigDecimal amount;
}
```

Saga Pattern

Purpose

Manages distributed transactions across multiple services without traditional ACID transactions.

Choreography-Based Saga

Services coordinate through events:

```
java
```

```
@Service
public class OrderSagaService {

    @EventHandler
    public void on(OrderCreatedEvent event) {
        try {
            paymentService.processPayment(event.getPaymentInfo());
            eventPublisher.publishEvent(new PaymentRequestedEvent(event.getOrderId()));
        } catch (Exception e) {
            eventPublisher.publishEvent(new OrderCancelledEvent(event.getOrderId()));
        }
    }

    @EventHandler
    public void on(PaymentProcessedEvent event) {
        inventoryService.reserveItems(event.getOrderId());
        eventPublisher.publishEvent(new InventoryReservedEvent(event.getOrderId()));
    }
}
```

Orchestration-Based Saga

Central coordinator manages the flow:

```
java
```

```
@Service
public class OrderSagaOrchestrator {

    public void processOrder(Order order) {
        SagaTransaction saga = new SagaTransaction(order.getId());

        try {
            // Step 1: Process payment
            PaymentResult payment = paymentService.processPayment(order);
            saga.addCompensation(() -> paymentService.refund(payment.getId()));

            // Step 2: Reserve inventory
            ReservationResult reservation = inventoryService.reserve(order.getItems());
            saga.addCompensation(() -> inventoryService.cancelReservation(reservation.getId()));

            // Step 3: Confirm order
            orderService.confirmOrder(order.getId());

        } catch (Exception e) {
            saga.compensate(); // Execute compensating transactions
            throw new OrderProcessingException("Order processing failed", e);
        }
    }
}
```

Benefits

- Data independence and autonomy
- Technology diversity (polyglot persistence)
- Scalability per service needs
- Fault isolation
- Eventual consistency across services

---

7. Communication Patterns

Synchronous Communication

REST APIs

Traditional HTTP-based request-response pattern:

```java
@RestController
@RequestMapping("/api/users")
public class UserController {

    @GetMapping("/{id}")
    public ResponseEntity<User> getUser(@PathVariable Long id) {
        User user = userService.findById(id);
        return ResponseEntity.ok(user);
    }

    @PostMapping
    public ResponseEntity<User> createUser(@RequestBody CreateUserRequest request) {
        User user = userService.create(request);
        return ResponseEntity.status(HttpStatus.CREATED).body(user);
    }
}
```

OpenFeign Client

Declarative REST client:

```java
```

```
@FeignClient(name = "user-service", fallback = UserServiceFallback.class)
public interface UserServiceClient {

    @GetMapping("/api/users/{id}")
    User getUser(@PathVariable("id") Long id);

    @PostMapping("/api/users")
    User createUser(@RequestBody CreateUserRequest request);
}

@Component
public class UserServiceFallback implements UserServiceClient {

    @Override
    public User getUser(Long id) {
        return new User(id, "Default User", "default@example.com");
    }

    @Override
    public User createUser(CreateUserRequest request) {
        throw new ServiceUnavailableException("User service is currently unavailable");
    }
}
```

WebClient (Reactive)

Non-blocking, reactive HTTP client:

```
java
```

```
@Service
public class UserServiceClient {

    private final WebClient webClient;

    public UserServiceClient(WebClient.Builder webClientBuilder) {
        this.webClient = webClientBuilder
            .baseUrl("http://user-service")
            .build();
    }

    public Mono<User> getUser(Long id) {
        return webClient.get()
            .uri("/api/users/{id}", id)
            .retrieve()
            .bodyToMono(User.class)
            .onErrorReturn(new User(id, "Default User", "default@example.com"));
    }
}
```

Asynchronous Communication

Message Queues with RabbitMQ

```
java
```

```
@Configuration
@EnableRabbit
public class RabbitConfig {

    @Bean
    public Queue orderQueue() {
        return QueueBuilder.durable("order.queue").build();
    }

    @Bean
    public TopicExchange orderExchange() {
        return new TopicExchange("order.exchange");
    }

    @Bean
    public Binding orderBinding() {
        return BindingBuilder
            .bind(orderQueue())
            .to(orderExchange())
            .with("order.created");
    }
}

// Message Producer
@Service
public class OrderEventPublisher {

    @Autowired
    private RabbitTemplate rabbitTemplate;

    public void publishOrderCreated(Order order) {
        OrderCreatedEvent event = new OrderCreatedEvent(order);
        rabbitTemplate.convertAndSend("order.exchange", "order.created", event);
    }
}

// Message Consumer
@RabbitListener(queues = "order.queue")
public void handleOrderCreated(OrderCreatedEvent event) {
    // Process order creation
    log.info("Processing order: {}", event.getOrderId());
}
```

Event Streaming with Apache Kafka

```java
@Configuration
@EnableKafka
public class KafkaConfig {

    @Bean
    public ProducerFactory<String, Object> producerFactory() {
        Map<String, Object> props = new HashMap<>();
        props.put(ProducerConfig.BOOTSTRAP_SERVERS_CONFIG, "localhost:9092");
        props.put(ProducerConfig.KEY_SERIALIZER_CLASS_CONFIG, StringSerializer.class);
        props.put(ProducerConfig.VALUE_SERIALIZER_CLASS_CONFIG, JsonSerializer.class);
        return new DefaultKafkaProducerFactory<>(props);
    }

    @Bean
    public KafkaTemplate<String, Object> kafkaTemplate() {
        return new KafkaTemplate<>(producerFactory());
    }
}

// Event Producer
@Service
public class OrderEventProducer {

    @Autowired
    private KafkaTemplate<String, Object> kafkaTemplate;

    public void publishOrderEvent(OrderEvent event) {
        kafkaTemplate.send("order-events", event.getOrderId(), event);
    }
}

// Event Consumer
@KafkaListener(topics = "order-events", groupId = "payment-service")
public void handleOrderEvent(OrderEvent event) {
    if (event.getType() == EventType.ORDER_CREATED) {
        // Process payment
        paymentService.processPayment(event.getOrderId());
    }
}
```

Benefits

- **Synchronous**: Simple, immediate response, easier debugging
- **Asynchronous**: Better performance, loose coupling, resilience to failures

---

8. Monitoring and Observability Patterns

Distributed Tracing

Purpose

Tracks requests as they flow through multiple services, providing visibility into the entire request lifecycle.

Spring Cloud Sleuth + Zipkin

java

```
// Configuration
@Configuration
public class TracingConfig {

    @Bean
    public Sender sender() {
        return OkHttpSender.create("http://zipkin:9411/api/v2/spans");
    }

    @Bean
    public AsyncReporter<Span> spanReporter() {
        return AsyncReporter.create(sender());
    }
}

// Custom tracing
@Service
public class OrderService {

    private final Tracer tracer;

    public Order processOrder(CreateOrderRequest request) {
        Span span = tracer.nextSpan()
            .name("process-order")
            .tag("order.type", request.getType())
            .start();

        try (Tracer.SpanInScope ws = tracer.withSpanInScope(span)) {
            // Business logic
            Order order = createOrder(request);
            span.tag("order.id", order.getId().toString());
            return order;
        } finally {
            span.end();
        }
    }
}
```

Health Check Pattern

Spring Boot Actuator

```
java
```

```
@Component
public class DatabaseHealthIndicator implements HealthIndicator {

    @Autowired
    private DataSource dataSource;

    @Override
    public Health health() {
        try (Connection connection = dataSource.getConnection()) {
            if (connection.isValid(1)) {
                return Health.up()
                    .withDetail("database", "Available")
                    .withDetail("validationQuery", "SELECT 1")
                    .build();
            }
        } catch (SQLException e) {
            return Health.down()
                .withDetail("database", "Unavailable")
                .withException(e)
                .build();
        }

        return Health.down()
            .withDetail("database", "Connection invalid")
            .build();
    }
}

// Custom health endpoint
@RestController
public class HealthController {

    @Autowired
    private HealthEndpoint healthEndpoint;

    @GetMapping("/health/custom")
    public Map<String, Object> customHealth() {
        Health health = healthEndpoint.health();
        Map<String, Object> response = new HashMap<>();
        response.put("status", health.getStatus().getCode());
        response.put("details", health.getDetails());
        response.put("timestamp", Instant.now());
        return response;
    }
}
```

Centralized Logging

Structured Logging with Logback

xml

```xml
<!-- logback-spring.xml -->
<configuration>
    <appender name="STDOUT" class="ch.qos.logback.core.ConsoleAppender">
        <encoder class="net.logstash.logback.encoder.LoggingEventCompositeJsonEncoder">
            <providers>
                <timestamp/>
                <logLevel/>
                <loggerName/>
                <message/>
                <mdc/>
                <arguments/>
                <stackTrace/>
            </providers>
        </encoder>
    </appender>

    <root level="INFO">
        <appender-ref ref="STDOUT"/>
    </root>
</configuration>
```

java

```
// Structured logging in service
@Service
public class OrderService {

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

    public Order createOrder(CreateOrderRequest request) {
        MDC.put("userId", request.getUserId().toString());
        MDC.put("orderType", request.getType());

        try {
            logger.info("Creating order for user: {}", request.getUserId());

            Order order = new Order(request);
            orderRepository.save(order);

            MDC.put("orderId", order.getId().toString());
            logger.info("Order created successfully");

            return order;
        } catch (Exception e) {
            logger.error("Failed to create order", e);
            throw e;
        } finally {
            MDC.clear();
        }
    }
}
```

Metrics Collection

Micrometer Integration

java

```
@Service
public class OrderMetricsService {

    private final Counter orderCreatedCounter;
    private final Timer orderProcessingTimer;
    private final Gauge activeOrdersGauge;

    public OrderMetricsService(MeterRegistry meterRegistry) {
        this.orderCreatedCounter = Counter.builder("orders.created")
            .description("Number of orders created")
            .register(meterRegistry);

        this.orderProcessingTimer = Timer.builder("orders.processing.duration")
            .description("Order processing duration")
            .register(meterRegistry);

        this.activeOrdersGauge = Gauge.builder("orders.active")
            .description("Number of active orders")
            .register(meterRegistry, this, OrderMetricsService::getActiveOrderCount);
    }

    public Order processOrder(CreateOrderRequest request) {
        return orderProcessingTimer.recordCallable(() -> {
            Order order = createOrder(request);
            orderCreatedCounter.increment(
                Tags.of("type", request.getType(), "status", "success")
            );
            return order;
        });
    }

    private double getActiveOrderCount() {
        return orderRepository.countByStatus(OrderStatus.ACTIVE);
    }
}
```

Benefits

- Complete visibility into system behavior
- Faster problem identification and resolution
- Performance optimization insights
- Compliance and audit requirements

- Proactive issue detection

---

9. Security Patterns

Token-Based Authentication

OAuth 2.0 with Spring Security

java

```
@Configuration
@EnableWebSecurity
@EnableGlobalMethodSecurity(prePostEnabled = true)
public class SecurityConfig {

    @Bean
    public SecurityFilterChain filterChain(HttpSecurity http) throws Exception {
        http
            .oauth2ResourceServer(oauth2 -> oauth2
                .jwt(jwt -> jwt
                    .jwtAuthenticationConverter(jwtAuthenticationConverter())
                )
            )
            .authorizeHttpRequests(authz -> authz
                .requestMatchers("/actuator/health").permitAll()
                .requestMatchers("/api/public/**").permitAll()
                .requestMatchers(HttpMethod.GET, "/api/users/**").hasRole("USER")
                .requestMatchers(HttpMethod.POST, "/api/users/**").hasRole("ADMIN")
                .anyRequest().authenticated()
            );

        return http.build();
    }

    @Bean
    public JwtAuthenticationConverter jwtAuthenticationConverter() {
        JwtGrantedAuthoritiesConverter authoritiesConverter = new JwtGrantedAuthoritiesConverter();
        authoritiesConverter.setAuthorityPrefix("ROLE_");
        authoritiesConverter.setAuthoritiesClaimName("roles");

        JwtAuthenticationConverter converter = new JwtAuthenticationConverter();
        converter.setJwtGrantedAuthoritiesConverter(authoritiesConverter);
        return converter;
    }
}
```

JWT Token Validation

java

```
@Component
public class JwtTokenValidator {

    @Value("${jwt.secret}")
    private String secret;

    @Value("${jwt.expiration}")
    private Long expiration;

    public Claims validateToken(String token) {
        try {
            return Jwts.parserBuilder()
                .setSigningKey(secret.getBytes())
                .build()
                .parseClaimsJws(token)
                .getBody();
        } catch (JwtException e) {
            throw new InvalidTokenException("Invalid JWT token", e);
        }
    }

    public String generateToken(UserDetails userDetails) {
        Map<String, Object> claims = new HashMap<>();
        claims.put("roles", userDetails.getAuthorities().stream()
            .map(GrantedAuthority::getAuthority)
            .collect(Collectors.toList()));

        return Jwts.builder()
            .setClaims(claims)
            .setSubject(userDetails.getUsername())
            .setIssuedAt(new Date())
            .setExpiration(new Date(System.currentTimeMillis() + expiration))
            .signWith(SignatureAlgorithm.HS512, secret.getBytes())
            .compact();
    }
}
```

Service-to-Service Authentication

Mutual TLS (mTLS)

```
java
```

```
@Configuration
public class MtlsConfig {

    @Bean
    public RestTemplate mtlsRestTemplate() throws Exception {
        KeyStore keyStore = KeyStore.getInstance("PKCS12");
        keyStore.load(new FileInputStream("client-keystore.p12"), "password".toCharArray());

        KeyStore trustStore = KeyStore.getInstance("PKCS12");
        trustStore.load(new FileInputStream("client-truststore.p12"), "password".toCharArray());

        SSLContext sslContext = SSLContextBuilder.create()
            .loadKeyMaterial(keyStore, "password".toCharArray())
            .loadTrustMaterial(trustStore, null)
            .build();

        HttpClient httpClient = HttpClients.custom()
            .setSSLContext(sslContext)
            .build();

        HttpComponentsClientHttpRequestFactory requestFactory =
            new HttpComponentsClientHttpRequestFactory(httpClient);

        return new RestTemplate(requestFactory);
    }
}
```

API Rate Limiting

Custom Rate Limiting with Redis

java

```
@Component
public class RateLimitingFilter implements Filter {

    @Autowired
    private RedisTemplate<String, String> redisTemplate;

    private static final int RATE_LIMIT = 100; // requests per minute

    @Override
    public void doFilter(ServletRequest request, ServletResponse response,
                        FilterChain chain) throws IOException, ServletException {

        HttpServletRequest httpRequest = (HttpServletRequest) request;
        String clientId = getClientId(httpRequest);
        String key = "rate_limit:" + clientId;

        String currentCount = redisTemplate.opsForValue().get(key);

        if (currentCount == null) {
            redisTemplate.opsForValue().set(key, "1", Duration.ofMinutes(1));
        } else if (Integer.parseInt(currentCount) >= RATE_LIMIT) {
            HttpServletResponse httpResponse = (HttpServletResponse) response;
            httpResponse.setStatus(HttpStatus.TOO_MANY_REQUESTS.value());
            httpResponse.getWriter().write("Rate limit exceeded");
            return;
        } else {
            redisTemplate.opsForValue().increment(key);
        }

        chain.doFilter(request, response);
    }

    private String getClientId(HttpServletRequest request) {
        // Extract client ID from JWT token or API key
        String authHeader = request.getHeader("Authorization");
        if (authHeader != null && authHeader.startsWith("Bearer ")) {
            // Extract from JWT
            return extractClientIdFromJwt(authHeader.substring(7));
        }
        // Fallback to IP address
        return request.getRemoteAddr();
    }
}
```

Benefits

- Secure service-to-service communication
- Centralized authentication and authorization
- Protection against common attacks (CSRF, XSS, etc.)
- Audit trail for security events
- Compliance with security standards

---

10. Testing Patterns

Contract Testing

Spring Cloud Contract

```groovy
// contracts/user_service_should_return_user.groovy
Contract.make {
    description "should return user by id"
    request {
        method GET()
        url "/api/users/1"
        headers {
            contentType(applicationJson())
        }
    }
    response {
        status OK()
        body([
            id: 1,
            username: "john_doe",
            email: "john@example.com"
        ])
        headers {
            contentType(applicationJson())
        }
    }
}
```

```java
// Provider side test
@SpringBootTest(webEnvironment = SpringBootTest.WebEnvironment.MOCK)
@AutoConfigureStubRunner(ids = "com.example:user-service:+:stubs:8080")
class UserServiceContractTest {

    @MockBean
    private UserRepository userRepository;

    @BeforeEach
    void setUp() {
        User user = new User(1L, "john_doe", "john@example.com");
        when(userRepository.findById(1L)).thenReturn(Optional.of(user));
    }
}

// Consumer side test
@SpringBootTest
@AutoConfigureStubRunner(ids = "com.example:user-service:+:stubs:8080")
class OrderServiceContractTest {

    @Autowired
    private OrderService orderService;

    @Test
    void shouldCreateOrderWithValidUser() {
        CreateOrderRequest request = new CreateOrderRequest(1L, "ELECTRONICS", 100.0);
        Order order = orderService.createOrder(request);

        assertThat(order).isNotNull();
        assertThat(order.getUserId()).isEqualTo(1L);
    }
}
```

Integration Testing

TestContainers

```java
```

```
@SpringBootTest
@Testcontainers
class OrderServiceIntegrationTest {

    @Container
    static PostgreSQLContainer<?> postgres = new PostgreSQLContainer<>("postgres:13")
            .withDatabaseName("testdb")
            .withUsername("test")
            .withPassword("test");

    @Container
    static KafkaContainer kafka = new KafkaContainer(DockerImageName.parse("confluentinc
```