16. 모듈화 및 멀티모듈 프로젝트 구성

모노리식 구조의 한계

모노리식 아키텍처(Monolithic Architecture) 는 애플리케이션을 **단일 코드베이스**로 구성하여, 모든 기능(웹, API, 서비스 로직, 데이터 접근 등)을 **하나의 프로세스** 안에서 실행하는 구조이다.

초기 개발 및 배포는 간단하지만, 시스템 규모가 커지고 복잡해질수록 여러 한계가 발생한다.

1️⃣ 구조적 특성

- 단일 프로세스 → 하나의 애플리케이션 패키지 (JAR, WAR 등)로 배포
- 공통 데이터베이스 사용
- 모듈 구분은 코드 레벨에서만 존재 (패키지 구조로만 나뉨)
- 전체 빌드 및 전체 배포 필요

2️⃣ 한계 및 문제점 ⚠️

2.1 복잡성 증가에 따른 유지보수 어려움

- 기능이 많아질수록 코드베이스가 비대해짐.
- 모듈 간 경계가 모호해지고, **의존성 꼬임**(dependency spaghetti)이 발생하기 쉬움.
- 신규 개발자 온보딩 시 **전체 코드 이해**가 필수 → 학습 비용 증가 🤯.

2.2 배포/빌드 속도 저하

- 작은 기능 수정 시에도 전체 애플리케이션을 **재빌드** → **재배포**해야 함.
- 빌드 시간 증가 → 배포 속도 저하 🚧.
- 배포 시 다운타임 발생 가능성 ↑ → 고가용성 확보 어려움.

2.3 확장성(Scalability) 한계

- 전체 서비스가 하나의 프로세스에 묶여 있기 때문에 **독립적 확장 불가**.
- 일부 모듈만 높은 트래픽 발생 시, 전체 애플리케이션 인스턴스를 스케일 아웃해야 함 → 리소스 낭비
- 예: 검색 서비스만 고성능 필요 → 전체 애플리케이션 스케일 업 필요.

2.4 장애 전파 위험성

- 하나의 모듈에서 발생한 장애가 전체 프로세스에 영향을 미침.
- 메모리 누수, 스레드 고갈 등이 전역에 영향을 줌 → 서비스 전체 다운 가능성 ↑.

2.5 기술 스택 제한

- 단일 프로세스 → 전체 모듈이 **동일 기술 스택/프레임워크**에 종속됨.
- 일부 기능에 특화된 언어나 프레임워크 사용 어려움 (ex. ML 모듈을 Python으로 따로 구성하기 어려움).

2.6 팀 규모 확대 시 협업 한계

- 많은 개발자가 하나의 코드베이스에서 협업 시 **충돌/병합 비용** 증가.
- 모듈 간 코드 소유권 불명확 → 코드 품질 저하 가능성 ↑.
- CI/CD 파이프라인 과부하 발생 가능.

2.7 CI/CD 운영 부담

- 배포 주기를 빨리 가져가려 해도 **전체 서비스의 안정성 검증** 필요.
- 단일 프로세스 배포 시 신규 기능 릴리즈 → 기존 기능에 영향 가능성 ↑ → QA 비용 증가.

3️⃣ 정리 비교표 🗂️

영역	모노리식 한계
유지보수성	코드 비대화, 의존성 꼬임 발생
빌드/배포	전체 재빌드/재배포 필요, 다운타임 발생 가능
확장성	부분 확장 불가, 전체 스케일 필요
장애 대응	장애 전파 가능성 높음
기술 스택 유연성	단일 기술 스택에 종속
협업	팀 규모 증가 시 충돌 빈도 증가
CI/CD	빠른 배포 주기 구현 어려움, QA 비용 증가

4️⃣ 결론 🚀

✅ 모노리식 아키텍처는 **초기 개발/운영 단계**에서는 생산성이 높고 간단한 구조를 제공한다.
✅ 하지만 서비스 규모가 커지거나 팀 규모가 확대될 경우 유지보수, 확장성, 장애 대응 측면에서 **명확한 한계**에 직면하게 된다.
✅ 이러한 한계를 극복하기 위해 마이크로서비스 아키텍처(MSA) 또는 모듈화 전략 적용이 고려된다.

멀티 모듈 프로젝트 구성 (`:core`, `:api`, `:batch`)

Spring Boot 기반 애플리케이션에서 **멀티 모듈 프로젝트(Multi-Module Project)** 는 다음과 같은 목적을 위해 사용된다:

- **코드 재사용성 향상** → 공통 모듈 분리
- **모듈 간 경계 명확화** → 의존성 관리 강화

- **빌드 속도 최적화** → 모듈 단위 빌드
- **독립 배포 구조 마련 (단계적 MSA 전환도 수월)**

이번 절에서는 실무에서 많이 사용하는 `:core`, `:api`, `:batch` 구조를 예로 들어 설명한다.

1️⃣ 기본 구조 예시 📦

```
project-root/
├── build.gradle (root)
├── settings.gradle
├── core/          # :core 모듈 (공통 라이브러리, 도메인 모델, 유틸 등)
|   └── build.gradle
├── api/           # :api 모듈 (REST API 서버)
|   └── build.gradle
├── batch/         # :batch 모듈 (배치 처리용)
|   └── build.gradle
└── gradle/        # (선택) gradle wrapper
```

모듈 간 관계

```
core → api 사용 가능
core → batch 사용 가능
api ↔ batch 간 직접 의존성 금지 (간접 참조 필요 시 core 통해서만 가능)
```

👉 모듈 간 **단방향 의존성** 원칙 준수 → 코드 품질/유지보수성 향상 🚀.

2️⃣ settings.gradle 구성

모듈 등록.

```
rootProject.name = 'my-multi-module-project'

include 'core'
include 'api'
include 'batch'
```

3️⃣ 모듈별 구성 설명

:core 모듈

- **공통 유틸리티**
- **도메인 모델 (Entity, DTO 등)**
- **공용 인터페이스/추상 클래스**
- **공통 Exception 정의**
- 외부에 노출되지 않음 (라이브러리 역할).

```
plugins {
    id 'java'
}

dependencies {
    // 공통 라이브러리 예시
    implementation 'org.springframework.boot:spring-boot-starter-data-jpa'
    implementation 'org.springframework.boot:spring-boot-starter-validation'

    // Lombok 사용 시
    compileOnly 'org.projectlombok:lombok'
    annotationProcessor 'org.projectlombok:lombok'

    testImplementation 'org.springframework.boot:spring-boot-starter-test'
}
```

:api 모듈

- Spring Boot 기반 **REST API 서버** 역할
- UI / API Gateway 연동 가능
- `@SpringBootApplication` 위치 모듈

```
plugins {
    id 'org.springframework.boot' version '3.2.4'
    id 'io.spring.dependency-management' version '1.1.4'
    id 'java'
}

dependencies {
    implementation project(':core')

    implementation 'org.springframework.boot:spring-boot-starter-web'
    implementation 'org.springframework.boot:spring-boot-starter-security'
    implementation 'org.springframework.boot:spring-boot-starter-data-jpa'

    runtimeOnly 'com.mysql:mysql-connector-j'

    testImplementation 'org.springframework.boot:spring-boot-starter-test'
}
```

주의

- 반드시 `@SpringBootApplication`은 **api 모듈에 위치** (core에는 X).
- core → api는 의존 금지 (순환 참조 발생 위험).

:batch 모듈

- Spring Boot 기반 **Batch 처리 전용 모듈**
- Spring Batch, Quartz 등을 활용한 비동기/주기적 처리
- 배치 전용 Application Runner 구성 가능.

```
plugins {
    id 'org.springframework.boot' version '3.2.4'
    id 'io.spring.dependency-management' version '1.1.4'
    id 'java'
}

dependencies {
    implementation project(':core')

    implementation 'org.springframework.boot:spring-boot-starter-batch'
    implementation 'org.springframework.boot:spring-boot-starter-jdbc'
    runtimeOnly 'com.mysql:mysql-connector-j'

    testImplementation 'org.springframework.boot:spring-boot-starter-test'
}
```

BatchApplication 예시:

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

👉 :batch 모듈은 독립적으로 실행 가능 → 운영 시 **Batch 전용 Profile 구성** 가능.

4️⃣ 의존성 원칙 🎯

```
core 모듈 → 다른 모듈에서 재사용 허용
api 모듈 ↔ batch 모듈 간 직접 참조 금지 (core 통해서만 의존)
```

- 공통 데이터 모델은 **core에서 정의**.
- 각 모듈별 책임과 경계 분리 → 유지보수성과 확장성 향상.

5️⃣ 빌드 전략

- 전체 빌드 → `./gradlew build`
- 특정 모듈 빌드 → `./gradlew :api:build`, `./gradlew :batch:build`

👉 CI/CD 파이프라인에서 **모듈 단위 빌드/배포 최적화 가능** 🚀.

6️⃣ 장점 & 한계

항목	장점	한계
코드 분리	모듈 간 명확한 경계 구성	초기 설계 시 의존성 관리 필요
빌드 최적화	모듈 단위 빌드 가능 → 속도 향상	Gradle 설정 복잡성 ↑
배포 전략	모듈 별 독립 배포 구성 가능	종속성 추적 필요
MSA 단계적 전환	모듈 → 점진적 MSA 분리 가능	MSA 전환 시 infra 설계 추가 필요

결론 🗂️

✅ 멀티 모듈 구성은 **대규모 서비스** 개발 시 모듈화를 통해 코드 품질과 빌드/배포 전략을 최적화할 수 있다.
✅ `:core`, `:api`, `:batch` 구성은 실무에서 많이 사용하는 **전형적인 구조**로, 유지보수성과 확장성을 높인다.
✅ 모듈 간 의존성 원칙을 엄격하게 지켜야 **장기적인 코드 품질**을 유지할 수 있다.
✅ 점진적 **MSA 전환 기반 구성으로도 활용** 가능하다.

공통 모듈, 도메인 분리 전략

멀티 모듈 프로젝트에서 **공통 모듈(core)** 과 **도메인 별 모듈 분리 전략**은 **코드 품질 유지**와 **서비스 확장성 확보**의 핵심이다.
구조적 설계가 제대로 되어 있지 않으면 모듈 간 의존성 꼬임, 테스트 복잡도, 빌드 속도 저하 등의 문제가 발생한다.

이번 절에서는 다음을 기준으로 설명한다:

- 공통 모듈(core)
- 도메인 별 모듈 설계 전략
- 의존성 방향 설계 원칙
- 공통 모듈 설계 시 주의사항

1️⃣ 공통 모듈(core)의 역할 🎁

핵심 원칙

- **공통적인 코드만 포함** → 특정 도메인 종속적인 코드는 절대 포함하지 않음 ❌.
- 모든 모듈에서 **재사용 가능하도록 설계**.
- core 모듈 변경 시 영향 범위가 넓어지므로 **변경 안정성** 최우선 고려.

주요 구성 요소

영역	포함 여부
DTO (공용 API 규약용)	O
Enum (공용 비즈니스 용어 정의)	O
Exception 정의	O
유틸리티 클래스	O
공용 인터페이스/추상 클래스	O
Entity (공통으로 참조되는 도메인 모델)	△ (주의 필요)
서비스 로직	❌
컨트롤러	❌
리포지토리 구현체	❌

공통 예시 구성

```
core/
└── src/main/java/com/example/core
    ├── dto
    ├── exception
    ├── enums
    ├── utils
    ├── model  (Entity 포함 시 주의 깊게 설계)
    ├── constants
```

Entity 포함 시 주의사항 ⚠️

- 특정 도메인에 종속적인 Entity는 core로 분리 금지.
- **정말 공용으로 필요한 것만** → ex. Member, Organization, AuditEntity 등.

2️⃣ 도메인 모듈 분리 전략 🧩

기본 설계 원칙

- 도메인마다 **독립된 모듈**로 구성 → 관심사 분리(Separation of Concerns).
- 각 모듈 내부에서 **Service Layer, Domain Layer, Repository Layer** 구성.
- 도메인 모듈 간 **직접 참조 금지** → 필요한 경우 core 또는 명시적 API Layer 통해 통신.

예시 구조

```
project-root/
├── core/
├── api/
├── batch/
├── domain-user/
|   ├── UserEntity, UserService, UserRepository
├── domain-order/
|   ├── OrderEntity, OrderService, OrderRepository
├── domain-product/
|   ├── ProductEntity, ProductService, ProductRepository
```

모듈 간 의존성

```
core → 모든 도메인 모듈에서 참조 가능
domain-x ↔ domain-y 직접 참조 금지
api → 필요 도메인 모듈 참조
batch → 필요 도메인 모듈 참조
```

👉 도메인 간 직접 참조를 허용하면 **결합도 증가 → 유지보수성 급격히 저하** 🚧.

3️⃣ 의존성 방향 설계 원칙 🔄

방향	허용 여부
core → domain-x	❌ (금지)
domain-x → core	✅ (허용)
api → domain-x	✅ (허용)
batch → domain-x	✅ (허용)
domain-x → domain-y	❌ (금지)

👉 도메인 간 통신은 **core 계층의 공통 인터페이스** 또는 **Event/Message 기반 통신**으로 설계 권장.

4️⃣ 공통 모듈 설계 시 주의사항 ⚠️

경계 흐림 주의

- core에 "공통처럼 보이는 것"이 무분별하게 쌓이면 → **도메인 침투** 발생 위험.
- 공통으로 묶을 때는 반드시 → **다중 모듈에서 실질적 재사용성 검증** 후 이동.

Entity 설계 시 주의

- Entity → 무조건 core로 올리지 않기.
- 공통 모델의 경우에도 **모듈 간 변경 리스크**가 크므로 꼭 필요한 경우만 포함.

예:

- `AuditEntity` (BaseEntity) → 가능
- `User`, `Order` 등 → 각 도메인 모듈 소속 유지.

5️⃣ 모듈간 의존성 예시 (Gradle)

api/build.gradle

```
implementation project(':core')
implementation project(':domain-user')
implementation project(':domain-order')
```

batch/build.gradle

```
implementation project(':core')
implementation project(':domain-order')
implementation project(':domain-product')
```

domain-x/build.gradle

```
implementation project(':core')
```

👉 모든 도메인 모듈은 core만 참조, 타 도메인 모듈 참조 금지 🚫.

6️⃣ 종합 원칙 🗂️

영역	설계 원칙
core 구성	정말로 공통인 것만 포함
Entity 위치	도메인 모듈 내 위치 유지 (공통 Entity는 예외적 허용)
도메인간 참조	직접 참조 금지, Event/Message 기반 통신 권장
의존성 방향	core → X / domain-x → core O / api → domain-x O
서비스 계층 구성	각 도메인 모듈 내부에 서비스/리포지토리 구성

결론 🚀

- ✅ **멀티 모듈 + 도메인 분리**는 서비스가 커질수록 유지보수성과 확장성을 크게 향상시킨다.
- ✅ 공통 모듈(core)은 **최소한의 범위**로 유지하고, 각 도메인 모듈은 **독립성**을 최대한 확보하는 것이 핵심.
- ✅ 도메인 간 결합도는 **Event/Message 기반으로 느슨하게 유지**하여 시스템의 탄력성을 확보해야 한다.
- ✅ **점진적 MSA 전환**의 기반으로도 멀티 모듈 + 도메인 분리 전략은 매우 유효하다.