21. 실전 프로젝트 설계

도메인 주도 설계(DDD)

1️⃣ DDD란?

기본 개념

- **복잡한 비즈니스 요구사항**을 효과적으로 반영하기 위한 **소프트웨어 설계 방법론**
- **비즈니스 중심적 설계** → "기술"이 아니라 "도메인 모델"이 설계의 중심이 됨
- 개발자와 도메인 전문가(비즈니스 담당자)가 **공통 언어(ubiquitous language)** 를 기반으로 모델을 정제
- **비즈니스 로직**을 모델 안에 명확하게 녹여낸다 → "Rich Domain Model"

2️⃣ DDD 핵심 원칙

원칙	설명
도메인 모델 중심 설계	비즈니스 모델을 코드의 구조와 1:1로 매핑
Ubiquitous Language	개발자와 도메인 전문가가 공유하는 언어 사용
모델링	실세계 도메인을 분석하여 적절한 추상화 모델을 설계
경계 컨텍스트 (Bounded Context)	모델의 의미가 일관되게 유지되는 범위를 정의
명시적 설계	엔티티, 값 객체, 도메인 서비스 등 명확한 설계 구분

3️⃣ DDD 주요 패턴

3.1 Entity

- **식별자 (ID)** 를 갖는 객체
- 식별자를 기준으로 동일성을 판단
- 변경 가능한 상태를 가짐

```
@Entity
public class Order {
    @Id
    private Long id;

    private OrderStatus status;

    // Business logic
    public void complete() {
        if (status == OrderStatus.PAID) {
            status = OrderStatus.COMPLETED;
        }
```

```
    }
}
```

3.2 Value Object (VO)

- 고유 식별자 없음
- 속성 값으로 동등성 판단
- 불변(Immutable) 객체 설계 권장

```
@Embeddable
public class Address {
    private String street;
    private String city;
    private String zipcode;

    // equals and hashCode must be overridden based on field values
}
```

3.3 Aggregate

- **도메인의 일관성을 유지하는 트랜잭션 경계**
- 하나 이상의 Entity + Value Object 로 구성됨
- 반드시 하나의 **Aggregate Root** 가 존재 → 외부에서 Aggregate 내부 Entity 직접 접근 불가

```
Order (Aggregate Root)
└── OrderItem (Entity)
```

- 외부에서는 항상 **Order** 를 통해 **OrderItem** 을 관리해야 함

3.4 Repository

- **Aggregate Root** 를 저장/조회하는 역할 담당
- 일반적으로 Interface + 구현체 구성

```
public interface OrderRepository {
    Order findById(Long id);
    void save(Order order);
}
```

Spring Data JPA 사용 시:

```
public interface OrderRepository extends JpaRepository<Order, Long> {}
```

3.5 Domain Service

- Entity 또는 Value Object 에 넣기 어려운 **도메인 로직**을 담당
- 비즈니스 개념이지만 특정 Entity 에 속하지 않는 경우 사용

```
public class PaymentService {
    public void pay(Order order, PaymentInfo info) {
        // 결제 처리 로직
    }
}
```

3.6 Application Service

- **유스케이스 단위**의 API를 제공
- 트랜잭션 관리, 도메인 서비스 호출 등을 담당
- 비즈니스 로직은 도메인 영역에 위임

```
@Service
@Transactional
public class OrderApplicationService {

    private final OrderRepository orderRepository;
    private final PaymentService paymentService;

    public void completeOrder(Long orderId) {
        Order order = orderRepository.findById(orderId);
        paymentService.pay(order, ...);
        order.complete();
    }
}
```

4️⃣ Bounded Context (경계 컨텍스트)

개념

- DDD에서 가장 중요한 개념 중 하나
- **모델의 의미와 규칙이 일관되게 유지되는 경계**를 정의
- 서비스/팀/모듈을 나누는 기준으로도 활용 가능
- Bounded Context 간에는 **명시적 인터페이스(Integration)** 를 정의

예시

```
[ 주문(Order) 컨텍스트 ]      [ 결제(Payment) 컨텍스트 ]      [ 배송(Shipping) 컨텍스트 ]
→ Context 별로 모델/용어가 다를 수 있음
→ API / Event / Shared Kernel 등으로 연계
```

---

5 DDD 레이어드 아키텍처

```
+--------------------+
| Presentation Layer | → REST API / Controller
+--------------------+
| Application Layer  | → Application Service / 유스케이스 중심
+--------------------+
| Domain Layer       | → Entity / VO / Domain Service / Business Logic
+--------------------+
| Infrastructure     | → Repository 구현, External System 연동
+--------------------+
```

---

6 DDD 적용 시 장점

✅ **비즈니스 복잡성을 코드 구조로 명확히 반영 가능**
✅ 개발자와 비즈니스 담당자 간 **용어 정렬(언어의 일치)**
✅ 코드의 **변경 용이성 향상**
✅ 큰 시스템에서 **경계(Context) 관리가 용이**
✅ Domain Model 기반으로 다양한 전략(이벤트 소싱, CQRS 등) 확장 가능

---

7 DDD 적용 시 단점

❌ 초기 학습 비용 존재 → 조직/팀 전체가 이해해야 함
❌ 설계에 더 많은 시간 필요 → 단순 CRUD 서비스에는 과도할 수 있음
❌ 모델과 경계를 잘못 정의하면 오히려 복잡성 증가
❌ Value Object, Aggregate, Entity 등 구분이 어려울 때가 있음 (초기 경험 부족 시 혼란)

---

8 적용 추천 상황

상황	DDD 적용 여부
단순 CRUD 시스템	추천 ❌ (단순 레이어드 아키텍처 사용)
복잡한 비즈니스 도메인 (B2B, 금융, 커머스 등)	추천 ✅
서비스 간 Bounded Context 가 명확한 경우	추천 ✅
유스케이스 중심 설계가 필요한 경우	추천 ✅
개발팀과 비즈니스 팀 간 협업이 빈번한 경우	추천 ✅

9️⃣ 정리

핵심 개념	역할
Entity	식별자 기반 비즈니스 객체
Value Object	값 기반 객체 (불변)
Aggregate	일관성 트랜잭션 경계
Repository	Aggregate 영속화 책임
Domain Service	Entity에 넣기 어려운 도메인 로직
Application Service	유스케이스 중심 서비스
Bounded Context	명확한 도메인 경계 정의

TDD 기반 개발

1️⃣ TDD란?

정의

- **Test-Driven Development (테스트 주도 개발)**
- 개발 시 "테스트 코드 → 프로덕션 코드" 순으로 진행하는 개발 방법론
- **Red → Green → Refactor** 주기 기반 개발

핵심 흐름

```
1. 실패하는 테스트 작성 (Red)
2. 테스트가 통과할 만큼의 최소한의 코드 작성 (Green)
3. 코드 개선(리팩토링) (Refactor)
→ 반복
```

2️⃣ TDD의 기본 사이클 (Red → Green → Refactor)

1단계: Red

- 테스트 먼저 작성
- 아직 기능 구현이 없으므로 테스트가 **실패**해야 정상

```
@Test
void shouldReturnZeroWhenNoItems() {
    ShoppingCart cart = new ShoppingCart();
    assertEquals(0, cart.getTotalPrice());
}
```

2단계: Green

- 테스트 통과를 위해 **가장 단순한 코드 작성**

```
public class ShoppingCart {
    public int getTotalPrice() {
        return 0;
    }
}
```

3단계: Refactor

- 중복 제거, 코드 개선
- 테스트는 여전히 **Green 상태 유지**가 목표

3️⃣ TDD의 장점

✅ 높은 코드 품질 → 코드가 테스트로 "검증 가능"하게 설계됨
✅ 빠른 피드백 루프 → 기능 깨짐을 조기에 발견 가능
✅ 코드 변경 시 리팩토링 안전성 확보
✅ 요구사항을 "실행 가능한 문서"로 만들 수 있음 (테스트 = 문서)
✅ 테스트 커버리지 자연스럽게 확보됨

4️⃣ TDD와 일반 개발 비교

측면	일반 개발	TDD
코드 작성 순서	프로덕션 코드 → 테스트 코드	테스트 코드 → 프로덕션 코드
테스트 작성 시기	구현 완료 후 작성	구현 전 작성
테스트 커버리지	낮거나 불균일	높음
코드 품질	개발자 역량에 따라 차이 큼	일관된 품질 확보 용이
리팩토링 안전성	낮음	높음

5️⃣ TDD의 3가지 법칙 (Kent Beck)

1️⃣ **아직 실패하지 않은 테스트에 대한 프로덕션 코드는 작성하지 않는다**
2️⃣ **실패하는 테스트가 나오도록 테스트 코드를 작성한다**
3️⃣ **현재 실패하는 테스트를 통과할 만큼만 프로덕션 코드를 작성한다**
→ 과도한 설계/구현 방지

---

6 테스트 종류와 TDD 적용 범위

테스트 수준	설명	TDD 적용 가능 여부
단위 테스트 (Unit Test)	가장 작은 단위 (클래스/메서드) 테스트	가장 강하게 적용
통합 테스트 (Integration Test)	시스템 간 상호작용 테스트 (DB, API 등 포함)	제한적으로 적용
UI 테스트	화면 / 사용자 인터랙션 테스트	TDD 적용 어려움 (가능은 함)

→ TDD는 주로 **단위 테스트** 레벨에서 적극 활용됨
→ 통합 테스트/시스템 테스트는 **검증용 추가 단계**로 구성

---

7 스프링 부트에서 TDD 개발 흐름

주요 도구

도구	설명
JUnit 5	테스트 프레임워크
Mockito	Mock 객체 생성
AssertJ	표현력 높은 Assertion 지원
Spring Boot Test	스프링 통합 테스트 지원 (`@SpringBootTest`, `@DataJpaTest` 등)

일반적인 구성 흐름

```
1. 테스트 클래스 작성 → @Test 붙이기
2. 기능 실패 확인 (Red)
3. 최소한의 코드 구현 (Green)
4. 리팩토링 (Refactor)
→ 반복
```

예시

```
@SpringBootTest
public class UserServiceTest {

    @Autowired
    private UserService userService;

    @Test
    void shouldCreateNewUser() {
        UserDto dto = new UserDto("Alice", "alice@example.com");
        User user = userService.createUser(dto);

        assertThat(user.getId()).isNotNull();
        assertThat(user.getName()).isEqualTo("Alice");
```

```
    }
}
```

8️⃣ TDD 적용 시 주의사항

✅ **테스트는 가독성이 좋아야 한다**
✅ **테스트 코드는 프로덕션 코드보다 더 많은 "비용"을 들여서 작성하는 것이 정상**
✅ TDD 는 "테스트 먼저 작성"이지 "테스트 커버리지를 100% 만드는 것"이 아님
✅ **Mock 과 Stub 남용 주의** → 지나친 Mocking 은 코드 유지보수성을 해침
✅ 테스트가 깨졌을 때 원인을 쉽게 파악할 수 있어야 함

9️⃣ DDD + TDD 함께 적용하기

- DDD에서 도메인 모델은 **풍부한 비즈니스 로직을 담아야 함**
- 이를 TDD 기반으로 검증 가능 → 도메인 모델 단위 테스트 매우 중요

예시: Order Entity 테스트

```
@Test
void orderShouldTransitionToCompletedWhenPaid() {
    Order order = new Order();
    order.pay();
    order.complete();

    assertEquals(OrderStatus.COMPLETED, order.getStatus());
}
```

→ 도메인 규칙이 테스트 코드로 **명확하게 표현됨**

→ → 좋은 TDD 기반 DDD 코드 == "코드만 봐도 비즈니스 규칙을 이해할 수 있음"

1️⃣0️⃣ TDD 적용 Best Practice

✅ "테스트 하기 쉬운 구조"로 코드를 설계하게 됨 → **Low Coupling / High Cohesion**
✅ 테스트는 **빠르게 실행**되어야 함 → 단위 테스트는 몇 초 이내
✅ **CI/CD 파이프라인에 자동 테스트 포함 필수**
✅ 도메인 핵심 로직은 반드시 **TDD 기반으로 작성**
✅ UI/REST API 레이어는 **통합 테스트 수준에서 적절히 테스트 구성** (UI 레이어는 과도한 TDD 적용 비효율적임)

결론

TDD 기반 개발은 단순한 "테스트 먼저 작성하기"가 아님.
→ **"코드가 검증 가능하도록 설계하는 사고방식"** 에 가깝다.
→ DDD 와 매우 잘 어울리는 개발 방법 → **도메인 로직의 품질**을 극대화할 수 있음.

실전에서는:

- 도메인 계층 → **철저하게 TDD 적용**
- Application / Service 계층 → **유스케이스 중심 TDD 적용**
- Infra / 외부 API 연동 → **적절한 통합 테스트 중심 구성**

헥사고날 아키텍처 적용

1️⃣ 정의

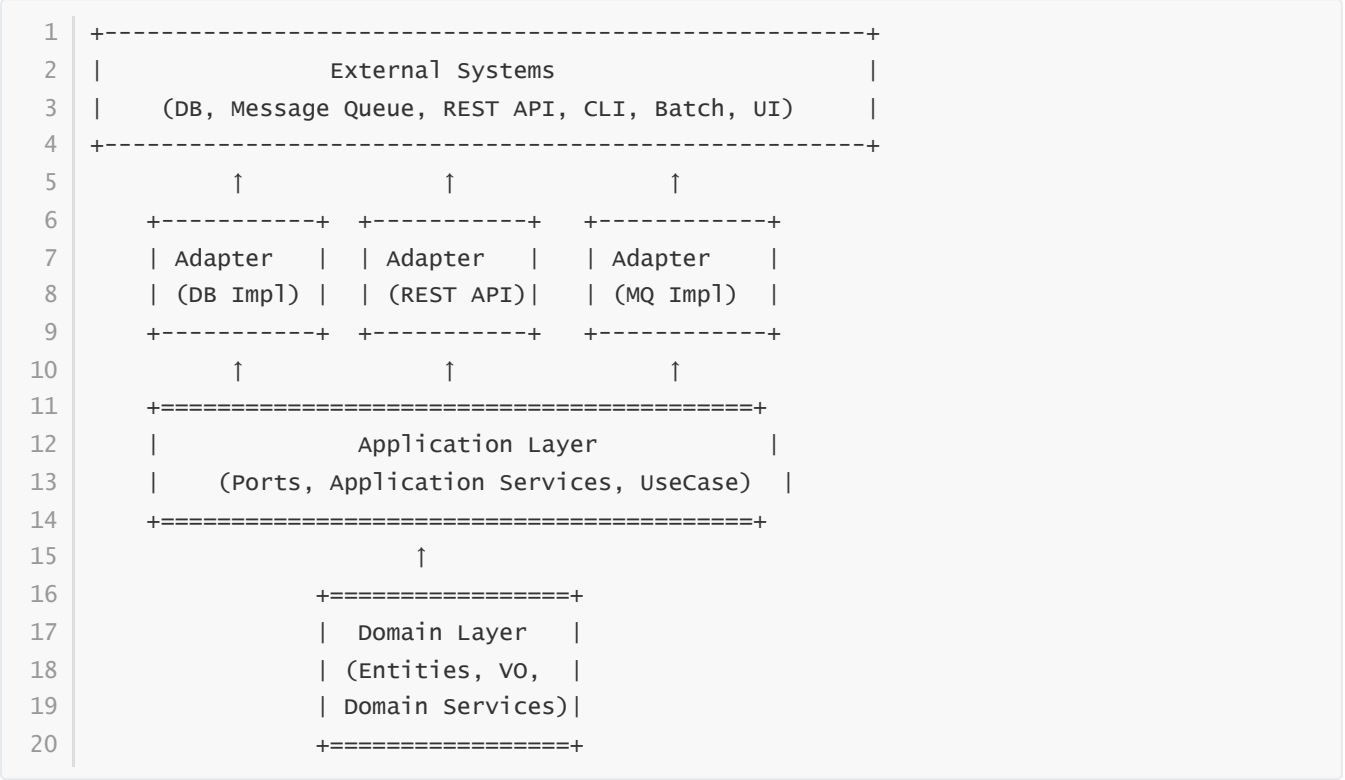
다른 이름들

- **Ports and Adapters Architecture**
- **Hexagonal Architecture** → 육각형으로 그려서 붙은 이름
- **Clean Architecture**, **Onion Architecture** 와 개념상 유사

기본 개념

- **애플리케이션의 핵심 비즈니스 로직(Domain, Application Layer)** 을 외부 세계로부터 **명확히 격리**한다.
- **"포트(Port)"와 "어댑터(Adapter)"** 를 통해 외부와 통신한다.
- 의존성은 항상 **밖에서 안으로만 향함** → Domain 은 외부에 절대 의존하지 않음.

2️⃣ 구조도



---

3 핵심 구성 요소

구성 요소	역할
Domain Layer	비즈니스 규칙, 핵심 로직, Entity, VO
Application Layer	유스케이스/프로세스 구현, Port 인터페이스 제공
Port	외부 시스템과 연결하는 인터페이스 정의
Adapter	외부 시스템 구현체 (DB, MQ, REST API Client 등)

---

4 흐름 설명

내부 → 외부

- Domain 은 외부에 의존 X → 독립적
- Application Layer 는 **Port Interface** 만 의존 → 어댑터가 실제 구현체 제공

외부 → 내부

- 외부에서 들어오는 이벤트 (REST API 호출, MQ 수신 등)는 Adapter 에서 처리
- Adapter 가 Application Layer Port 를 호출 → 비즈니스 로직 실행

---

5 Port와 Adapter

Port

- Interface
- Domain/Application 이 외부에 요청하고 싶은 기능을 정의

```
public interface PaymentPort {
    PaymentResult requestPayment(PaymentRequest request);
}
```

Adapter

- Port 의 구현체 → 외부 시스템과 직접 통신

```
@Component
public class PaymentAdapter implements PaymentPort {

    @Override
    public PaymentResult requestPayment(PaymentRequest request) {
        // 실제 외부 PG API 호출
    }
}
```

Application Layer에서는

```
@Service
@Transactional
public class OrderService {

    private final PaymentPort paymentPort;

    public OrderService(PaymentPort paymentPort) {
        this.paymentPort = paymentPort;
    }

    public void placeOrder(Order order) {
        // 비즈니스 로직
        paymentPort.requestPayment(...);
    }
}
```

---

6 헥사고날 아키텍처 적용시 장점

✅ **Domain 완벽 격리** → 변경에 강함
✅ 다양한 Adapter 교체 가능 → 예: DB 교체, MQ 교체 시 Domain 영향 없음
✅ 테스트 용이 → Port 인터페이스 Mock 사용 가능
✅ 통합 테스트/단위 테스트 경계 명확
✅ Application Layer가 **유스케이스 중심 설계**에 집중 가능

---

7 적용 시 패키지 구조 예시 (Spring Boot)

```
com.myapp
├── domain
|   ├── model
|   ├── service
|   ├── event
├── application
|   ├── port
|   |   ├── in
|   |   └── out
|   ├── service
├── adapter
|   ├── in
|   |   ├── web (REST API Adapter)
|   |   └── messaging (MQ Consumer 등)
|   ├── out
|   |   ├── persistence (DB Adapter)
|   |   └── external (REST API Client 등)
├── config
|   └── BeanConfig.java
```

Port 구분

방향	위치	역할
IN Port	`port.in`	외부 → Application Layer (UseCase Interface)
OUT Port	`port.out`	Application Layer → 외부 (Repository, API 등)

Adapter 구분

방향	위치	역할
IN Adapter	`adapter.in`	REST Controller, Event Consumer 등
OUT Adapter	`adapter.out`	Repository 구현, 외부 API Client 등

8️⃣ DDD + TDD + Hexagonal Architecture 적용 흐름

계층	테스트 적용 전략
Domain Layer	TDD 적극 적용 (Entity, VO, Domain Service)
Application Layer (Port/UseCase)	TDD 적용 (Application Service 단위 테스트)
Adapter	통합 테스트 중심 (Adapter만 별도 테스트)

9️⃣ 실전 주의사항

✅ Adapter 에서 비즈니스 로직 넣지 말 것 → Adapter 는 오직 I/O 책임만 가짐
✅ Port 에는 **비즈니스 의미 중심 인터페이스** 정의 → 기술적 의존 용어 피하기
✅ Domain Layer 는 절대 외부 기술에 의존하지 말 것 (Spring @Component 등 사용 X)
✅ Application Layer에서 **Port 주입은 명시적으로 구성** (의존성 명확히 함)

🔟 결론

헥사고날 아키텍처는:

- 시스템을 **비즈니스 로직 중심**으로 설계하고
- **외부 변화에 유연하게 대응**하게 만들고
- **테스트와 개발 생산성을 극대화**하는 설계 패턴.

스프링 부트 프로젝트에서 **DDD + TDD + Hexagonal Architecture** 조합은
복잡한 시스템을 견고하고 확장 가능하게 만드는 실전적 최상 구성으로 매우 많이 채택되고 있어.

CQRS / 이벤트 소싱

1️⃣ CQRS란?

Command Query Responsibility Segregation (명령과 조회의 책임 분리)

기본 개념

- **명령(Command)**: 데이터를 변경하는 요청 → ex) 주문 생성, 결제 처리
- **조회(Query)**: 데이터를 읽어오는 요청 → ex) 주문 목록 조회, 사용자 정보 조회
- **쓰기 모델**과 **읽기 모델**을 **분리**하여 구성

기존 CRUD 구조

```
+------------+
| Controller |
+------------+
      ↓
+------------+
| Service    | → RDB (CRUD)
+------------+
```

→ 모든 요청이 **하나의 도메인 모델 / 서비스 계층**을 통과

CQRS 구조

```
Command Flow:                Query Flow:

+------------+               +------------+
| Controller |               | Controller |
+------------+               +------------+
      ↓                            ↓
+------------+               +------------+
| CommandService |            | QueryService |
+------------+               +------------+
      ↓                            ↓
+------------+               +------------+
| Command Model |            | Query Model |
+------------+               +------------+
      ↓                            ↓
+------------+               +------------+
| Event Store |              | Read DB    |
+------------+               +------------+
```

2️⃣ CQRS 적용 이유

장점

✅ 쓰기/읽기 성능 최적화 가능 (각각 별도로 튜닝)
✅ 읽기 모델을 **조회 특화 모델로 구성 가능** (ex: denormalized view)
✅ 고성능 조회 API 구성 가능 → BI/Analytics에 유리
✅ 복잡한 비즈니스 도메인에서 **쓰기 모델 복잡성**과 **조회 모델 단순성**을 분리 가능
✅ **이벤트 기반 아키텍처**와 잘 어울림

단점

❌ 시스템 복잡성 증가
❌ 데이터 정합성 관리 필요 (Eventual Consistency 발생 가능)
❌ 운영 인프라 비용 증가 (Read DB 별도 필요)

3️⃣ 이벤트 소싱 (Event Sourcing)

기본 개념

- 기존 시스템에서는 **현재 상태(state)** 를 RDB 테이블 등에 저장
- 이벤트 소싱은 **모든 상태 변경 이벤트 자체를 영속화** → 현재 상태는 이벤트 재생(Replay)으로 구성 가능

예시: Order Aggregate

순서	이벤트
1	OrderCreated(orderId=1, items=3)
2	OrderItemAdded(orderId=1, item=ABC)
3	OrderPaid(orderId=1, paymentId=XYZ)
4	OrderShipped(orderId=1, trackingNo=123)

→ 현재 상태 = **이벤트 1~4 재생 결과**

4️⃣ 이벤트 소싱과 CQRS 관계

개념	역할
CQRS	Command/Query 를 분리하여 각 흐름 최적화
Event Sourcing	상태 변경을 Event 기반으로 저장

→ 둘은 **매우 자연스럽게 결합됨**:

1️⃣ Command → Domain → Event 발생 → Event Store 저장
2️⃣ Event → Read Model Projection → Query Model 갱신

→ Read Model 은 별도 테이블 / NoSQL / ElasticSearch 등 사용 가능

---

5 이벤트 소싱의 장단점

장점

✅ 모든 변경 기록 보존 → **Audit/History 용도**에 최적
✅ Rollback 및 복구 가능 → 이벤트 Replay
✅ Read Model 을 다양하게 구성 가능 → BI/Analytics
✅ CQRS 와 결합 시 성능/확장성 극대화

단점

❌ 모델 설계가 어려움 (Event 설계 신중 필요)
❌ 이벤트 재생 비용 발생 가능
❌ 이벤트 스키마 변경 시 Migration 필요
❌ Eventually Consistent 구조를 잘 이해하고 설계 필요

---

6 기술 구성 예시

구성 요소	추천 기술
Command Layer	Spring Boot + Domain Layer
Event Store	RDB 테이블 / EventStoreDB / Kafka / Axon Server
Read Model	RDB Read Table / Elasticsearch / Redis
Event Bus	Kafka / RabbitMQ / Axon Framework
Query Layer	Spring Boot Query API (QueryService)

---

7 Spring Boot 기반 CQRS + Event Sourcing 예시 흐름

Command 흐름

```
Client → REST API → CommandService → Aggregate 처리 → Domain Event 발행 → EventStore 저장 → Event Bus 발행
```

Query 흐름

```
Event Bus 수신 → Read Model Projection → Read Model 저장 → REST API → Client
```

---

8 단순 예시 코드

Domain Event 정의

```
public class OrderCreatedEvent {
    private final String orderId;
    private final List<String> items;

    // Constructor, Getter
}
```

CommandService 예시

```
public class OrderService {

    private final EventPublisher eventPublisher;

    public void createOrder(String orderId, List<String> items) {
        // Domain Logic 처리
        OrderCreatedEvent event = new OrderCreatedEvent(orderId, items);

        // Event 저장 + Event Bus 발행
        eventPublisher.publish(event);
    }
}
```

Projection 예시 (Read Model 갱신)

```
@Component
public class OrderProjection {

    @EventListener
    public void handle(OrderCreatedEvent event) {
        // Read DB 업데이트 → ReadModelRepository.save(...)
    }
}
```

---

9 결론

구성 요소	CQRS 적용 여부	Event Sourcing 적용 여부
단순 CRUD 서비스	비추	비추
복잡한 도메인 / 고성능 조회 요구	적극 추천	추천
Event Driven Architecture 기반	추천	적극 추천
MSA 환경에서 서비스간 상태 공유 필요	추천	적극 추천

패턴 요약:

✅ CQRS → Command/Query 흐름 분리
✅ Event Sourcing → 상태 변경 자체를 Event 기반으로 저장
✅ 둘을 함께 사용 → **대규모 확장성 / 고성능 시스템** 설계 가능

1️⃣0️⃣ 적용 단계

1️⃣ 먼저 **CQRS 만 도입** → Command/Query 흐름 분리
2️⃣ 다음 단계에서 **Event Sourcing 점진적 도입** → 핵심 Aggregate 부터 적용
3️⃣ Event Projection Layer 구성 → BI/Reporting 에도 활용

MSA 전환 프로젝트 구조

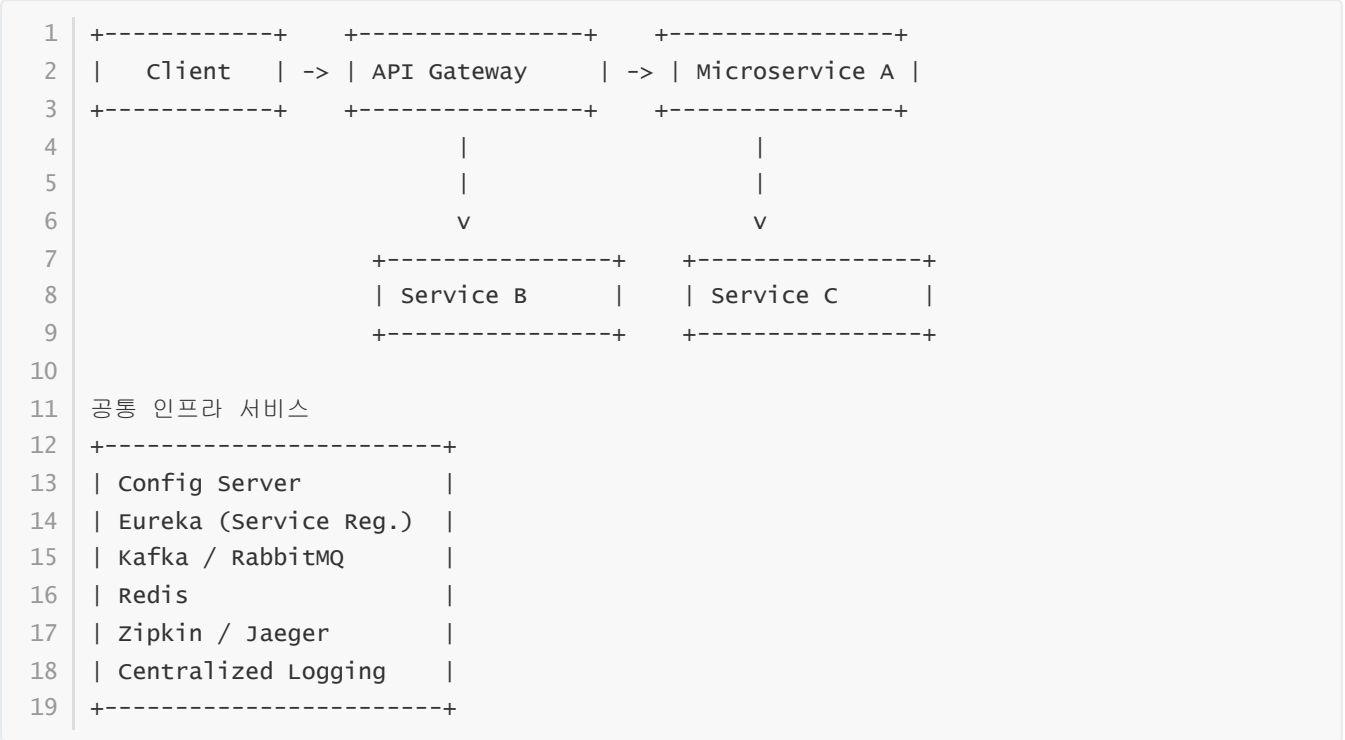
1️⃣ 전환 기본 전략

모놀리식 → MSA 로 "한 번에 갈아엎는 것"은 대부분 실패함 → 단계적 전환이 실전에서 성공 전략

단계

단계	설명
1단계	기존 모놀리식 서비스에서 **Bounded Context 식별**
2단계	**독립 서비스 단위**로 분리 설계
3단계	서비스별 별도 배포 → 기존 시스템과 공존
4단계	점진적으로 Monolith 기능 제거 → 완전한 MSA 구성

2️⃣ 기본적인 MSA 서비스 구조



3️⃣ 전환 시 프로젝트 구조 예시 (Spring Boot 기반)

1단계: 초기 구조 (모놀리식 내부 Bounded Context 식별)

```
com.company.app
├── order
├── payment
├── user
├── inventory
├── common
```

→ 내부 레이어는 분리되어 있지만 **물리적으로는 하나의 Deployable Artifact** (jar/war)

2단계: 서비스별 프로젝트로 분리

```
repo-root
├── order-service
├── payment-service
├── user-service
├── inventory-service
├── common-lib (공통 코드 라이브러리)
└── infra (infra 구성, ex: Helm chart, Docker Compose 등)
```

→ Gradle/Maven 멀티모듈 구조 or Git Monorepo 구성

→ 또는 Git Repo 별도 분리도 가능

3️⃣ 서비스 내부 구조 예시

```
order-service
├── src/main/java/com.company.order
│   ├── adapter
│   │   ├── in (API Adapter: REST Controller 등)
│   │   ├── out (Persistence Adapter, External API Adapter 등)
│   ├── application
│   │   ├── port.in
│   │   ├── port.out
│   │   ├── service
│   ├── domain
│   │   ├── model
│   │   ├── service
│   │   ├── event
├── src/test/java
├── build.gradle
└── Dockerfile
```

→ **Hexagonal Architecture 적용 구조 추천** → 유지보수성 / 테스트 용이성 최적화됨

---

4 서비스간 통신 구조

통신 유형	용도	기술 예시
REST API (Sync)	실시간 요청 응답	Spring Web / Feign Client
Event 기반 (Async)	상태 변경 전파	Kafka / RabbitMQ / Spring Cloud Stream
Shared Database	X (반드시 피해야 함)	-

→ MSA 전환 시 **서비스간 DB 공유는 반드시 금지** → API 또는 Event 기반 통신으로 전환

---

5 공통 인프라 서비스 구성

구성 요소	역할	기술 예시
Config Server	구성 정보 중앙 관리	Spring Cloud Config
Service Registry	서비스 인스턴스 관리	Eureka / Consul
Gateway	API Gateway 기능	Spring Cloud Gateway / NGINX / Kong
Event Bus	서비스간 비동기 통신	Kafka / RabbitMQ
Centralized Logging	로그 수집 및 분석	ELK Stack (Elasticsearch, Logstash, Kibana)
Distributed Tracing	서비스 호출 흐름 추적	Zipkin / Jaeger
Security	인증/인가	OAuth2 / JWT / Keycloak

---

6 전환 시 점진적 적용 전략

1. 먼저 **읽기 API 부터 분리** → ReadModel 서비스 별도 분리 가능 (CQRS 적용 시작)
2. 이후 **신규 기능은 MSA 기반으로 개발**
3. 기존 기능은 서비스 단위로 Gradual Migration
4. 기존 Monolith 와 MSA 서비스간 **연계 브리지 레이어** 설계 필요 (ex: API Adapter)
5. 점진적 Traffic Migration → 최종적으로 Monolith 기능 제거 가능

---

7 조직 구조 관점

- MSA 전환은 기술적 작업 뿐 아니라 **조직 구조도 변화 필요**

역할	추천 구조
서비스 오너십	서비스별 전담팀 구성
배포 파이프라인	서비스별 CI/CD 구성
모니터링 / 장애 대응	서비스별 모니터링 구성 + 공통 운영팀 구축

역할	추천 구조
API 계약 관리	API 문서화 (Swagger/OpenAPI 기반) + Contract Test 적용

---

8 결론

MSA 전환 프로젝트 구조는:

- ✅ 서비스 단위 **독립 배포** 가능 구조로 재구성
- ✅ 서비스 내부는 Hexagonal Architecture 추천
- ✅ 서비스간 통신은 REST + Event 기반으로 설계
- ✅ Config / Service Discovery / Gateway / Observability 체계 필수 구축
- ✅ 조직/운영 체계도 **서비스 중심 구조**로 전환 필요