🎯 MICROSERVICES MASTERY ROADMAP

Complete Learning Guide - From Basics to Industry Expert

📋 TABLE OF CONTENTS

📖 INTRODUCTION {#introduction}

Your Current Project Status

You have a Food Delivery System with 3 microservices:

- ☑ **User Service** - JWT Authentication, CRUD operations
- ☑ **Product Service** - Products, Categories, Menus with relationships
- ☑ **Order Service** - Order management with cross-service references

Learning Approach

Each "Story" below is a learning milestone. Complete them in order. Each story has:

- 📚 **LEARN**: Concepts to understand
- 💻 **IMPLEMENT**: Hands-on tasks in YOUR project
- ☑ **VERIFY**: How to confirm you've learned it
- ❓ **INTERVIEW PREP**: Questions you should be able to answer

🏗 PHASE 1: FOUNDATION - Understanding Microservices {#phase-1-foundation}

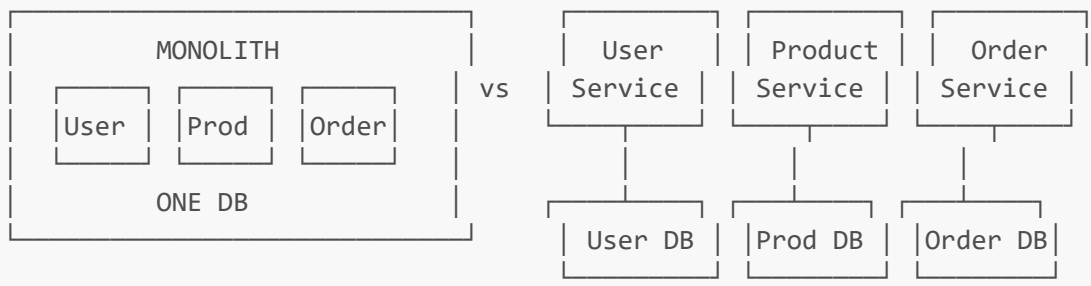
STORY 1.1: What Are Microservices? (Fundamentals)

LEARN

Microservices Architecture is an architectural style where an application is built as a collection of small, independent services that:

- Run in their own process
- Communicate via lightweight mechanisms (usually HTTP/REST or messaging)
- Are independently deployable
- Are organized around business capabilities

Monolith vs Microservices:



12-Factor App Principles:

1. Codebase - One codebase per service
2. Dependencies - Explicitly declare dependencies
3. Config - Store config in environment
4. Backing Services - Treat databases as attached resources
5. Build, Release, Run - Separate stages
6. Processes - Stateless processes
7. Port Binding - Export services via port binding
8. Concurrency - Scale via processes
9. Disposability - Fast startup, graceful shutdown
10. Dev/Prod Parity - Keep environments similar
11. Logs - Treat logs as event streams
12. Admin Processes - Run admin tasks as one-off processes

IMPLEMENT (Your Project)

Task 1.1.1: Document your current architecture

- Open `MICROSERVICES_ARCHITECTURE_DOCUMENTATION.md`
- Draw your current service boundaries
- Identify which 12-factor principles you follow

Task 1.1.2: Verify service independence

- Can you start User Service without Product Service running?
- Can you deploy Order Service independently?

☑ VERIFY

- ☐ I can explain monolith vs microservices trade-offs
- ☐ I understand why each service has its own database
- ☐ I can identify bounded contexts in my domain

? INTERVIEW PREP

1. "What are microservices? Explain with an example."
2. "What are the disadvantages of microservices?"
3. "When should you NOT use microservices?"
4. "Explain the 12-factor app methodology."

STORY 1.2: Domain-Driven Design (DDD) Basics

LEARN

Domain-Driven Design is an approach to software development that focuses on the core domain and domain logic.

Key Concepts:

1. **Bounded Context**: A boundary within which a particular domain model is defined

```
┌─────────────────────────┐   ┌─────────────────────────┐
│   USER CONTEXT          │   │   ORDER CONTEXT         │
│                         │   │                         │
│  User (full entity)     │   │  User (just userId)     │
│  - id                   │   │                         │
│  - name                 │   │  Order                  │
│  - email                │   │  - id                   │
│  - password             │   │  - userId (ref)         │
│  - address              │   │  - productId (ref)      │
└─────────────────────────┘   └─────────────────────────┘
```

2. **Aggregate**: A cluster of entities treated as a single unit

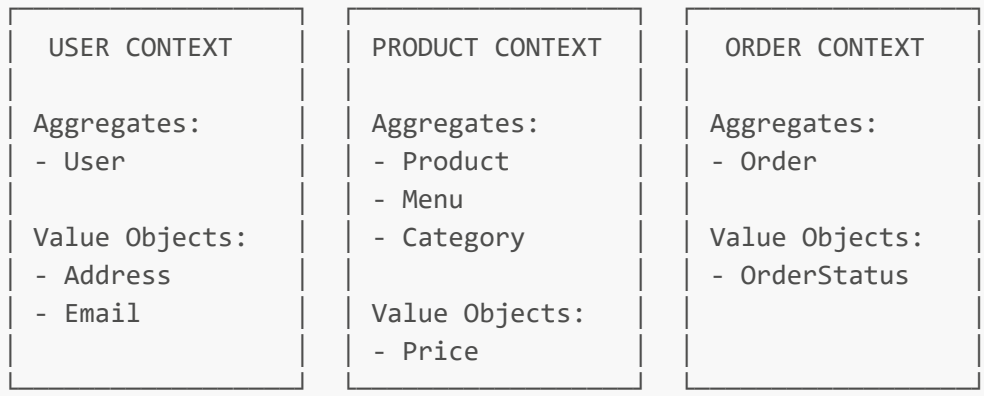
```
Product Aggregate:
┌──────────────────────────────┐
│  Product (Aggregate Root)    │
│  ├── Category (Entity)       │
│  └── Menu (Entity)           │
└──────────────────────────────┘
```

3. **Entities**: Objects with identity (e.g., User with ID)

4. **Value Objects**: Objects without identity (e.g., Address, Money)

5. **Domain Events**: Something that happened in the domain

Your Project's Bounded Contexts:



💻 IMPLEMENT

Task 1.2.1: Create Value Objects

```
// In user-service, create:
public class Email {
    private final String value;

    public Email(String value) {
        if (!value.matches("^[A-Za-z0-9+_.-]+@(.+)$")) {
            throw new IllegalArgumentException("Invalid email");
        }
        this.value = value;
    }
}
```

Task 1.2.2: Create an Address Value Object

- Extract address from User entity into a separate `Address` class
- Make it immutable (all fields final, no setters)

☑ VERIFY

- ☐ I can identify aggregates in my domain
- ☐ I understand bounded context boundaries
- ☐ I know when to use Entity vs Value Object

❓ INTERVIEW PREP

1. "What is Domain-Driven Design?"
2. "Explain Bounded Context with an example."
3. "What is the difference between Entity and Value Object?"

4. "What is an Aggregate Root?"

STORY 1.3: Service Design & API Best Practices

LEARN

REST API Design Principles:

1. **Resource Naming**: Use nouns, not verbs

```
☑ GET /api/users
❌ GET /api/getUsers

☑ POST /api/orders
❌ POST /api/createOrder
```

2. **HTTP Methods & Status Codes:**

Method	Use Case	Success Code
GET	Retrieve resource	200 OK
POST	Create resource	201 Created
PUT	Full update	200 OK
PATCH	Partial update	200 OK
DELETE	Remove resource	204 No Content

3. **Error Response Structure:**

```
{
  "timestamp": "2024-01-15T10:30:00Z",
  "status": 400,
  "error": "Bad Request",
  "message": "Validation failed",
  "path": "/api/users",
  "details": [
    {"field": "email", "message": "must be a valid email"}
  ]
}
```

4. **Versioning Strategies:**
   - URL: `/api/v1/users` ☑ (Most common)
   - Header: `Accept: application/vnd.api.v1+json`
   - Query: `/api/users?version=1`

5. **HATEOAS** (Hypermedia as the Engine of Application State):

```
{
  "id": 1,
  "name": "John",
  "_links": {
    "self": {"href": "/api/users/1"},
    "orders": {"href": "/api/users/1/orders"}
  }
}
```

💻 IMPLEMENT

Task 1.3.1: Create Global Exception Handler

```
@RestControllerAdvice
public class GlobalExceptionHandler {

    @ExceptionHandler(ResourceNotFoundException.class)
    public ResponseEntity<ErrorResponse> handleNotFound(ResourceNotFoundException
ex) {
        ErrorResponse error = new ErrorResponse(
            LocalDateTime.now(),
            HttpStatus.NOT_FOUND.value(),
            "Not Found",
            ex.getMessage()
        );
        return ResponseEntity.status(HttpStatus.NOT_FOUND).body(error);
    }

    @ExceptionHandler(MethodArgumentNotValidException.class)
    public ResponseEntity<ErrorResponse>
handleValidation(MethodArgumentNotValidException ex) {
        // Handle validation errors
    }
}
```

Task 1.3.2: Add API Versioning

- Add `/api/v1/` prefix to all your endpoints
- Update Swagger documentation

Task 1.3.3: Standardize Response Structure

```
public class ApiResponse<T> {
    private boolean success;
    private String message;
    private T data;
```

```
    private LocalDateTime timestamp;
}
```

☑ VERIFY

- ☐ All endpoints return consistent error format
- ☐ API uses proper HTTP status codes
- ☐ Swagger documentation is complete

❓ INTERVIEW PREP

1. "How do you version REST APIs?"
2. "Explain HATEOAS."
3. "What status code do you return for validation errors?"
4. "How do you handle exceptions in Spring Boot microservices?"

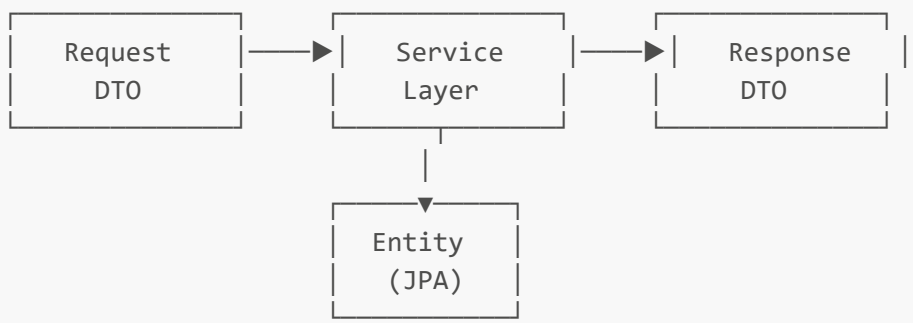
STORY 1.4: DTO Pattern & Data Mapping

📚 LEARN

Why DTOs (Data Transfer Objects)?

- Separate API contract from internal entities
- Hide sensitive fields (password, internal IDs)
- Version API independently
- Optimize network payload

DTO Pattern:



Mapping Options:

1. **Manual Mapping** - Most control
2. **MapStruct** - Compile-time code generation (Recommended)
3. **ModelMapper** - Runtime reflection

💻 IMPLEMENT

Task 1.4.1: Create Request/Response DTOs for User Service

```
// UserRequest.java - For creating/updating
public class UserRequest {
    @NotBlank
    private String name;

    @Email
    private String email;

    @Size(min = 6)
    private String password;

    private String phone;
    private String address;
}

// UserResponse.java - For API responses (NO password!)
public class UserResponse {
    private Long id;
    private String name;
    private String email;
    private String phone;
    private String address;
    private LocalDateTime createdAt;
}
```

Task 1.4.2: Add MapStruct Dependency and Create Mapper

```
<dependency>
    <groupId>org.mapstruct</groupId>
    <artifactId>mapstruct</artifactId>
    <version>1.5.5.Final</version>
</dependency>
```

```
@Mapper(componentModel = "spring")
public interface UserMapper {
    UserResponse toResponse(User user);
    User toEntity(UserRequest request);
    List<UserResponse> toResponseList(List<User> users);
}
```

Task 1.4.3: Apply DTOs to all services

- Product Service: ProductRequest, ProductResponse
- Order Service: OrderRequest, OrderResponse

☑ VERIFY

- ☐ No entity is directly exposed in API
- ☐ Password is never returned in responses
- ☐ All services use DTO pattern

❓ INTERVIEW PREP

1. "What is the DTO pattern and why do we use it?"
2. "How do you map between entities and DTOs?"
3. "What is MapStruct? How is it different from ModelMapper?"

🔧 PHASE 2: CORE PATTERNS - Building Blocks {#phase-2-core-patterns}

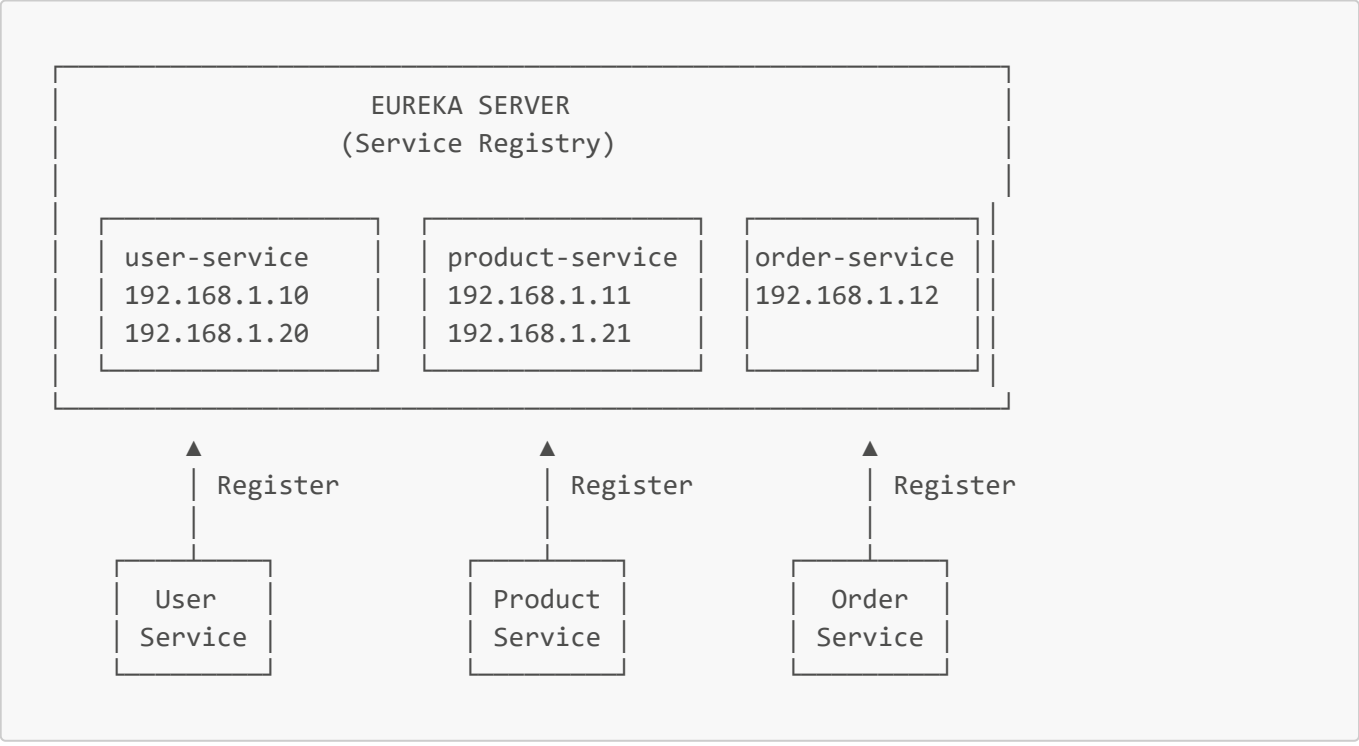
STORY 2.1: Service Discovery with Eureka

📚 LEARN

Problem: In microservices, services need to find each other. Hardcoding URLs doesn't work because:

- Services can have multiple instances
- IPs change in cloud environments
- Load balancing is needed

Solution: Service Discovery Pattern



How it works:

1. Services register themselves with Eureka Server on startup

2. Services send heartbeats every 30 seconds
3. When a service needs another, it queries Eureka for available instances
4. Client-side load balancing picks an instance

💻 IMPLEMENT

Task 2.1.1: Create Discovery Server (New Service)

Create a new Maven project `discovery-server`:

```
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-netflix-eureka-server</artifactId>
</dependency>
```

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

```
# application.properties
server.port=8761
eureka.client.register-with-eureka=false
eureka.client.fetch-registry=false
```

Task 2.1.2: Register All Services as Eureka Clients

Add to each service's pom.xml:

```
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-netflix-eureka-client</artifactId>
</dependency>
```

Add to each service's application.properties:

```
spring.application.name=user-service
eureka.client.service-url.defaultZone=http://localhost:8761/eureka/
```

Task 2.1.3: Verify Registration

- Start Discovery Server
- Start all services
- Visit http://localhost:8761
- Verify all services appear in the dashboard

☑ VERIFY

- ☐ Eureka Server running on port 8761
- ☐ All services registered in Eureka dashboard
- ☐ Services can start/stop independently

? INTERVIEW PREP

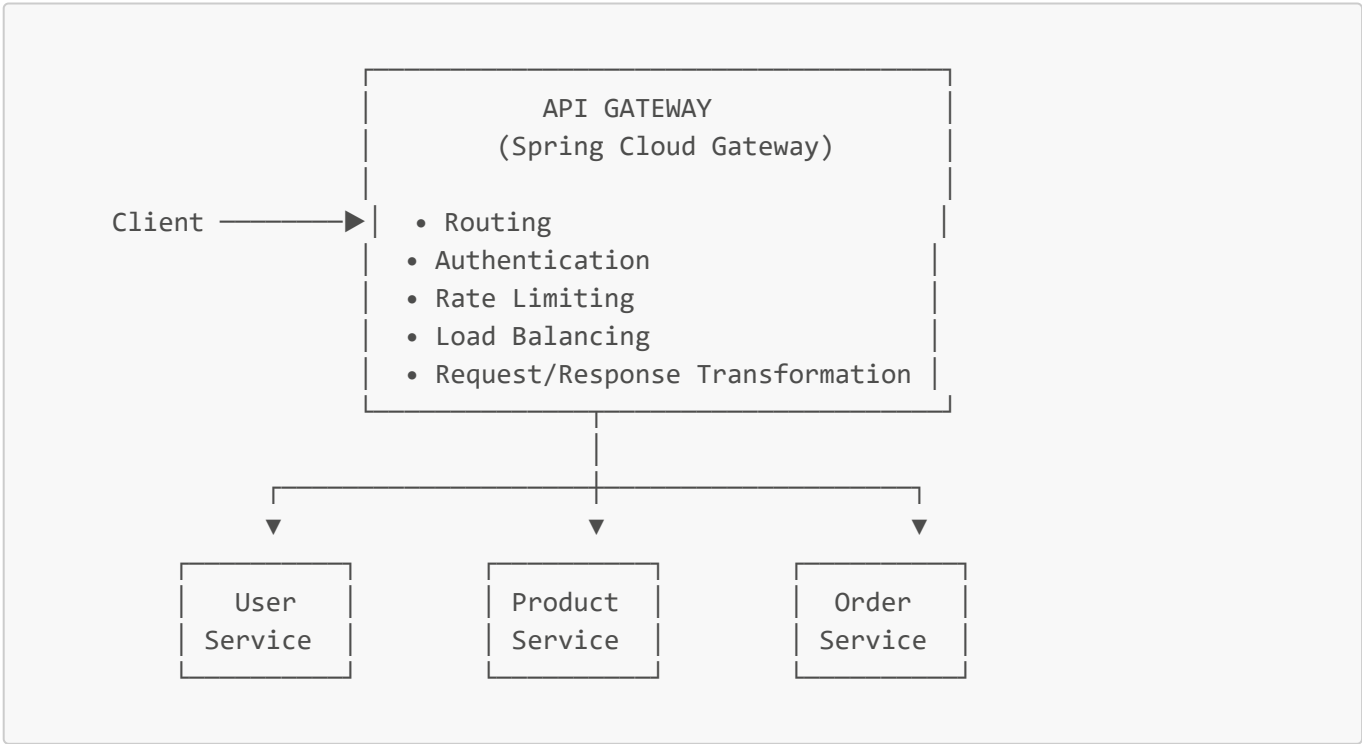
1. "What is Service Discovery? Why do we need it?"
2. "Explain how Eureka works."
3. "What happens if Eureka Server goes down?"
4. "What is the difference between client-side and server-side discovery?"

STORY 2.2: API Gateway Pattern

📚 LEARN

Problem: Clients need to know multiple service URLs, handle authentication at each service, and manage cross-cutting concerns.

Solution: API Gateway - Single entry point for all clients



Spring Cloud Gateway Concepts:

- **Route**: Maps URL path to downstream service
- **Predicate**: Condition to match request (path, header, method)
- **Filter**: Modify request/response (add header, rate limit)

💻 IMPLEMENT

Task 2.2.1: Create API Gateway Service

Create new project `api-gateway`:

```
<dependencies>
    <dependency>
        <groupId>org.springframework.cloud</groupId>
        <artifactId>spring-cloud-starter-gateway</artifactId>
    </dependency>
    <dependency>
        <groupId>org.springframework.cloud</groupId>
        <artifactId>spring-cloud-starter-netflix-eureka-client</artifactId>
    </dependency>
</dependencies>
```

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

Task 2.2.2: Configure Routes

```
# application.yml
server:
  port: 8080

spring:
  application:
    name: api-gateway
  cloud:
    gateway:
      routes:
        - id: user-service
          uri: lb://user-service
          predicates:
            - Path=/api/users/**
          filters:
            - StripPrefix=0
```

```
        - id: product-service
          uri: lb://product-service
          predicates:
            - Path=/api/products/**

        - id: order-service
          uri: lb://order-service
          predicates:
            - Path=/orders/**

eureka:
  client:
    service-url:
      defaultZone: http://localhost:8761/eureka/
```

Task 2.2.3: Add Global Filters

```
@Component
public class LoggingFilter implements GlobalFilter {

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

    @Override
    public Mono<Void> filter(ServerWebExchange exchange, GatewayFilterChain chain)
{
        logger.info("Request: {} {}",
            exchange.getRequest().getMethod(),
            exchange.getRequest().getURI());
        return chain.filter(exchange);
    }
}
```

Task 2.2.4: Test Gateway

- All requests go through `http://localhost:8080`
- `/api/users/**` routes to User Service
- `/api/products/**` routes to Product Service

☑ VERIFY

- ☐ API Gateway running on port 8080
- ☐ All services accessible via gateway
- ☐ Load balancing works with multiple instances

? INTERVIEW PREP

1. "What is an API Gateway? What problems does it solve?"
2. "Explain the difference between API Gateway and Load Balancer."

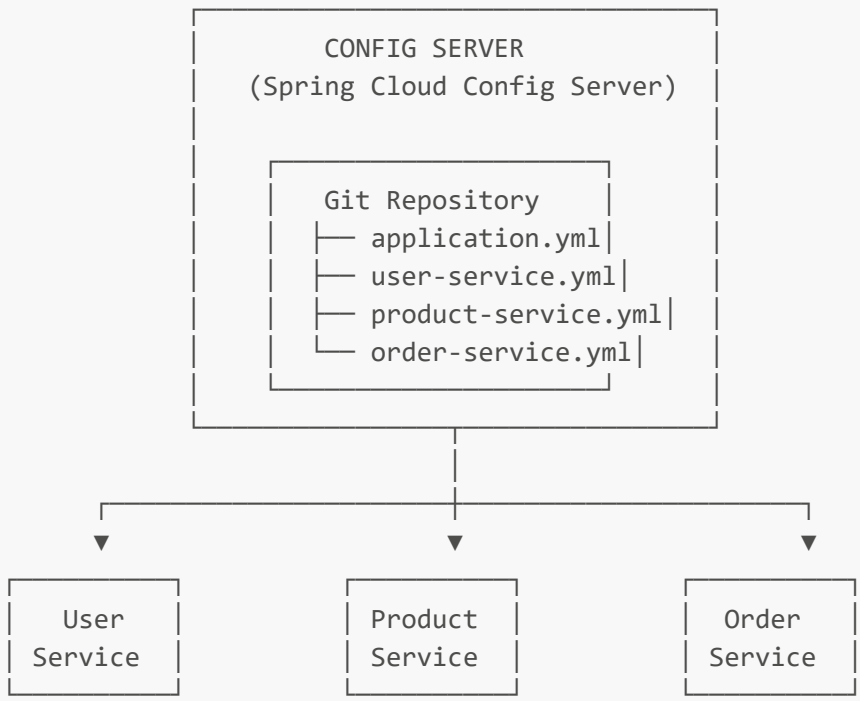
3. "What is Spring Cloud Gateway? How is it different from Zuul?"
4. "How do you handle authentication at the gateway level?"

STORY 2.3: Centralized Configuration

LEARN

Problem: Each service has its own configuration. Managing configs across multiple services and environments is hard.

Solution: Config Server - Centralized configuration management



Benefits:

- Single source of truth for all configurations
- Environment-specific configs (dev, staging, prod)
- Encryption for sensitive values
- Dynamic config refresh without restart

IMPLEMENT

Task 2.3.1: Create Config Repository Create a Git repository with config files:

```
config-repo/
├── application.yml          # Common configs
├── user-service.yml         # User service specific
├── user-service-dev.yml     # Dev profile
├── user-service-prod.yml    # Prod profile
```

```
├── product-service.yml
└── order-service.yml
```

Task 2.3.2: Create Config Server

```xml
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-config-server</artifactId>
</dependency>
```

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

```properties
server.port=8888
spring.cloud.config.server.git.uri=https://github.com/your-repo/config-repo
spring.cloud.config.server.git.default-label=main
```

Task 2.3.3: Configure Clients Add to each service:

```xml
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-config</artifactId>
</dependency>
```

```properties
# bootstrap.properties
spring.application.name=user-service
spring.cloud.config.uri=http://localhost:8888
spring.profiles.active=dev
```

Task 2.3.4: Add @RefreshScope for Dynamic Updates

```java
@RestController
@RefreshScope
public class ConfigTestController {
```

```
    @Value("${custom.message}")
    private String message;

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

☑ VERIFY

- ☐ Config Server running on port 8888
- ☐ Services fetch config from Config Server
- ☐ Can update config without restarting services

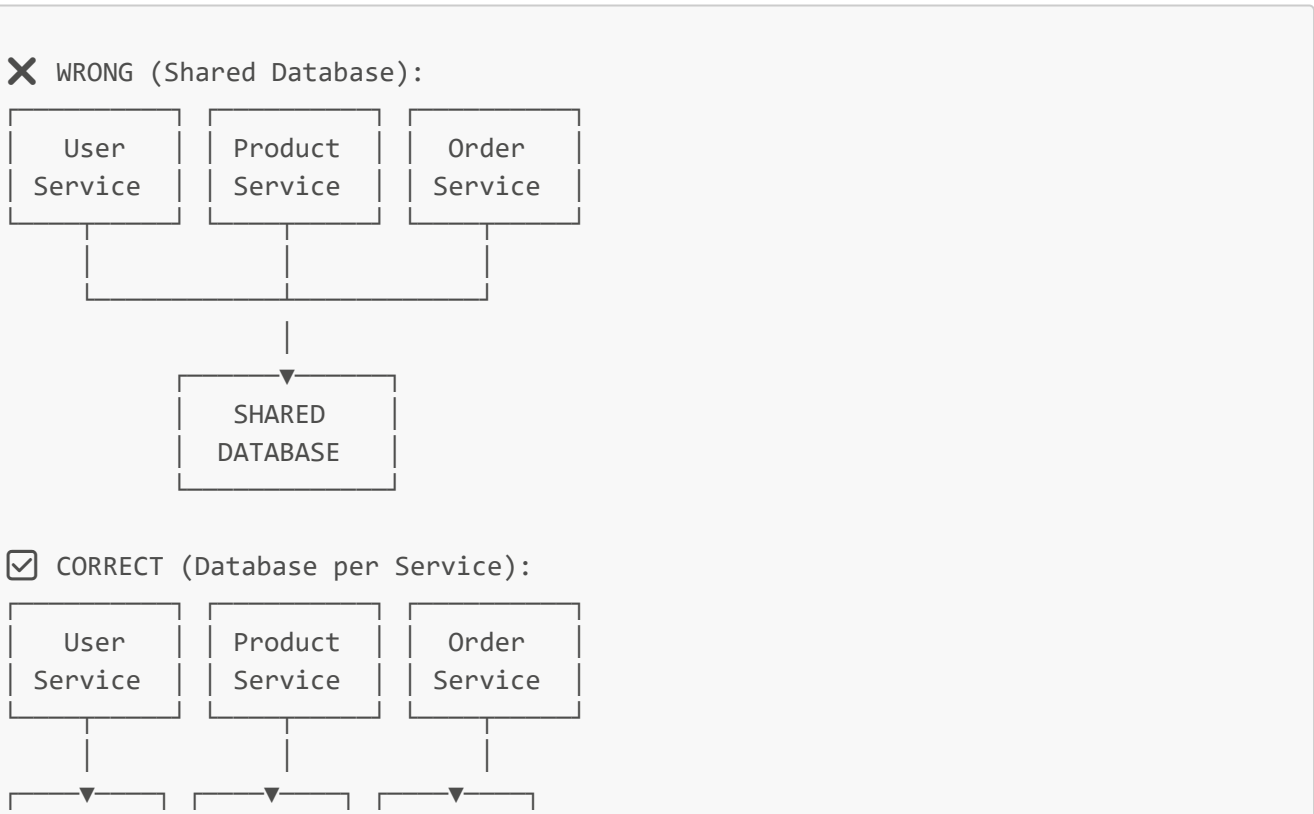
? INTERVIEW PREP

1. "What is Spring Cloud Config?"
2. "How do you manage different configurations for different environments?"
3. "How does @RefreshScope work?"
4. "How do you secure sensitive configuration values?"

STORY 2.4: Database Per Service Pattern

LEARN

Pattern: Each microservice owns its data and database. No direct database access between services.



```
  │ User DB │  │ Prod DB │  │Order DB │
  └─────────┘  └─────────┘  └─────────┘
```

Benefits:

- Services are loosely coupled
- Can use different database technologies (polyglot persistence)
- Independent scaling
- No schema conflicts

Challenges:

- Data consistency across services
- Distributed transactions
- Data duplication

💻 IMPLEMENT

Task 2.4.1: Create Separate Databases

```
CREATE DATABASE food_users;
CREATE DATABASE food_products;
CREATE DATABASE food_orders;
```

Task 2.4.2: Update Each Service's Configuration

```
# user-service application.properties
spring.datasource.url=jdbc:mysql://localhost:3306/food_users

# product-service application.properties
spring.datasource.url=jdbc:mysql://localhost:3306/food_products

# order-service application.properties
spring.datasource.url=jdbc:mysql://localhost:3306/food_orders
```

Task 2.4.3: Verify Isolation

- Each service can only access its own database
- No foreign key constraints across databases

☑ VERIFY

- ☐ Each service has its own database
- ☐ No cross-database joins or foreign keys
- ☐ Services work independently

❓ INTERVIEW PREP

1. "Why does each microservice need its own database?"
2. "What is polyglot persistence?"
3. "How do you handle transactions across multiple services?"
4. "What are the challenges of database per service pattern?"

📡 PHASE 3: COMMUNICATION PATTERNS {#phase-3-communication-patterns}

STORY 3.1: Synchronous Communication (REST + Feign)

📚 LEARN

Synchronous Communication: Service A calls Service B and waits for response.

OpenFeign: Declarative REST client that simplifies service-to-service calls.

```
Order Service needs User info:

┌──────────────┐                    HTTP Request                    ┌──────────────┐
│    Order     │ ──────────────────────────────────────────────▶ │     User     │
│   Service    │ ◀────────────────────────────────────────────── │   Service    │
└──────────────┘                    HTTP Response                   └──────────────┘
      │
      │  Uses FeignClient interface
      ▼
@FeignClient(name = "user-service")
public interface UserServiceClient {
    @GetMapping("/api/users/{id}")
    UserResponse getUser(@PathVariable Long id);
}
```

Why Feign over RestTemplate?

- Declarative (just define interface)
- Automatic load balancing with Eureka
- Built-in circuit breaker support
- Less boilerplate code

💻 IMPLEMENT

Task 3.1.1: Add Feign to Order Service

```
<dependency>
    <groupId>org.springframework.cloud</groupId>
```

```
    <artifactId>spring-cloud-starter-openfeign</artifactId>
</dependency>
```

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

Task 3.1.2: Create Feign Clients

```
@FeignClient(name = "user-service")
public interface UserServiceClient {

    @GetMapping("/api/users/{id}")
    ResponseEntity<UserResponse> getUserById(@PathVariable("id") Long id);
}

@FeignClient(name = "product-service")
public interface ProductServiceClient {

    @GetMapping("/api/products/{id}")
    ResponseEntity<ProductResponse> getProductById(@PathVariable("id") Long id);
}
```

Task 3.1.3: Create Enhanced Order Response

```
public class OrderDetailResponse {
    private Long orderId;
    private String status;
    private UserResponse user;      // Full user details
    private ProductResponse product; // Full product details
}
```

Task 3.1.4: Update Order Service to Fetch Related Data

```
@Service
public class OrderService {

    @Autowired
    private UserServiceClient userClient;

    @Autowired
```

```
    private ProductServiceClient productClient;

    public OrderDetailResponse getOrderWithDetails(Long orderId) {
        Order order = orderRepository.findById(orderId)
            .orElseThrow(() -> new ResourceNotFoundException("Order not found"));

        UserResponse user = userClient.getUserById(order.getUserId()).getBody();
        ProductResponse product =
productClient.getProductById(order.getProductId()).getBody();

        return new OrderDetailResponse(order, user, product);
    }
}
```

☑ VERIFY

- ☐ Order Service can fetch User details via Feign
- ☐ Order Service can fetch Product details via Feign
- ☐ Load balancing works with multiple instances

? INTERVIEW PREP

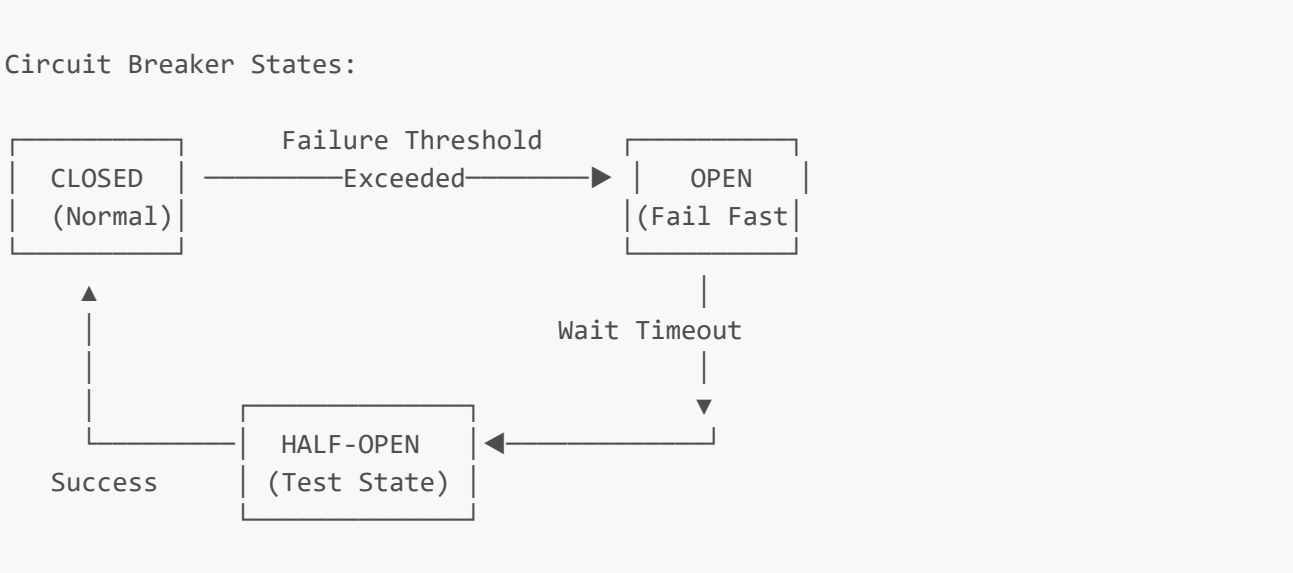
1. "What is OpenFeign? How does it work with Eureka?"
2. "What are the pros and cons of synchronous communication?"
3. "How do you handle timeouts in Feign?"
4. "What happens if the called service is down?"

STORY 3.2: Circuit Breaker Pattern (Resilience4j)

LEARN

Problem: If a downstream service is slow or down, requests pile up and can cascade failures.

Solution: Circuit Breaker - Fails fast and provides fallback



Resilience4j Features:

- Circuit Breaker - Prevent cascade failures
- Retry - Automatic retry with backoff
- Rate Limiter - Limit request rate
- Bulkhead - Limit concurrent calls
- Time Limiter - Timeout handling

💻 IMPLEMENT

Task 3.2.1: Add Resilience4j Dependencies

```
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-circuitbreaker-resilience4j</artifactId>
</dependency>
```

Task 3.2.2: Configure Circuit Breaker

```
# application.yml
resilience4j:
  circuitbreaker:
    instances:
      userService:
        slidingWindowSize: 10
        failureRateThreshold: 50
        waitDurationInOpenState: 10000
        permittedNumberOfCallsInHalfOpenState: 3

  retry:
    instances:
      userService:
        maxAttempts: 3
        waitDuration: 500

  timelimiter:
    instances:
      userService:
        timeoutDuration: 3s
```

Task 3.2.3: Apply Circuit Breaker to Feign Client

```
@FeignClient(name = "user-service", fallback = UserServiceFallback.class)
public interface UserServiceClient {
    @GetMapping("/api/users/{id}")
    ResponseEntity<UserResponse> getUserById(@PathVariable("id") Long id);
}
```

```
@Component
public class UserServiceFallback implements UserServiceClient {

    @Override
    public ResponseEntity<UserResponse> getUserById(Long id) {
        // Return cached or default response
        UserResponse fallback = new UserResponse();
        fallback.setName("Unknown User");
        return ResponseEntity.ok(fallback);
    }
}
```

Task 3.2.4: Add Manual Circuit Breaker

```
@Service
public class OrderService {

    private final CircuitBreaker circuitBreaker;

    public OrderService(CircuitBreakerRegistry registry) {
        this.circuitBreaker = registry.circuitBreaker("userService");
    }

    public UserResponse getUserWithCircuitBreaker(Long userId) {
        return circuitBreaker.executeSupplier(() ->
            userClient.getUserById(userId).getBody()
        );
    }
}
```

☑ VERIFY

- ☐ Service returns fallback when downstream is down
- ☐ Circuit opens after threshold failures
- ☐ Circuit closes after successful calls

? INTERVIEW PREP

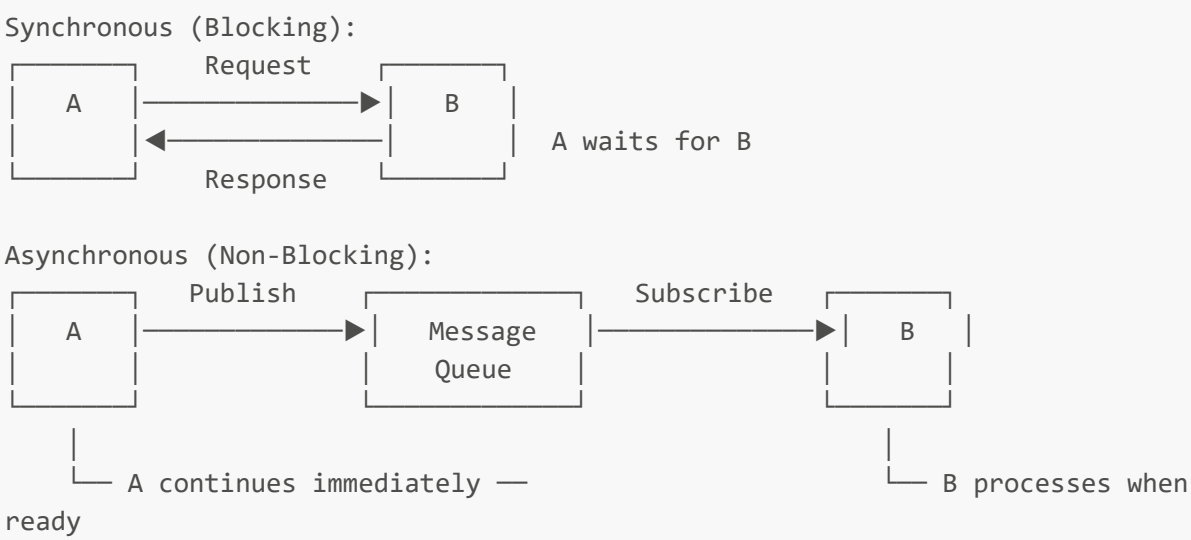
1. "What is the Circuit Breaker pattern?"
2. "Explain the three states of a Circuit Breaker."
3. "What is the difference between Retry and Circuit Breaker?"
4. "How do you configure Resilience4j in Spring Boot?"

STORY 3.3: Asynchronous Communication (Message Queues)

LEARN

Problem: Synchronous calls create tight coupling and blocking operations.

Solution: Asynchronous messaging for loose coupling



Message Queue Options:

- **RabbitMQ** - AMQP protocol, feature-rich
- **Apache Kafka** - High throughput, event streaming
- **Amazon SQS** - AWS managed service
- **Redis Pub/Sub** - Simple, fast

💻 IMPLEMENT

Task 3.3.1: Add RabbitMQ Dependency

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

Task 3.3.2: Configure RabbitMQ

```
@Configuration
public class RabbitMQConfig {

    public static final String ORDER_QUEUE = "order-queue";
    public static final String ORDER_EXCHANGE = "order-exchange";
    public static final String ORDER_ROUTING_KEY = "order.created";

    @Bean
    public Queue orderQueue() {
        return new Queue(ORDER_QUEUE, true);
    }

    @Bean
```

```
    public DirectExchange orderExchange() {
        return new DirectExchange(ORDER_EXCHANGE);
    }

    @Bean
    public Binding binding(Queue queue, DirectExchange exchange) {
        return BindingBuilder.bind(queue).to(exchange).with(ORDER_ROUTING_KEY);
    }
}
```

Task 3.3.3: Create Event Publisher (Order Service)

```
@Service
public class OrderEventPublisher {

    @Autowired
    private RabbitTemplate rabbitTemplate;

    public void publishOrderCreated(Order order) {
        OrderCreatedEvent event = new OrderCreatedEvent(
            order.getId(),
            order.getUserId(),
            order.getProductId(),
            order.getStatus()
        );

        rabbitTemplate.convertAndSend(
            RabbitMQConfig.ORDER_EXCHANGE,
            RabbitMQConfig.ORDER_ROUTING_KEY,
            event
        );
    }
}
```

Task 3.3.4: Create Event Consumer (Notification Service - New)

```
@Service
public class NotificationService {

    @RabbitListener(queues = RabbitMQConfig.ORDER_QUEUE)
    public void handleOrderCreated(OrderCreatedEvent event) {
        // Send email notification
        // Send push notification
        System.out.println("Order created: " + event.getOrderId());
    }
}
```

☑ VERIFY

- ☐ RabbitMQ is running
- ☐ Order creation publishes event
- ☐ Consumer receives and processes event

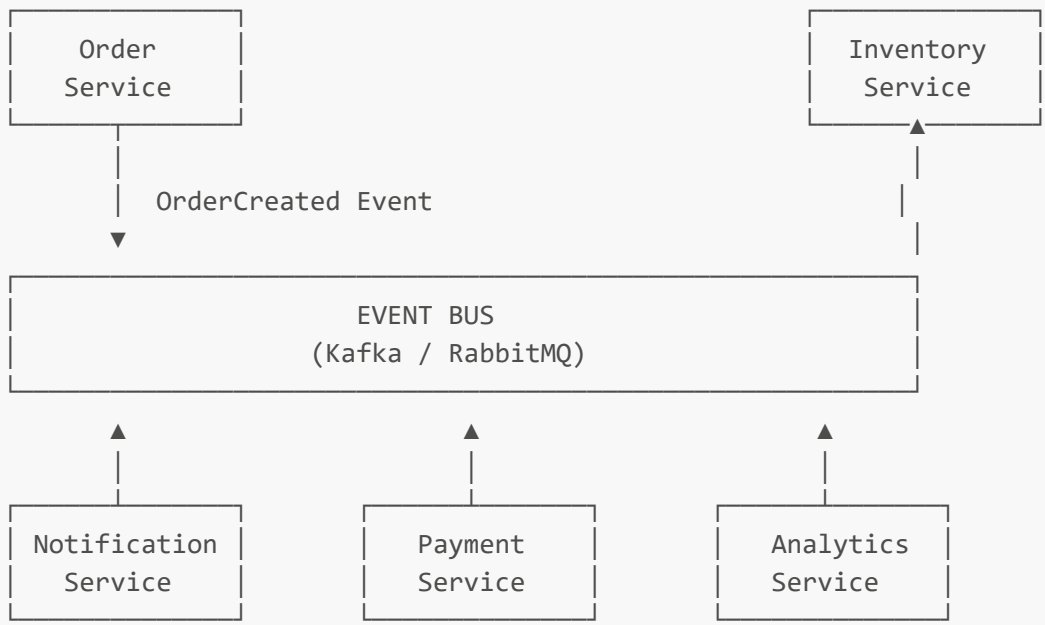
? INTERVIEW PREP

1. "When would you use async vs sync communication?"
2. "What is the difference between RabbitMQ and Kafka?"
3. "How do you ensure message delivery?"
4. "What is the publish-subscribe pattern?"

STORY 3.4: Event-Driven Architecture & Event Sourcing

LEARN

Event-Driven Architecture: Services communicate by producing and consuming events.



Event Sourcing: Store all changes as a sequence of events, not current state.

```
Traditional (Current State):
┌──────────────────────────────┐
│ Order: id=1, status=SHIPPED  │
└──────────────────────────────┘

Event Sourcing (Event Log):
┌──────────────────────────────────────────────────┐
│ 1. OrderCreated   { id: 1, user: 5, product: 10 }    │
│ 2. PaymentReceived { orderId: 1, amount: 50.00 }     │
│ 3. OrderPrepared  { orderId: 1, preparedBy: "chef1"} │
│ 4. OrderShipped   { orderId: 1, trackingNo: "ABC" }  │
```

```
└──────────────────────────────────────────────────────────┘
     ▼ Replay events to get current state
```

💻 IMPLEMENT

Task 3.4.1: Design Domain Events

```
// Base Event
public abstract class DomainEvent {
    private String eventId;
    private LocalDateTime timestamp;
    private String aggregateId;
    private String aggregateType;
}

// Order Events
public class OrderCreatedEvent extends DomainEvent {
    private Long userId;
    private Long productId;
    private BigDecimal amount;
}

public class OrderPaidEvent extends DomainEvent {
    private String transactionId;
}

public class OrderDeliveredEvent extends DomainEvent {
    private LocalDateTime deliveredAt;
}
```

Task 3.4.2: Create Event Store

```
@Entity
@Table(name = "event_store")
public class StoredEvent {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    private String aggregateId;
    private String aggregateType;
    private String eventType;

    @Lob
    private String eventData; // JSON

    private LocalDateTime timestamp;
}
```

```
@Repository
public interface EventStoreRepository extends JpaRepository<StoredEvent, Long> {
    List<StoredEvent> findByAggregateIdOrderByTimestamp(String aggregateId);
}
```

Task 3.4.3: Implement Event Publishing

```
@Service
public class DomainEventPublisher {

    @Autowired
    private EventStoreRepository eventStore;

    @Autowired
    private RabbitTemplate rabbitTemplate;

    @Autowired
    private ObjectMapper objectMapper;

    public void publish(DomainEvent event) throws JsonProcessingException {
        // 1. Store event
        StoredEvent stored = new StoredEvent();
        stored.setAggregateId(event.getAggregateId());
        stored.setEventType(event.getClass().getSimpleName());
        stored.setEventData(objectMapper.writeValueAsString(event));
        stored.setTimestamp(event.getTimestamp());
        eventStore.save(stored);

        // 2. Publish to message queue
        rabbitTemplate.convertAndSend("events", event);
    }
}
```

☑ VERIFY

- ☐ Events are stored in event store
- ☐ Events are published to message queue
- ☐ Can replay events to rebuild state

? INTERVIEW PREP

1. "What is Event-Driven Architecture?"
2. "What is Event Sourcing? What are its benefits?"
3. "What is CQRS and how does it relate to Event Sourcing?"
4. "How do you handle event versioning?"

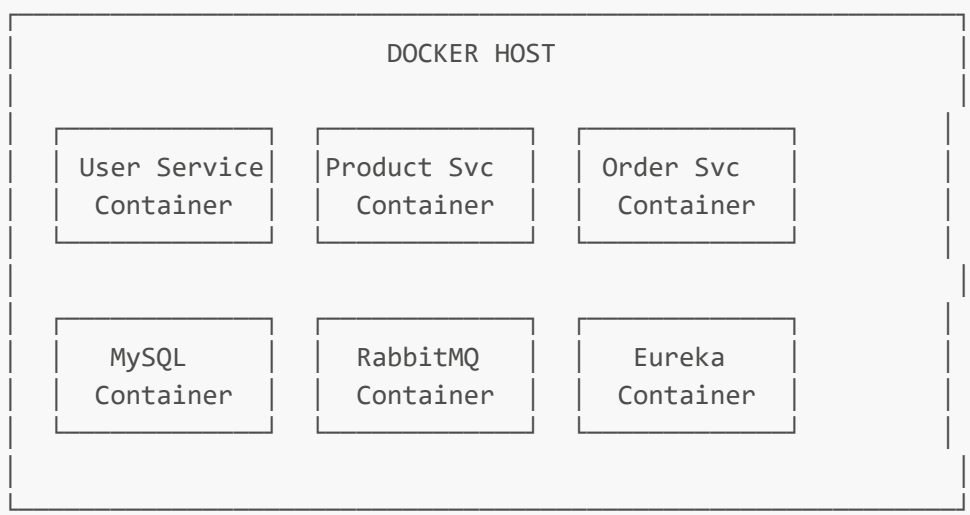
🏭 PHASE 4: INFRASTRUCTURE & DEVOPS {#phase-4-infrastructure}

STORY 4.1: Containerization with Docker

📚 LEARN

Docker Concepts:

- **Image**: Blueprint for a container (like a class)
- **Container**: Running instance of an image (like an object)
- **Dockerfile**: Instructions to build an image
- **Docker Compose**: Define multi-container applications



💻 IMPLEMENT

Task 4.1.1: Create Dockerfile for Each Service

```
# Dockerfile for user-service
FROM openjdk:8-jdk-alpine
VOLUME /tmp
ARG JAR_FILE=target/*.jar
COPY ${JAR_FILE} app.jar
ENTRYPOINT ["java","-jar","/app.jar"]
EXPOSE 8081
```

Task 4.1.2: Create docker-compose.yml

```yaml
version: '3.8'

services:
  mysql:
    image: mysql:8.0
    environment:
      MYSQL_ROOT_PASSWORD: root
      MYSQL_DATABASE: food_delivery
    ports:
      - "3306:3306"
    volumes:
      - mysql_data:/var/lib/mysql
    networks:
      - food-network

  rabbitmq:
    image: rabbitmq:3-management
    ports:
      - "5672:5672"
      - "15672:15672"
    networks:
      - food-network

  discovery-server:
    build: ./discovery-server
    ports:
      - "8761:8761"
    networks:
      - food-network

  api-gateway:
    build: ./api-gateway
    ports:
      - "8080:8080"
    depends_on:
      - discovery-server
    environment:
      - EUREKA_CLIENT_SERVICEURL_DEFAULTZONE=http://discovery-server:8761/eureka/
    networks:
      - food-network

  user-service:
    build: ./user-service/user-service
    ports:
      - "8081:8081"
    depends_on:
      - mysql
      - discovery-server
    environment:
      - SPRING_DATASOURCE_URL=jdbc:mysql://mysql:3306/food_users
      - EUREKA_CLIENT_SERVICEURL_DEFAULTZONE=http://discovery-server:8761/eureka/
    networks:
      - food-network
```

```
  product-service:
    build: ./product-service/product-service
    ports:
      - "8082:8082"
    depends_on:
      - mysql
      - discovery-server
    environment:
      - SPRING_DATASOURCE_URL=jdbc:mysql://mysql:3306/food_products
      - EUREKA_CLIENT_SERVICEURL_DEFAULTZONE=http://discovery-server:8761/eureka/
    networks:
      - food-network

  order-service:
    build: ./order-service/order-service
    ports:
      - "8083:8083"
    depends_on:
      - mysql
      - discovery-server
      - rabbitmq
    environment:
      - SPRING_DATASOURCE_URL=jdbc:mysql://mysql:3306/food_orders
      - EUREKA_CLIENT_SERVICEURL_DEFAULTZONE=http://discovery-server:8761/eureka/
      - SPRING_RABBITMQ_HOST=rabbitmq
    networks:
      - food-network

networks:
  food-network:
    driver: bridge

volumes:
  mysql_data:
```

Task 4.1.3: Build and Run

```
# Build all services
mvn clean package -DskipTests

# Build and start all containers
docker-compose up --build

# Stop all containers
docker-compose down
```

☑ VERIFY

- ☐ All services run in containers

- ☐ Services can communicate within Docker network
- ☐ Data persists with volumes

❓ INTERVIEW PREP

1. "What is the difference between Docker image and container?"
2. "How do containers communicate in Docker Compose?"
3. "What are Docker volumes used for?"
4. "Explain multi-stage Docker builds."

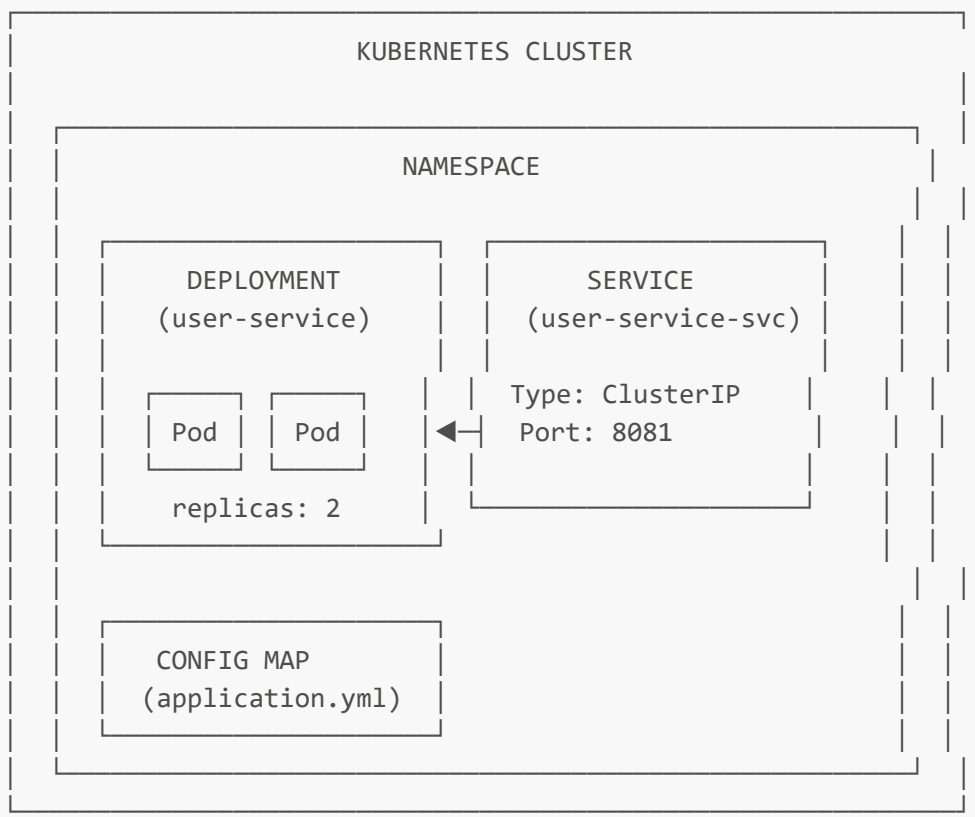
STORY 4.2: Container Orchestration with Kubernetes

📚 LEARN

Why Kubernetes?

- Automatic scaling
- Self-healing (restarts failed containers)
- Load balancing
- Rolling updates
- Service discovery

Kubernetes Objects:



💻 IMPLEMENT

Task 4.2.1: Create Kubernetes Manifests

```yaml
# k8s/user-service-deployment.yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: user-service
  labels:
    app: user-service
spec:
  replicas: 2
  selector:
    matchLabels:
      app: user-service
  template:
    metadata:
      labels:
        app: user-service
    spec:
      containers:
        - name: user-service
          image: food-delivery/user-service:latest
          ports:
            - containerPort: 8081
          env:
            - name: SPRING_DATASOURCE_URL
              valueFrom:
                configMapKeyRef:
                  name: user-service-config
                  key: database-url
          resources:
            requests:
              memory: "256Mi"
              cpu: "250m"
            limits:
              memory: "512Mi"
              cpu: "500m"
          livenessProbe:
            httpGet:
              path: /actuator/health
              port: 8081
            initialDelaySeconds: 60
            periodSeconds: 10
          readinessProbe:
            httpGet:
              path: /actuator/health
              port: 8081
            initialDelaySeconds: 30
            periodSeconds: 5
---
apiVersion: v1
kind: Service
metadata:
```

```
  name: user-service
spec:
  selector:
    app: user-service
  ports:
    - protocol: TCP
      port: 8081
      targetPort: 8081
  type: ClusterIP
```

Task 4.2.2: Create ConfigMap and Secrets

```
# k8s/configmap.yaml
apiVersion: v1
kind: ConfigMap
metadata:
  name: user-service-config
data:
  database-url: jdbc:mysql://mysql-service:3306/food_users
  eureka-url: http://discovery-server:8761/eureka/
---
apiVersion: v1
kind: Secret
metadata:
  name: db-credentials
type: Opaque
data:
  username: cm9vdA==  # base64 encoded
  password: cm9vdA==
```

Task 4.2.3: Deploy to Kubernetes

```
# Apply all manifests
kubectl apply -f k8s/

# Check deployments
kubectl get deployments

# Check pods
kubectl get pods

# Check services
kubectl get services

# View logs
kubectl logs -f deployment/user-service
```

☑ VERIFY

- [ ] All pods are running
- [ ] Services are accessible within cluster
- [ ] Scaling works (increase replicas)

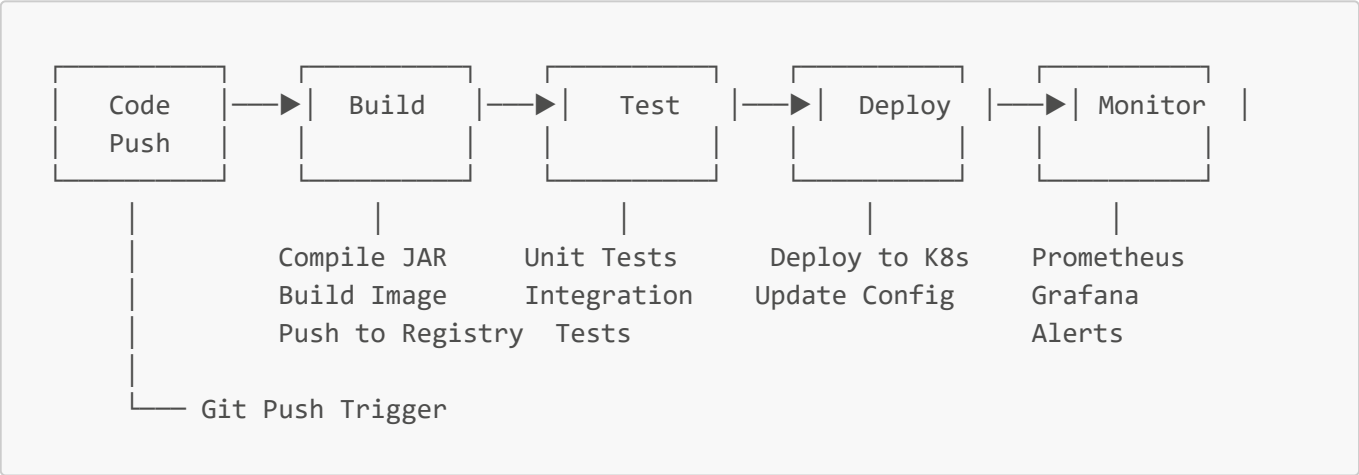
❓ INTERVIEW PREP

1. "What is Kubernetes? Why do we need it?"
2. "Explain Pod, Deployment, Service, and Ingress."
3. "What is the difference between ClusterIP, NodePort, and LoadBalancer?"
4. "How does Kubernetes handle self-healing?"

STORY 4.3: CI/CD Pipeline

📚 LEARN

CI/CD Flow:



💻 IMPLEMENT

Task 4.3.1: Create GitHub Actions Workflow

```yaml
# .github/workflows/ci-cd.yml
name: CI/CD Pipeline

on:
  push:
    branches: [ main ]
  pull_request:
    branches: [ main ]

jobs:
  build:
    runs-on: ubuntu-latest

    steps:
    - uses: actions/checkout@v3
```

```
    - name: Set up JDK 8
      uses: actions/setup-java@v3
      with:
        java-version: '8'
        distribution: 'temurin'

    - name: Build User Service
      run: |
        cd user-service/user-service
        mvn clean package -DskipTests

    - name: Build Product Service
      run: |
        cd product-service/product-service
        mvn clean package -DskipTests

    - name: Build Order Service
      run: |
        cd order-service/order-service
        mvn clean package -DskipTests

    - name: Run Tests
      run: |
        cd user-service/user-service
        mvn test
        cd ../../product-service/product-service
        mvn test
        cd ../../order-service/order-service
        mvn test

  docker:
    needs: build
    runs-on: ubuntu-latest
    if: github.ref == 'refs/heads/main'

    steps:
    - uses: actions/checkout@v3

    - name: Login to Docker Hub
      uses: docker/login-action@v2
      with:
        username: ${{ secrets.DOCKER_USERNAME }}
        password: ${{ secrets.DOCKER_PASSWORD }}

    - name: Build and Push Docker Images
      run: |
        docker build -t youruser/user-service:${{ github.sha }} ./user-
service/user-service
        docker push youruser/user-service:${{ github.sha }}

        docker build -t youruser/product-service:${{ github.sha }} ./product-
service/product-service
        docker push youruser/product-service:${{ github.sha }}
```

```
        docker build -t youruser/order-service:${{ github.sha }} ./order-
service/order-service
        docker push youruser/order-service:${{ github.sha }}

  deploy:
    needs: docker
    runs-on: ubuntu-latest
    if: github.ref == 'refs/heads/main'

    steps:
    - name: Deploy to Kubernetes
      run: |
        kubectl set image deployment/user-service user-service=youruser/user-
service:${{ github.sha }}
        kubectl set image deployment/product-service product-
service=youruser/product-service:${{ github.sha }}
        kubectl set image deployment/order-service order-service=youruser/order-
service:${{ github.sha }}
```

☑ VERIFY

- ☐ Pipeline triggers on push
- ☐ Tests run automatically
- ☐ Docker images are built and pushed
- ☐ Kubernetes deployments update

? INTERVIEW PREP

1. "What is CI/CD? Why is it important?"
2. "Explain the stages of your CI/CD pipeline."
3. "How do you handle rollbacks in deployment?"
4. "What is blue-green deployment? Canary deployment?"

STORY 4.4: Observability (Logging, Monitoring, Tracing)

LEARN

Three Pillars of Observability:

```
┌─────────────────────────────────────────────────────────────────────┐
│                          OBSERVABILITY                              │
├──────────────────────┬──────────────────────┬───────────────────────┤
│      LOGGING         │      MONITORING      │       TRACING         │
│                      │                      │                       │
│  • Application logs  │  • Metrics           │  • Request flow       │
│  • Error tracking    │  • CPU, Memory       │  • Cross-service calls│
│  • Audit logs        │  • Response times    │  • Latency breakdown  │
│                      │  • Error rates       │                       │
│  Tools:              │  Tools:              │  Tools:               │
```

```
│  • ELK Stack          │  • Prometheus          │  • Jaeger                 │
│  • Loki               │  • Grafana             │  • Zipkin                 │
│                       │  • Micrometer          │  • Spring Cloud Sleuth    │
└───────────────────────┴────────────────────────┴───────────────────────────┘
```

💻 IMPLEMENT

Task 4.4.1: Add Spring Boot Actuator

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

```properties
# application.properties
management.endpoints.web.exposure.include=health,info,metrics,prometheus
management.endpoint.health.show-details=always
```

Task 4.4.2: Add Distributed Tracing with Sleuth

```xml
<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>
```

```properties
spring.sleuth.sampler.probability=1.0
spring.zipkin.base-url=http://zipkin:9411
```

Task 4.4.3: Configure Structured Logging

```xml
<!-- logback-spring.xml -->
<configuration>
    <appender name="JSON" class="ch.qos.logback.core.ConsoleAppender">
        <encoder class="net.logstash.logback.encoder.LogstashEncoder">
            <includeMdcKeyName>traceId</includeMdcKeyName>
```

```
            <includeMdcKeyName>spanId</includeMdcKeyName>
        </encoder>
    </appender>

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

Task 4.4.4: Create Custom Metrics

```
@Component
public class OrderMetrics {

    private final Counter orderCounter;
    private final Timer orderTimer;

    public OrderMetrics(MeterRegistry registry) {
        this.orderCounter = Counter.builder("orders.created")
            .description("Number of orders created")
            .register(registry);

        this.orderTimer = Timer.builder("order.creation.time")
            .description("Time to create order")
            .register(registry);
    }

    public void recordOrderCreated() {
        orderCounter.increment();
    }

    public void recordOrderCreationTime(Duration duration) {
        orderTimer.record(duration);
    }
}
```

☑ VERIFY

- ☐ Actuator endpoints accessible
- ☐ Prometheus scrapes metrics
- ☐ Traces visible in Zipkin/Jaeger
- ☐ Logs are structured JSON

? INTERVIEW PREP

1. "What are the three pillars of observability?"
2. "How do you implement distributed tracing?"
3. "What metrics do you monitor in microservices?"
4. "How do you correlate logs across services?"

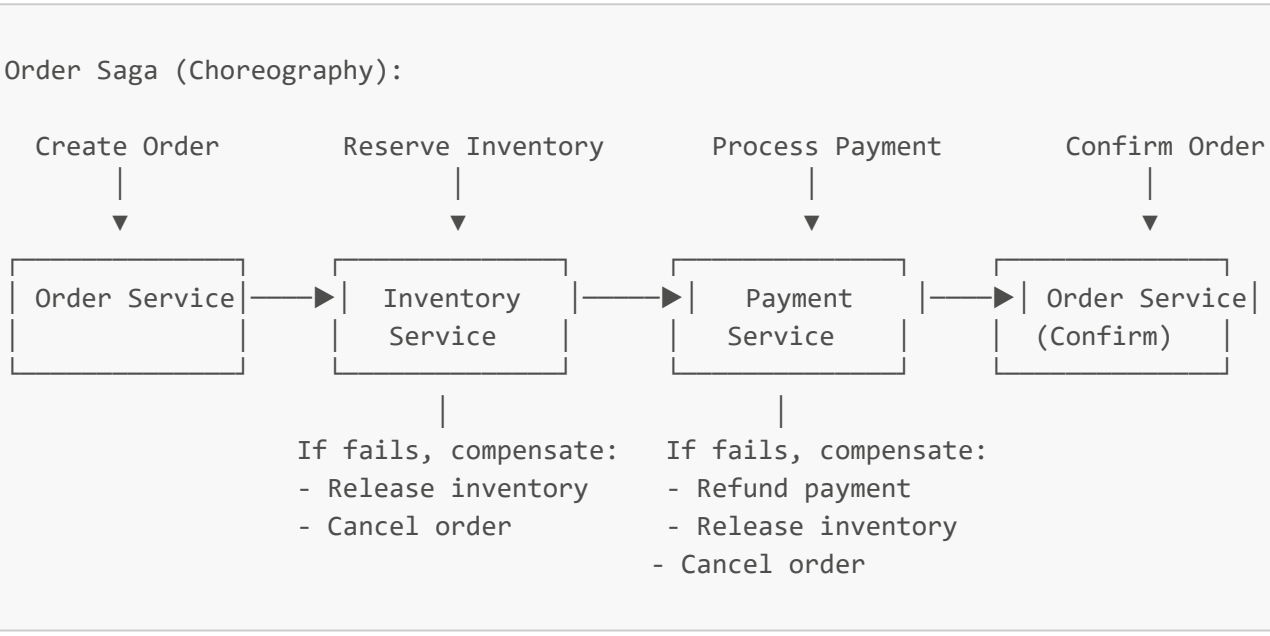
🚀 PHASE 5: ADVANCED PATTERNS {#phase-5-advanced-patterns}

STORY 5.1: Saga Pattern (Distributed Transactions)

📚 LEARN

Problem: How to maintain data consistency across multiple services without distributed transactions?

Saga Pattern: A sequence of local transactions where each step has a compensating transaction.



Two Approaches:

1. **Choreography**: Each service publishes events and listens to others
2. **Orchestration**: Central coordinator manages the saga

💻 IMPLEMENT

Task 5.1.1: Design Order Saga Events

```
// Events
public class OrderCreatedEvent { Long orderId; Long userId; Long productId;
BigDecimal amount; }
public class InventoryReservedEvent { Long orderId; Long productId; }
public class InventoryReservationFailedEvent { Long orderId; String reason; }
public class PaymentProcessedEvent { Long orderId; String transactionId; }
public class PaymentFailedEvent { Long orderId; String reason; }
public class OrderConfirmedEvent { Long orderId; }
public class OrderCancelledEvent { Long orderId; String reason; }
```

Task 5.1.2: Implement Saga Orchestrator

```
@Service
public class OrderSagaOrchestrator {

    @Autowired
    private OrderService orderService;

    @Autowired
    private InventoryServiceClient inventoryClient;

    @Autowired
    private PaymentServiceClient paymentClient;

    @Transactional
    public OrderResult processOrder(CreateOrderRequest request) {
        // Step 1: Create Order
        Order order = orderService.createOrder(request);

        try {
            // Step 2: Reserve Inventory
            inventoryClient.reserveInventory(order.getProductId(),
order.getQuantity());

            try {
                // Step 3: Process Payment
                paymentClient.processPayment(order.getUserId(),
order.getAmount());

                // Step 4: Confirm Order
                orderService.confirmOrder(order.getId());
                return OrderResult.success(order);

            } catch (PaymentException e) {
                // Compensate: Release Inventory
                inventoryClient.releaseInventory(order.getProductId(),
order.getQuantity());
                orderService.cancelOrder(order.getId(), "Payment failed");
                return OrderResult.failure("Payment failed: " + e.getMessage());
            }

        } catch (InventoryException e) {
            // Compensate: Cancel Order
            orderService.cancelOrder(order.getId(), "Insufficient inventory");
            return OrderResult.failure("Inventory unavailable: " +
e.getMessage());
        }
    }
}
```

☑ VERIFY

- ☐ Order saga completes successfully
- ☐ Compensation runs on failure
- ☐ Data remains consistent

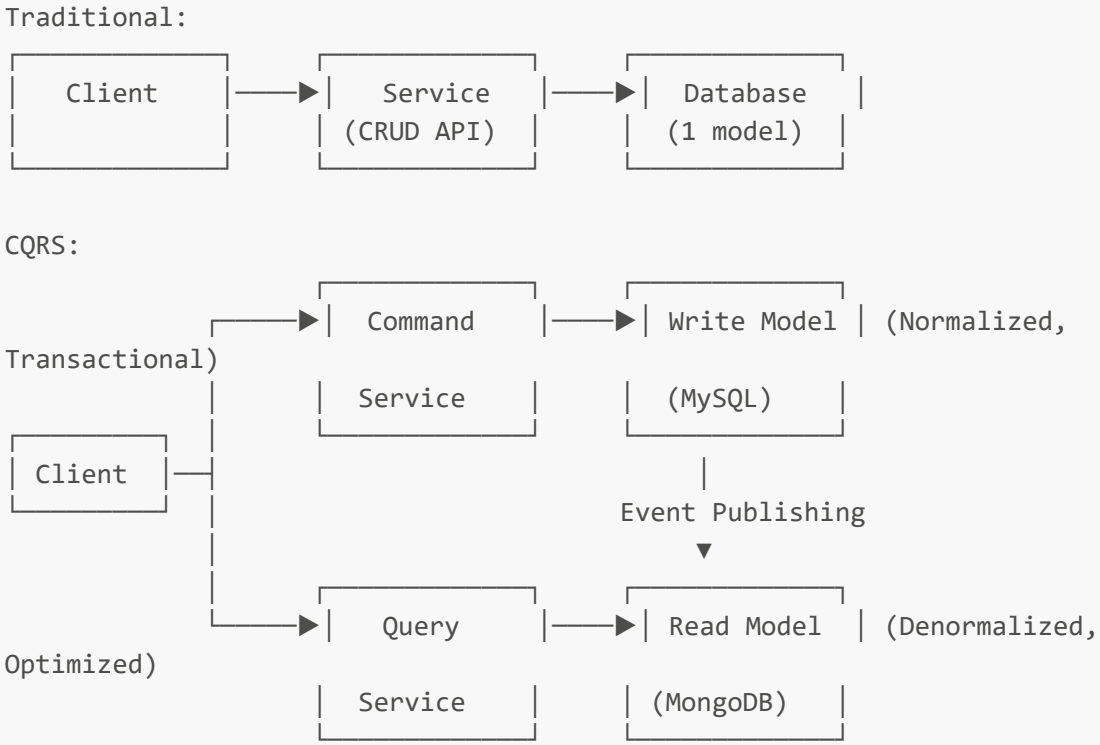
❓ INTERVIEW PREP

1. "What is the Saga pattern?"
2. "Explain choreography vs orchestration."
3. "How do you handle saga failures?"
4. "What are the challenges of the Saga pattern?"

STORY 5.2: CQRS Pattern

📚 LEARN

CQRS (Command Query Responsibility Segregation): Separate read and write models.



Benefits:

- Optimize read and write independently
- Scale read-heavy operations
- Different data stores for different needs

💻 IMPLEMENT

Task 5.2.1: Create Command Side (Order Write Service)

```
@RestController
@RequestMapping("/orders/commands")
public class OrderCommandController {

    @Autowired
    private OrderCommandService commandService;

    @PostMapping
    public ResponseEntity<OrderId> createOrder(@RequestBody CreateOrderCommand
command) {
        Long orderId = commandService.createOrder(command);
        return ResponseEntity.status(HttpStatus.CREATED).body(new
OrderId(orderId));
    }

    @PutMapping("/{id}/status")
    public ResponseEntity<Void> updateStatus(
            @PathVariable Long id,
            @RequestBody UpdateStatusCommand command) {
        commandService.updateStatus(id, command);
        return ResponseEntity.ok().build();
    }
}
```

Task 5.2.2: Create Query Side (Order Read Service)

```
@RestController
@RequestMapping("/orders/queries")
public class OrderQueryController {

    @Autowired
    private OrderQueryService queryService;

    @GetMapping("/{id}")
    public ResponseEntity<OrderView> getOrder(@PathVariable Long id) {
        return ResponseEntity.ok(queryService.getOrderById(id));
    }

    @GetMapping("/user/{userId}")
    public ResponseEntity<List<OrderSummary>> getUserOrders(@PathVariable Long
userId) {
        return ResponseEntity.ok(queryService.getOrdersByUser(userId));
    }

    @GetMapping("/dashboard")
    public ResponseEntity<OrderDashboard> getDashboard() {
        return ResponseEntity.ok(queryService.getDashboardStats());
    }
}
```

Task 5.2.3: Sync Read Model with Events

```
@Service
public class OrderProjectionService {

    @Autowired
    private OrderViewRepository viewRepository;

    @RabbitListener(queues = "order-events")
    public void handleOrderEvent(OrderEvent event) {
        if (event instanceof OrderCreatedEvent) {
            createProjection((OrderCreatedEvent) event);
        } else if (event instanceof OrderStatusUpdatedEvent) {
            updateProjection((OrderStatusUpdatedEvent) event);
        }
    }

    private void createProjection(OrderCreatedEvent event) {
        OrderView view = new OrderView();
        view.setOrderId(event.getOrderId());
        view.setUserName(fetchUserName(event.getUserId())); // Denormalized
        view.setProductName(fetchProductName(event.getProductId())); //
Denormalized
        view.setStatus(event.getStatus());
        viewRepository.save(view);
    }
}
```

☑ VERIFY

- ☐ Write operations go to command service
- ☐ Read operations go to query service
- ☐ Read model is eventually consistent

? INTERVIEW PREP

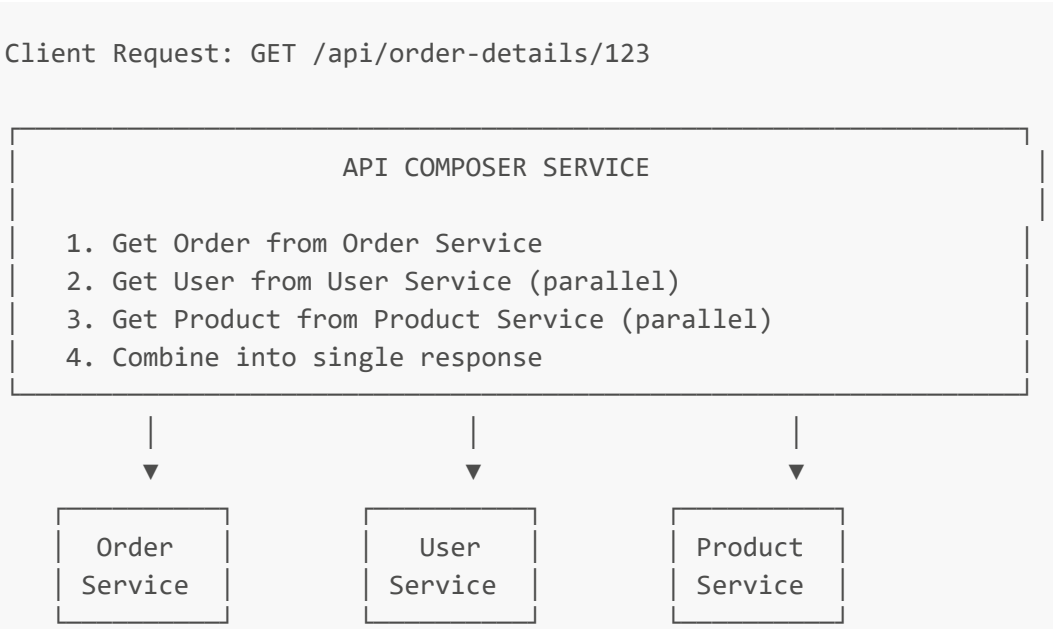
1. "What is CQRS?"
2. "When should you use CQRS?"
3. "What is eventual consistency?"
4. "How do you keep read and write models in sync?"

STORY 5.3: API Composition Pattern

LEARN

Problem: How to query data that spans multiple services?

API Composition: Aggregate data from multiple services into a single response.



💻 IMPLEMENT

Task 5.3.1: Create Order Details Aggregator

```
@Service
public class OrderDetailsAggregator {

    @Autowired
    private OrderServiceClient orderClient;

    @Autowired
    private UserServiceClient userClient;

    @Autowired
    private ProductServiceClient productClient;

    public OrderDetailsResponse getOrderDetails(Long orderId) {
        // Get order
        Order order = orderClient.getOrder(orderId).getBody();

        // Parallel calls for user and product
        CompletableFuture<UserResponse> userFuture =
CompletableFuture.supplyAsync(() ->
            userClient.getUser(order.getUserId()).getBody()
        );

        CompletableFuture<ProductResponse> productFuture =
CompletableFuture.supplyAsync(() ->
            productClient.getProduct(order.getProductId()).getBody()
        );

        // Wait for both and combine
        try {
            UserResponse user = userFuture.get(3, TimeUnit.SECONDS);
```

```
            ProductResponse product = productFuture.get(3, TimeUnit.SECONDS);

            return OrderDetailsResponse.builder()
                .orderId(order.getId())
                .status(order.getStatus())
                .user(user)
                .product(product)
                .build();

        } catch (Exception e) {
            throw new ServiceUnavailableException("Failed to aggregate order
details", e);
        }
    }
}
```

Task 5.3.2: Add Caching for Performance

```
@Service
public class OrderDetailsAggregator {

    @Autowired
    private CacheManager cacheManager;

    @Cacheable(value = "orderDetails", key = "#orderId", unless = "#result ==
null")
    public OrderDetailsResponse getOrderDetails(Long orderId) {
        // Aggregation logic
    }

    @CacheEvict(value = "orderDetails", key = "#orderId")
    public void evictOrderDetailsCache(Long orderId) {
        // Called when order is updated
    }
}
```

☑ VERIFY

- ☐ Aggregated response contains data from all services
- ☐ Parallel calls reduce latency
- ☐ Caching improves performance

? INTERVIEW PREP

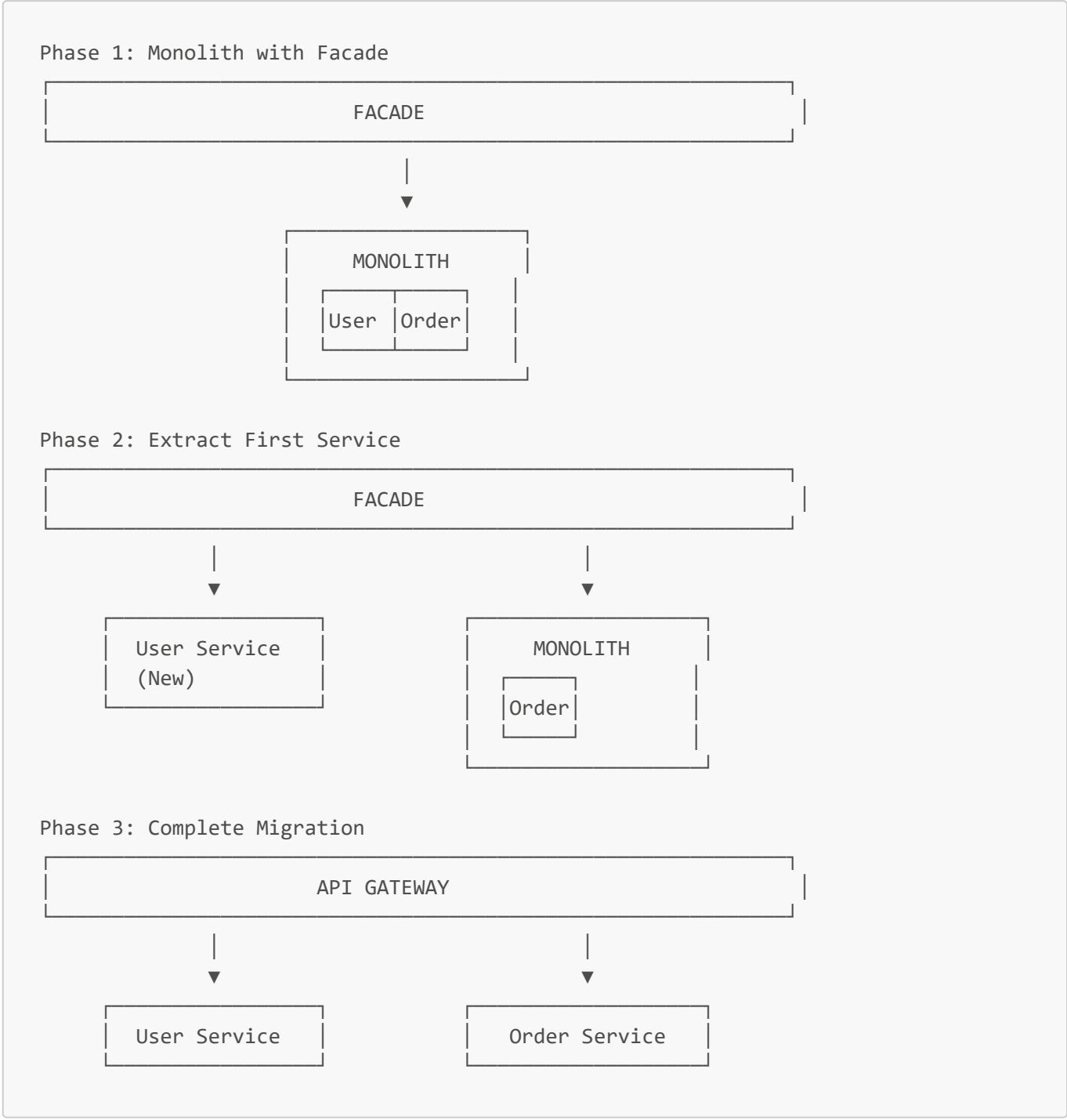
1. "What is the API Composition pattern?"
2. "How do you handle failures in API composition?"
3. "How do you optimize API composition performance?"

STORY 5.4: Strangler Fig Pattern (Microservices Migration)

LEARN

Problem: How to gradually migrate from monolith to microservices?

Strangler Fig Pattern: Incrementally replace monolith components with microservices.



IMPLEMENT

Task 5.4.1: Create Migration Facade

```
@Configuration
public class MigrationRoutingConfig {

    @Bean
```

```
    public RouteLocator migrationRoutes(RouteLocatorBuilder builder) {
        return builder.routes()
            // New microservice handles users
            .route("user-route", r -> r
                .path("/api/users/**")
                .uri("lb://user-service"))

            // Monolith still handles products (temporary)
            .route("product-route", r -> r
                .path("/api/products/**")
                .uri("http://legacy-monolith:8080"))

            // Feature flag: Gradually shift order traffic
            .route("order-route", r -> r
                .path("/api/orders/**")
                .filters(f -> f.filter(new TrafficSplitter())) // 10% to new
service
                .uri("lb://order-service"))

            .build();
    }
}
```

Task 5.4.2: Implement Data Sync During Migration

```
@Service
public class DataSyncService {

    // During migration, keep both systems in sync
    @Async
    public void syncOrderToLegacy(Order order) {
        // Write to legacy system for backward compatibility
    }

    @Async
    public void syncOrderFromLegacy(LegacyOrder legacyOrder) {
        // Import from legacy to new system
    }
}
```

☑ VERIFY

- ☐ New services coexist with monolith
- ☐ Traffic gradually shifts to microservices
- ☐ Data remains consistent during migration

? INTERVIEW PREP

1. "What is the Strangler Fig pattern?"
2. "How do you handle data migration during the strangling process?"

3. "What are feature flags and how do they help in migration?"

🔒 PHASE 6: PRODUCTION READINESS {#phase-6-production-readiness}

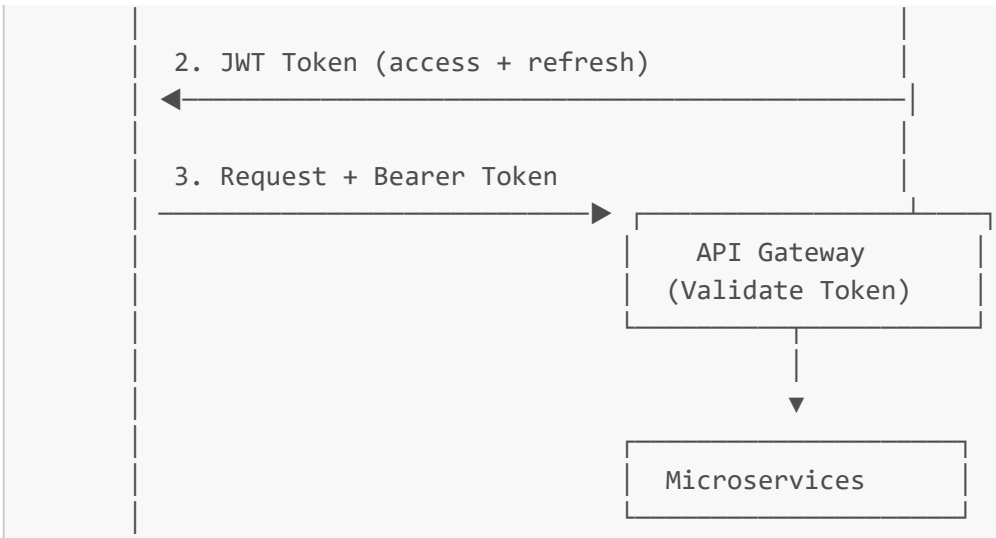
STORY 6.1: Security in Microservices

📚 LEARN

Security Patterns:

```
┌─────────────────────────────────────────────────────────────────────┐
│                         SECURITY LAYERS                             │
├─────────────────────────────────────────────────────────────────────┤
│                                                                     │
│  1. PERIMETER SECURITY                                              │
│     ├── API Gateway Authentication                                  │
│     ├── Rate Limiting                                               │
│     └── WAF (Web Application Firewall)                              │
│                                                                     │
│  2. SERVICE-TO-SERVICE SECURITY                                     │
│     ├── mTLS (Mutual TLS)                                           │
│     ├── Service Mesh (Istio)                                        │
│     └── Internal tokens                                             │
│                                                                     │
│  3. APPLICATION SECURITY                                            │
│     ├── JWT Token Validation                                        │
│     ├── Role-Based Access Control (RBAC)                            │
│     └── Input Validation                                            │
│                                                                     │
│  4. DATA SECURITY                                                   │
│     ├── Encryption at Rest                                          │
│     ├── Encryption in Transit                                       │
│     └── Data Masking                                                │
│                                                                     │
└─────────────────────────────────────────────────────────────────────┘
```

OAuth 2.0 + JWT Flow:



IMPLEMENT

Task 6.1.1: JWT Validation at Gateway

```
@Component
public class JwtAuthenticationFilter implements GlobalFilter, Ordered {

    @Override
    public Mono<Void> filter(ServerWebExchange exchange, GatewayFilterChain chain)
{
        String authHeader =
exchange.getRequest().getHeaders().getFirst("Authorization");

        if (authHeader == null || !authHeader.startsWith("Bearer ")) {
            return unauthorized(exchange);
        }

        String token = authHeader.substring(7);

        try {
            Claims claims = validateToken(token);
            // Add user info to headers for downstream services
            ServerHttpRequest request = exchange.getRequest().mutate()
                .header("X-User-Id", claims.getSubject())
                .header("X-User-Roles", claims.get("roles", String.class))
                .build();
            return chain.filter(exchange.mutate().request(request).build());
        } catch (Exception e) {
            return unauthorized(exchange);
        }
    }

    @Override
    public int getOrder() {
        return -1;
```

```
    }
}
```

Task 6.1.2: Role-Based Access Control

```
@RestController
@RequestMapping("/api/admin")
public class AdminController {

    @PreAuthorize("hasRole('ADMIN')")
    @GetMapping("/users")
    public List<UserResponse> getAllUsers() {
        return userService.findAll();
    }

    @PreAuthorize("hasAnyRole('ADMIN', 'MANAGER')")
    @GetMapping("/orders")
    public List<OrderResponse> getAllOrders() {
        return orderService.findAll();
    }
}
```

Task 6.1.3: Secure Service-to-Service Communication

```
@Configuration
public class FeignSecurityConfig {

    @Bean
    public RequestInterceptor serviceTokenInterceptor() {
        return requestTemplate -> {
            String serviceToken = generateServiceToken();
            requestTemplate.header("X-Service-Token", serviceToken);
        };
    }
}
```

☑ VERIFY

- ☐ JWT validation works at gateway
- ☐ Unauthorized requests are rejected
- ☐ RBAC restricts access properly

? INTERVIEW PREP

1. "How do you secure microservices?"
2. "Explain OAuth 2.0 and JWT."
3. "How do you handle service-to-service authentication?"

4. "What is mTLS?"

STORY 6.2: Rate Limiting & Throttling

LEARN

Rate Limiting Algorithms:

1. **Token Bucket**: Allows bursts up to bucket size
2. **Leaky Bucket**: Constant output rate
3. **Fixed Window**: Reset counter each time window
4. **Sliding Window**: Smooth rate limiting

IMPLEMENT

Task 6.2.1: Add Rate Limiting to Gateway

```
@Bean
public KeyResolver userKeyResolver() {
    return exchange -> Mono.just(
        exchange.getRequest().getHeaders().getFirst("X-User-Id") != null
            ? exchange.getRequest().getHeaders().getFirst("X-User-Id")
            :
exchange.getRequest().getRemoteAddress().getAddress().getHostAddress()
    );
}
```

```
spring:
  cloud:
    gateway:
      routes:
        - id: user-service
          uri: lb://user-service
          predicates:
            - Path=/api/users/**
          filters:
            - name: RequestRateLimiter
              args:
                redis-rate-limiter.replenishRate: 10
                redis-rate-limiter.burstCapacity: 20
                key-resolver: "#{@userKeyResolver}"
```

VERIFY

- [ ] Rate limiting rejects excess requests
- [ ] Different limits for different endpoints
- [ ] 429 status returned when limit exceeded

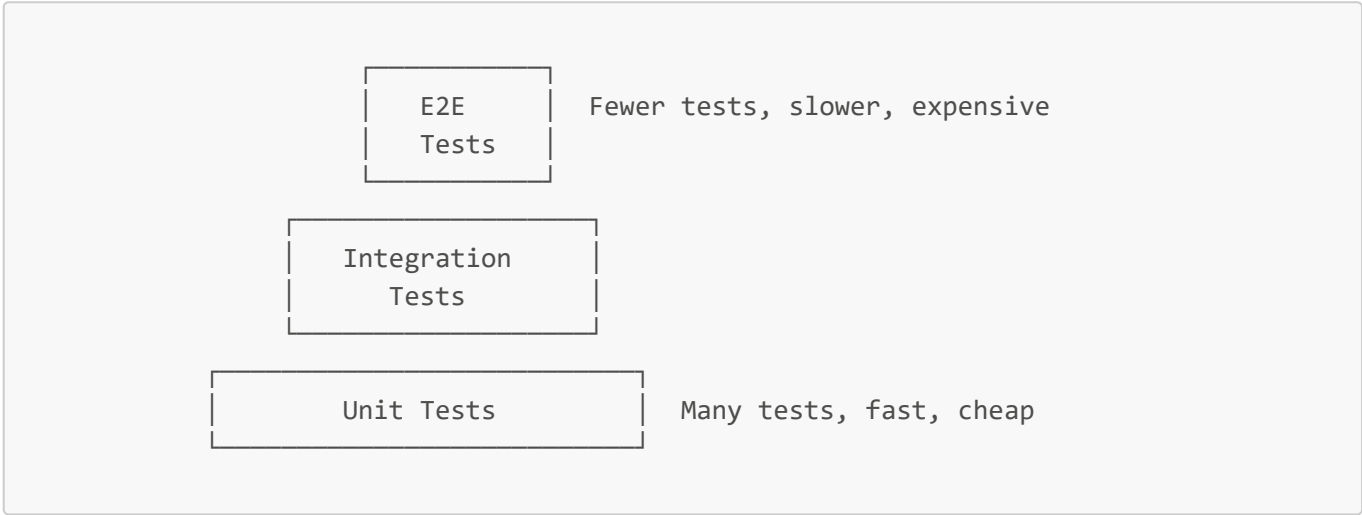
❓ INTERVIEW PREP

1. "What is rate limiting? Why is it important?"
2. "Explain different rate limiting algorithms."
3. "How do you implement rate limiting in microservices?"

STORY 6.3: Testing Strategies

📚 LEARN

Testing Pyramid:



Contract Testing: Verify API contracts between services using Pact or Spring Cloud Contract.

💻 IMPLEMENT

Task 6.3.1: Unit Tests with Mocking

```
@ExtendWith(MockitoExtension.class)
class OrderServiceTest {

    @Mock
    private OrderRepository orderRepository;

    @Mock
    private UserServiceClient userClient;

    @InjectMocks
    private OrderService orderService;

    @Test
    void shouldCreateOrder() {
        // Given
        CreateOrderRequest request = new CreateOrderRequest(1L, 1L);
        Order expectedOrder = new Order(1L, 1L, 1L, "PLACED");

        when(orderRepository.save(any())).thenReturn(expectedOrder);
```

```
        // When
        Order result = orderService.createOrder(request);

        // Then
        assertThat(result.getStatus()).isEqualTo("PLACED");
        verify(orderRepository).save(any());
    }
}
```

Task 6.3.2: Integration Tests with TestContainers

```
@SpringBootTest
@Testcontainers
class OrderServiceIntegrationTest {

    @Container
    static MySQLContainer<?> mysql = new MySQLContainer<>("mysql:8.0");

    @DynamicPropertySource
    static void configureProperties(DynamicPropertyRegistry registry) {
        registry.add("spring.datasource.url", mysql::getJdbcUrl);
        registry.add("spring.datasource.username", mysql::getUsername);
        registry.add("spring.datasource.password", mysql::getPassword);
    }

    @Autowired
    private TestRestTemplate restTemplate;

    @Test
    void shouldCreateAndRetrieveOrder() {
        // Create order
        CreateOrderRequest request = new CreateOrderRequest(1L, 1L);
        ResponseEntity<Order> createResponse = restTemplate.postForEntity(
            "/orders", request, Order.class);

        assertThat(createResponse.getStatusCode()).isEqualTo(HttpStatus.CREATED);

        // Retrieve order
        Long orderId = createResponse.getBody().getId();
        ResponseEntity<Order> getResponse = restTemplate.getForEntity(
            "/orders/" + orderId, Order.class);

        assertThat(getResponse.getBody().getStatus()).isEqualTo("PLACED");
    }
}
```

Task 6.3.3: Contract Tests with Spring Cloud Contract

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

☑ VERIFY

- ☐ Unit tests cover business logic
- ☐ Integration tests verify database operations
- ☐ Contract tests ensure API compatibility

? INTERVIEW PREP

1. "What is the testing pyramid?"
2. "How do you test microservices?"
3. "What is contract testing?"
4. "How do you use TestContainers?"

STORY 6.4: Performance Optimization

LEARN

Optimization Strategies:

1. **Caching** - Redis, in-memory
2. **Connection Pooling** - HikariCP, Jedis
3. **Async Processing** - CompletableFuture, WebFlux
4. **Database Optimization** - Indexes, query optimization
5. **Compression** - GZIP responses

IMPLEMENT

Task 6.4.1: Add Redis Caching

```
@Configuration
@EnableCaching
public class CacheConfig {

    @Bean
    public RedisCacheManager cacheManager(RedisConnectionFactory factory) {
        RedisCacheConfiguration config =
RedisCacheConfiguration.defaultCacheConfig()
            .entryTtl(Duration.ofMinutes(10))
            .serializeValuesWith(
                RedisSerializationContext.SerializationPair.fromSerializer(
                    new GenericJackson2JsonRedisSerializer()
                )
            );

        return RedisCacheManager.builder(factory)
            .cacheDefaults(config)
            .build();
    }
}

@Service
public class ProductService {

    @Cacheable(value = "products", key = "#id")
    public Product getProduct(Long id) {
        return productRepository.findById(id).orElseThrow();
    }

    @CacheEvict(value = "products", key = "#id")
    public Product updateProduct(Long id, ProductRequest request) {
        // Update logic
    }
}
```

Task 6.4.2: Connection Pool Configuration

```
# HikariCP settings
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
spring.datasource.hikari.max-lifetime=1200000
```

Task 6.4.3: Response Compression

```
server.compression.enabled=true
server.compression.mime-types=application/json,text/html,text/plain
server.compression.min-response-size=1024
```

☑ VERIFY

- [ ] Cache hit ratio is high
- [ ] Response times are improved
- [ ] Connection pool is properly sized

? INTERVIEW PREP

1. "How do you optimize microservices performance?"
2. "What caching strategies do you use?"
3. "How do you size connection pools?"
4. "What is N+1 query problem and how to solve it?"

INTERVIEW QUESTIONS BANK {#interview-questions}

Fundamentals

1. What is microservices architecture?
2. Monolith vs Microservices - pros and cons?
3. When should you use microservices?
4. What is the 12-factor app methodology?
5. What is Domain-Driven Design?

Communication

6. Synchronous vs Asynchronous communication?
7. How do services discover each other?
8. What is a message queue? When to use it?
9. REST vs gRPC vs GraphQL?
10. How do you handle inter-service communication failures?

Patterns

11. What is the Circuit Breaker pattern?
12. What is the Saga pattern?
13. What is CQRS?
14. What is Event Sourcing?
15. What is the API Gateway pattern?

Infrastructure

16. What is service discovery?
17. What is an API Gateway?
18. How do you handle configuration in microservices?

19. What is containerization?
20. What is Kubernetes?

Data Management

21. Why database per service?
22. How do you maintain data consistency?
23. What is eventual consistency?
24. How do you handle distributed transactions?
25. What is the Outbox pattern?

Security

26. How do you secure microservices?
27. What is OAuth 2.0?
28. How does JWT work?
29. What is mTLS?
30. How do you handle authorization?

Observability

31. What is distributed tracing?
32. How do you aggregate logs?
33. What metrics do you monitor?
34. What is the ELK stack?
35. How do you debug issues in microservices?

Testing

36. How do you test microservices?
37. What is contract testing?
38. What is the testing pyramid?
39. How do you do integration testing?
40. What is chaos engineering?

DevOps

41. What is CI/CD?
42. What is blue-green deployment?
43. What is canary deployment?
44. How do you handle rollbacks?
45. What is GitOps?

Troubleshooting

46. How do you handle a service being down?
47. How do you identify performance bottlenecks?
48. What is a cascading failure?
49. How do you handle network partitions?

50. How do you handle data inconsistency?

☑ HANDS-ON PROJECTS CHECKLIST {#projects-checklist}

Phase 1: Foundation

- ☐ Create 3 microservices with separate databases
- ☐ Implement REST APIs with proper error handling
- ☐ Add DTO pattern with MapStruct
- ☐ Add Swagger documentation

Phase 2: Core Patterns

- ☐ Set up Eureka Discovery Server
- ☐ Create API Gateway with Spring Cloud Gateway
- ☐ Configure centralized configuration with Config Server
- ☐ Implement database per service

Phase 3: Communication

- ☐ Implement Feign clients for inter-service calls
- ☐ Add Circuit Breaker with Resilience4j
- ☐ Set up RabbitMQ for async communication
- ☐ Implement event-driven architecture

Phase 4: Infrastructure

- ☐ Dockerize all services
- ☐ Create docker-compose.yml
- ☐ Deploy to Kubernetes (local/cloud)
- ☐ Set up CI/CD pipeline with GitHub Actions

Phase 5: Observability

- ☐ Add Spring Boot Actuator
- ☐ Set up Prometheus + Grafana
- ☐ Implement distributed tracing with Zipkin
- ☐ Configure structured logging

Phase 6: Advanced

- ☐ Implement Saga pattern for distributed transactions
- ☐ Add CQRS for complex queries
- ☐ Implement API composition
- ☐ Add Redis caching

Phase 7: Security

- [ ] Implement JWT authentication
- [ ] Add role-based access control
- [ ] Configure rate limiting
- [ ] Set up service-to-service security

Phase 8: Testing

- [ ] Write unit tests (80% coverage)
- [ ] Add integration tests with TestContainers
- [ ] Implement contract tests
- [ ] Set up load testing with JMeter/Gatling

🎓 COMPLETION CHECKLIST

After completing all stories, you should be able to:

- [ ] Design microservices from scratch using DDD
- [ ] Implement service discovery and API gateway
- [ ] Handle synchronous and asynchronous communication
- [ ] Implement distributed transactions with Saga pattern
- [ ] Containerize and orchestrate with Docker/Kubernetes
- [ ] Set up complete CI/CD pipeline
- [ ] Implement comprehensive observability
- [ ] Secure microservices properly
- [ ] Test microservices at all levels
- [ ] Answer any interview question about microservices

Good luck with your microservices journey!

This guide is based on your Food Delivery System project. All examples can be directly applied to your codebase.