|  |  |
| --- | --- |
| **개발표준 정의서** | |
| **AI 기반 성과 평가 시스템**  **Version 1.2**  SK㈜ AX | |
|  |  |
| SK AX | |

**제.개정 이력**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **버전** | **제.개정 일자** | **제.개정 내용** | **작성자** | **승인자** |
| 1.0 | 2025-07-01 | 최초 등록 | 김재현 | 김재현 |
| 1.1 | 2025-07-02 | 목차 정리 및 내용 추가 | 신택수 | 신택수 |
| 1.2 | 2025-07-03 | 초안 완성 | 신택수 | 신택수 |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |

**Index / 목차**

[**1. 개요 5**](#_ddcofl46xlev)

[1.1. 개발 표준 정의서의 필요성 5](#_vj5qxmkgfvwv)

[1.2. 적용 범위 5](#_fllde6csb5dw)

[1.3. 개발 표준 정의서 적용의 예외 5](#_vhlhq8yh9qwu)

[**2. 소프트웨어 아키텍처 6**](#_nls4lt13gie8)

[2.1. 시스템 아키텍처 개요 6](#_bayru7s8y28s)

[2.2. 주요 기술 스택 6](#_wbcmpkmrvony)

[2.3. 컴포넌트별 아키텍처 6](#_j6robxvo2f3e)

[2.4. CI/CD 파이프라인 표준 7](#_so0q8qle9fqc)

[2.4.1. 개발 워크플로우 (GitHub -> Jenkins -> ArgoCD) 7](#_6jmc5lxzz2yt)

[2.4.2. 컨테이너화 및 배포 (Docker & Kubernetes) 8](#_8ujx1ixaea7)

[2.4.3. 시스템 아키텍처 구성도 8](#_2salyxf9it54)

[**3. 개발 표준 9**](#_tzu6b810gbxk)

[3.1. Git 사용 전략 9](#_bwnwloo482j5)

[3.1.1. 브랜칭 전략 (Git-flow 기반) 9](#_wkvacq3fhckr)

[3.1.2. 커밋 메시지 규칙 10](#_6ulsh547js46)

[3.2. 명명 규칙 (Naming Convention) 11](#_3pgs5pkcw4gm)

[3.2.1. 케이스 스타일 (Case Styles) 11](#_6tei0m5lbjdw)

[3.2.2. 공통 명명 규칙 11](#_cu1l4u51zfkb)

[3.3. 코딩 스타일 (Coding Style) 12](#_cdms8roh4iax)

[3.3.1. 들여쓰기 및 공백 12](#_jtjqymhwen3d)

[3.3.2. 모듈/파일 라인 수 12](#_6ot23zni6o27)

[3.4. 주석 처리 표준 12](#_s2gk7u7uofas)

[3.5. API 공통 표준 12](#_5imz2r1vcreb)

[3.5.1. RESTful API 설계 원칙 12](#_43kjapwoqr6n)

[3.5.2. 공통 응답(Response) 형식 13](#_1y3kpci2c1cr)

[3.5.3. 공통 에러 코드 정의 13](#_52mromw42wce)

[3.6. 공통 예외 처리 및 로깅 표준 14](#_x7n3ue3ghp84)

[**4. 프론트엔드(FE) 개발 표준 15**](#_70o0pevd980h)

[4.1. 프로젝트 구조 15](#_s4jpc0iwjh14)

[4.1.1. 최상위 디렉터리 구조 15](#_x591kp1homhq)

[4.1.2. 컴포넌트 (Components) 디렉터리 구조 15](#_rtts3vpola1i)

[4.1.3. API 디렉터리 구조 16](#_9mwvs9qdlqpq)

[4.2. 상태 관리 표준 (Zustand) 16](#_ni58kdgn4kb6)

[4.2.1. Store의 기본 원칙 16](#_9xh0f3ujvtep)

[4.2.2. Store 구조 17](#_sh61suimohj)

[4.2.3. Store 사용 규칙](#_r0qxvs18iakx) 18

[4.3. 컴포넌트 개발 표준 (React) 18](#_h18sewh0hhgx)

[4.3.1. 컴포넌트 작성 원칙 18](#_tfu16d9dpmns)

[4.3.2. 명명 규칙 18](#_p1r7jtvfk596)

[4.3.3. 컴포넌트 파일 구조 19](#_gli6yj9c6qer)

[4.3.4. 훅 사용 규칙 20](#_lftutls6l3gc)

[4.4. API 연동 표준 (Axios) 20](#_cht4de5bsgbx)

[4.4.1. Axios 인스턴스 관리 20](#_4irkvoc8vi63)

[4.4.2. API 함수 구조 20](#_n5rvmnmxb4c5)

[4.4.3. 에러 처리 표준 21](#_65t4ojb32qy6)

[4.5. 스타일링 표준 (Tailwind CSS) 22](#_8nmm6jj42ap0)

[4.5.1. 기본 원칙 (Utility-First) 22](#_m7bfok3m9wqh)

[4.5.2. 설정 및 테마 관리 (tailwind.config.js) 22](#_m4vlwa4iifjb)

[4.5.3. 커스텀 컴포넌트 클래스 (@apply) 22](#_uohbnf9qwnd5)

[4.5.4. 반응형 디자인 (Responsive Design) 23](#_sbe18c3dpqcx)

[4.5.5. 동적 클래스 사용 시 주의사항 23](#_wypwdifelcln)

[4.6. 타입스크립트 (TypeScript) 사용 표준 24](#_ae8ukq6vnjhb)

[4.6.1. 기본 원칙 24](#_tyeog7kmacj9)

[4.6.2. 타입 정의 및 관리 24](#_ynwjf9urb5kd)

[4.6.3. API 데이터 타입 정의 24](#_5swq87rh50a8)

[4.6.4. 유틸리티 타입 활용 25](#_oxlufo46cj0b)

[**5. 백엔드(BE) 개발 표준 26**](#_figstmvhvz8w)

[5.1. 프로젝트 구조 (패키지 구조) 26](#_7m2l1dby9c1w)

[5.1.1. 최상위 패키지 구조 26](#_jo7cqt6cey32)

[5.1.2. 도메인 패키지 (domain) 구조 26](#_w7dngxac313y)

[5.1.3. 글로벌 패키지 (global) 구조 27](#_lyyimu59qbk8)

[5.2. Controller 개발 표준 27](#_ww4dsz4wtoz6)

[5.2.1. 기본 구조 및 책임 27](#_l27ko2vup6rx)

[5.2.2. API 엔드포인트 개발 표준 28](#_7aryjmkuk5w6)

[5.2.3. 응답(Response) 처리 표준 28](#_1f1jfpspc22q)

[5.2.4. API 문서화 29](#_a7x1nv6n6in9)

[5.3. Service 개발 표준 30](#_y0ruj939d23t)

[5.3.1. 기본 구조 및 책임 30](#_ezyzd5mz9tnt)

[5.3.2. 비즈니스 로직 구현 30](#_yp8pcbc6er8d)

[5.3.3. 예외 처리 31](#_j1xbicbaaxly)

[5.3.4. 명명 규칙 31](#_2sn66z14bzci)

[5.4. 데이터 접근 계층 표준 (Spring Data JPA) 31](#_1zzurmzgyqj4)

[5.4.1. Entity 작성 규칙 31](#_oacy9s3v3g7o)

[5.4.2. Repository 작성 규칙 32](#_kicdr43rxzva)

[5.4.3. DTO 정의 표준 33](#_y7o754n51sl4)

[5.5. 인증/인가 표준 (Spring Security & JWT) 33](#_y55p49kdx1v7)

[5.5.1. 기본 아키텍처 33](#_vt5qye88ov3b)

[5.5.2. 인증 (Authentication) 흐름 34](#_ey54r5bl3sio)

[5.5.3. 인가 (Authorization) 방식 34](#_bn5tuo4dshf4)

[5.5.4. 주요 컴포넌트 및 책임 34](#_jrnfkuzb8oxq)

[5.6. DTO 사용 표준 34](#_j8ji8pz1ne3w)

[5.6.1. 핵심 원칙: Entity 보호 34](#_9w2nvugp8rz0)

[5.6.2. DTO 명명 및 작성 규칙 34](#_91bxxscg5ldb)

[5.6.3. 변환 로직 35](#_djgevfs1304r)

[**6. AI 백엔드 개발 표준 37**](#_psejy9vtww7q)

[6.1. 프로젝트 구조 37](#_3uyfbhy4k2p2)

[6.1.1. 최상위 디렉터리 구조 37](#_xu9hnp6hbzl8)

[6.1.2. 핵심 기능 계층 구조 37](#_a3vac061w6e2)

[6.1.3. 공통 모듈 구조 38](#_ymi8ka21zeai)

[6.2. API 개발 표준 (FastAPI) 38](#_msv12cx5t8iw)

[6.2.1. 라우팅 (Routing) 및 컨트롤러 38](#_wr77kzfwxcg7)

[6.2.2. 데이터 유효성 및 DTO (Schema) 39](#_tdavj64ggurl)

[6.2.3. 의존성 주입 40](#_xcwxogyaugba)

[6.2.4. 비동기 처리 40](#_27moi2l8gleo)

[6.3. LangGraph 활용 표준 40](#_nv3jihnny78x)

[6.3.1. LangGraph 그래프(Graph) 기반 워크플로우 구성 원칙 40](#_6ukhgcicqo8p)

[6.3.2. 프롬프트 관리 표준 42](#_hori84byz97e)

[6.3.3. 모델 출력 처리 표준 43](#_w0n8tr194xwt)

[6.3.4. 에이전트 및 메모리 활용 표준 44](#_2yvqm45z0m22)

[6.4. 벡터 DB (Pinecone) 연동 표준 44](#_pm9iogpg57jx)

[6.4.1. 기본 원칙 및 구조 44](#_zd6qp0icb4x9)

[6.4.2. 데이터 색인 표준 45](#_xghb0h3p9gxu)

[6.4.3. 데이터 검색 표준 46](#_szn4v92497ek)

[6.5. 파이썬 코딩 스타일 47](#_20q2cvp5ih1y)

[6.5.1. 코드 포매터 (Code Formatter) 47](#_h8ssoxynlu8z)

[6.5.2. 린터 (Linter) 47](#_fqy678gv881k)

[6.5.3. 명명 규칙 47](#_ue0hobsx5b7)

[6.5.4. 타입 힌트 48](#_g4skwk2go9co)

[6.5.5. 독스트링과 주석 49](#_p1f4qcpz1g8a)

[**7. 인프라 및 환경 표준 50**](#_cmsqt3n276q0)

[7.1. Dockerfile 작성 표준 50](#_4tp95b9hvf6l)

[7.1.1. 멀티 스테이지 빌드 (Multi-stage Build) 사용 50](#_rf4zmzgvlm4h)

[7.1.2. 기본 이미지 선택 50](#_ar9zhfwv2cdc)

[7.1.3. 명령어 작성 규칙 51](#_8u1mniza8evb)

[7.2. Jenkinsfile 작성 표준 51](#_j1c4zce3m0v3)

[7.2.1. 선언형 파이프라인 사용 51](#_mb4ozvghjr4o)

[7.2.2. 환경 변수 관리 51](#_xt3n85ata8bd)

[7.2.3. 스테이지 구성 52](#_1tcrs6l8c9q2)

[7.2.4. 보안 및 인증 52](#_8xf3izdedfmr)

[7.2.5. GitOps 기반 배포 자동화 53](#_16tqycdmanh)

[7.2.6. 파이프라인 후처리 53](#_eu5cxfw63brk)

[7.3. Kubernetes YAML 작성 표준 54](#_dk1nqhwgh85o)

[7.3.1. 리소스 구조 및 관리 54](#_4eci6nireqa0)

[7.3.2. 메타데이터 표준 54](#_tbthct56fw4s)

[7.3.3. Deployment 명세 표준 55](#_radwpsw9nz4u)

[7.3.4. Service 및 Ingress 명세 표준 55](#_2kuyib2kdemr)

[7.3.5. GitOps - ArgoCD Application 표준 55](#_dip99z3pve7)

[7.4. 환경 변수 관리 표준 56](#_trnrqod9rktk)

[7.4.1. 기본 원칙: 설정의 외부화 56](#_hax2fx2rz3hy)

[7.4.2. 민감 정보 관리: Kubernetes Secret 사용 56](#_sy1lgsliqn6y)

[7.4.3. 일반 설정 정보 관리 57](#_7yvcjewvpvkr)

[7.4.4. 환경별 설정 관리 57](#_zfm6kyx78lmr)

# 개요

본 문서는 '성과 관리 프로젝트'의 성공적인 수행을 위한 개발 표준을 정의하는 가이드입니다. 프로젝트에 참여하는 모든 개발자는 본 문서에 명시된 **아키텍처와 개발 표준**을 숙지하고 준수하여, **일관성 있고 품질 높은 시스템**을 개발해야 합니다.

## 개발 표준 정의서의 필요성

개발 표준을 통해 얻을 수 있는 기대 효과는 다음과 같습니다.

* **생산성 향상:** 소스 코드의 가독성과 일관성을 확보하여 신규 개발 및 인수인계 시 생산성을 높입니다.
* **원활한 협업:** 표준화된 구조와 용어를 사용하여 개발자 간의 커뮤니케이션 비용을 줄이고 효율적인 협업을 지원합니다.
* **품질 향상:** 코드의 품질을 높여 잠재적인 오류를 줄이고, 안정적인 시스템 운영과 효율적인 유지보수를 가능하게 합니다.

## 적용 범위

본 개발 표준은 '성과 관리 프로젝트'와 관련된 모든 개발 활동 및 산출물에 적용됩니다. 적용 대상의 상세 범위는 다음과 같습니다.

* **프론트엔드 (Frontend):** React, TypeScript를 기반으로 하는 모든 UI/UX 개발 산출물
* **백엔드 (Backend):** Spring Boot를 기반으로 하는 모든 API 및 비즈니스 로직 개발 산출물
* **AI 서비스:** FastAPI, Python을 기반으로 하는 AI 모델 연동 및 관련 API 개발 산출물
* **데이터베이스 (Database):** MariaDB, Redis의 스키마 설계 및 데이터 접근 관련 로직
* **CI/CD 및 인프라:** Git, Docker, Kubernetes 등 프로젝트의 빌드, 테스트, 배포, 운영과 관련된 모든 설정 및 스크립트

따라서 프로젝트에 참여하는 모든 구성원(FE, BE, AI, DevOps 담당자 등)은 본 문서에 정의된 표준을 숙지하고 준수해야 합니다.

## 개발 표준 정의서 적용의 예외

프로젝트 진행 중 표준을 적용하기 어려운 예외 상황이 발생할 경우, 임의로 판단하지 않고 반드시 개발 표준 담당자와 협의해야 합니다. 협의를 통해 결정된 예외 사항과 대응 방안은 모든 팀원에게 투명하게 공유하여 혼선을 방지합니다.

페이지의 마지막 줄

# 소프트웨어 아키텍처

## 시스템 아키텍처 개요

본 프로젝트의 시스템은 각 기능이 독립적으로 개발 및 배포될 수 있도록 프론트엔드(FE), 백엔드(BE), AI 서비스로 분리하여 구성합니다. 모든 서비스는 Docker 컨테이너로 패키징되며, Kubernetes(K8s) 클러스터 환경에서 운영되어 확장성과 안정성을 보장합니다.

사용자는 React 기반의 프론트엔드를 통해 시스템과 상호작용하며, 프론트엔드는 백엔드 API를 호출하여 비즈니스 로직을 처리합니다. Spring Boot로 개발된 백엔드 서버는 데이터 영속성을 위해 MariaDB를 주 데이터베이스로 사용하고, 성능 향상을 위해 Redis를 캐시로 활용합니다. 인증/인가 관리는 Spring Security와 JWT(JSON Web Token) 방식을 통해 안전하게 처리됩니다.

AI 관련 기능은 Python/FastAPI로 구현된 별도의 AI 서버가 담당하며, Langchain과 벡터 DB인 Pinecone을 활용하여 고도화된 기능을 제공합니다.

전체 개발 파이프라인은 GitHub, Jenkins, ArgoCD를 연동한 CI/CD 환경을 통해 자동화되며, 개발자가 작성한 코드는 빌드, 테스트, 컨테이너화 과정을 거쳐 Kubernetes 클러스터에 안정적으로 배포됩니다.

## 주요 기술 스택

|  |  |
| --- | --- |
| 구분 | 주요 기술 및 버전 정보 |
| 언어 | Java (Open JDK 17 LTS 이상), Python, TypeScript |
| 프론트엔드 | React (19.1.0 이상), TypeScript, Zustand, Tailwind CSS |
| 백엔드 | Spring Boot (3.5.0), Spring Data JPA, Spring Security, JWT |
| AI | FastAPI, LangGraph, Pinecone |
| 데이터베이스 | MariaDB (11.4.6), Redis |
| CI/CD & 인프라 | Docker, Kubernetes, Jenkins, ArgoCD |
| 개발 도구(IDE) | IntelliJ IDEA (2023.2 이상) 또는 Eclipse (2023-03 이상) |
| WAS | Spring Boot 내장 WAS (Tomcat 10.1.41) |

## 컴포넌트별 아키텍처

|  |  |  |
| --- | --- | --- |
| 구분 | 제품명(제조사) | 주요 규격 및 기능 |
| 프론트엔드 (FE) | React (Meta) | • TypeScript 기반의 SPA (Single Page Application) • Zustand를 이용한 전역 상태 관리 • Tailwind CSS 기반 반응형 스타일 적용 • 백엔드 API와 비동기 통신 |
| 백엔드 (BE) | Spring Boot (Pivotal) | • 비즈니스 로직 처리 및 RESTful API 제공 • Spring Security, JWT 기반 인증/인가 • Spring Data JPA를 통한 데이터 영속성 관리 • Redis를 활용한 캐시 처리 |
| AI 서버 | FastAPI (Tiangolo) | • Python 기반 AI 추론 API 서버 • Langchain을 활용한 LLM 워크플로우 관리 • Pinecone 벡터 DB 연동 및 검색 • AI 연산 작업의 독립적 수행 |

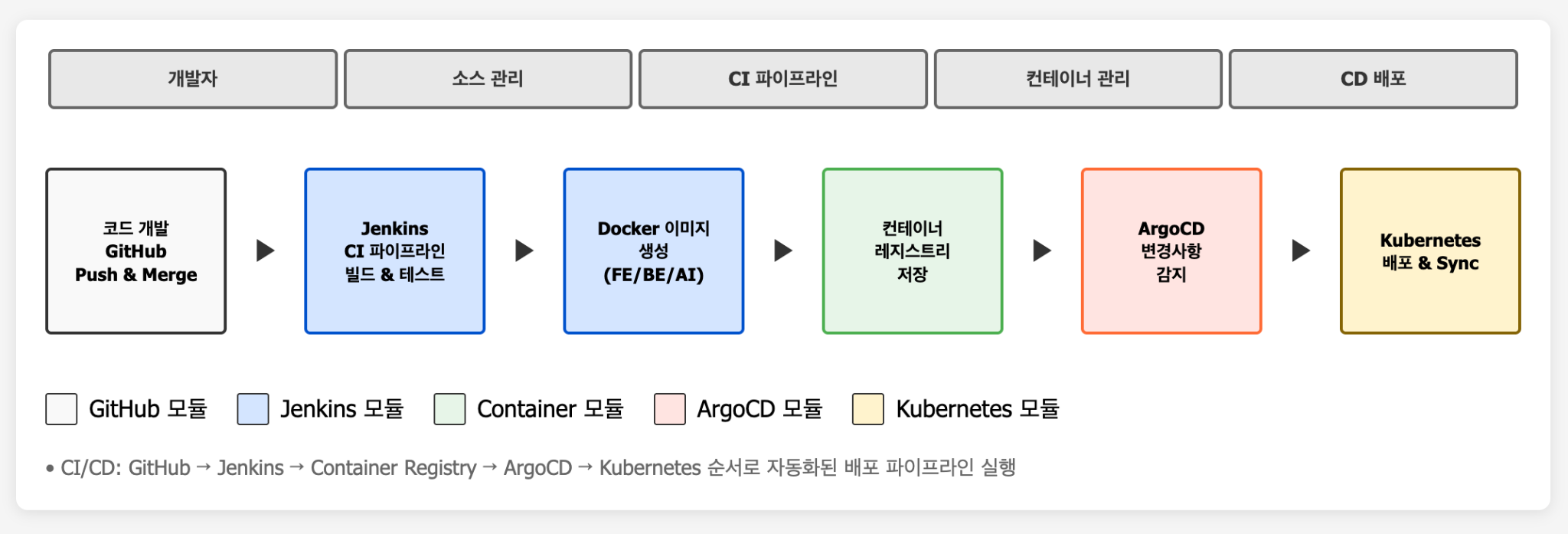
## CI/CD 파이프라인 표준

본 프로젝트는 코드의 통합부터 배포까지의 전 과정을 자동화하여 개발 생산성을 높이고 안정적인 서비스를 제공하는 것을 목표로 합니다.

### 개발 워크플로우 (GitHub -> Jenkins -> ArgoCD)

모든 소스 코드는 **GitHub**를 통해 형상 관리되며, 특정 브랜치에 코드가 통합(Merge)되면 CI/CD 파이프라인이 자동으로 실행됩니다.

1. **코드 Commit & Push**: 개발자는 기능 개발 후 자신의 코드를 GitHub에 Push 합니다.
2. **Pull Request 및 Merge**: 코드 리뷰 후 develop 브랜치로 코드를 Merge 합니다.
3. main **브랜치로 Merge**: develop 브랜치에서 테스트에 통과하면 main 브랜치로 Merge 합니다.
4. **CI (Continuous Integration) 실행**: Merge 이벤트가 **Jenkins** 파이프라인을 자동으로 트리거합니다. Jenkins는 소스 코드를 가져와 빌드 및 자동화된 테스트를 수행합니다.
5. **컨테이너 이미지 생성**: 빌드가 성공하면 Jenkins는 각 서비스(FE, BE, AI)에 맞는 **Docker 이미지를 생성**하여 컨테이너 레지스트리(Container Registry)에 Push 합니다.
6. **CD (Continuous Deployment) 실행**: Jenkins는 배포 정보를 담은 Git 리포지토리를 업데이트합니다. **ArgoCD**는 이 리포지토리의 변경 사항을 감지하고, 지정된 **Kubernetes 클러스터**에 새로운 버전의 애플리케이션을 자동으로 배포(Sync)합니다.



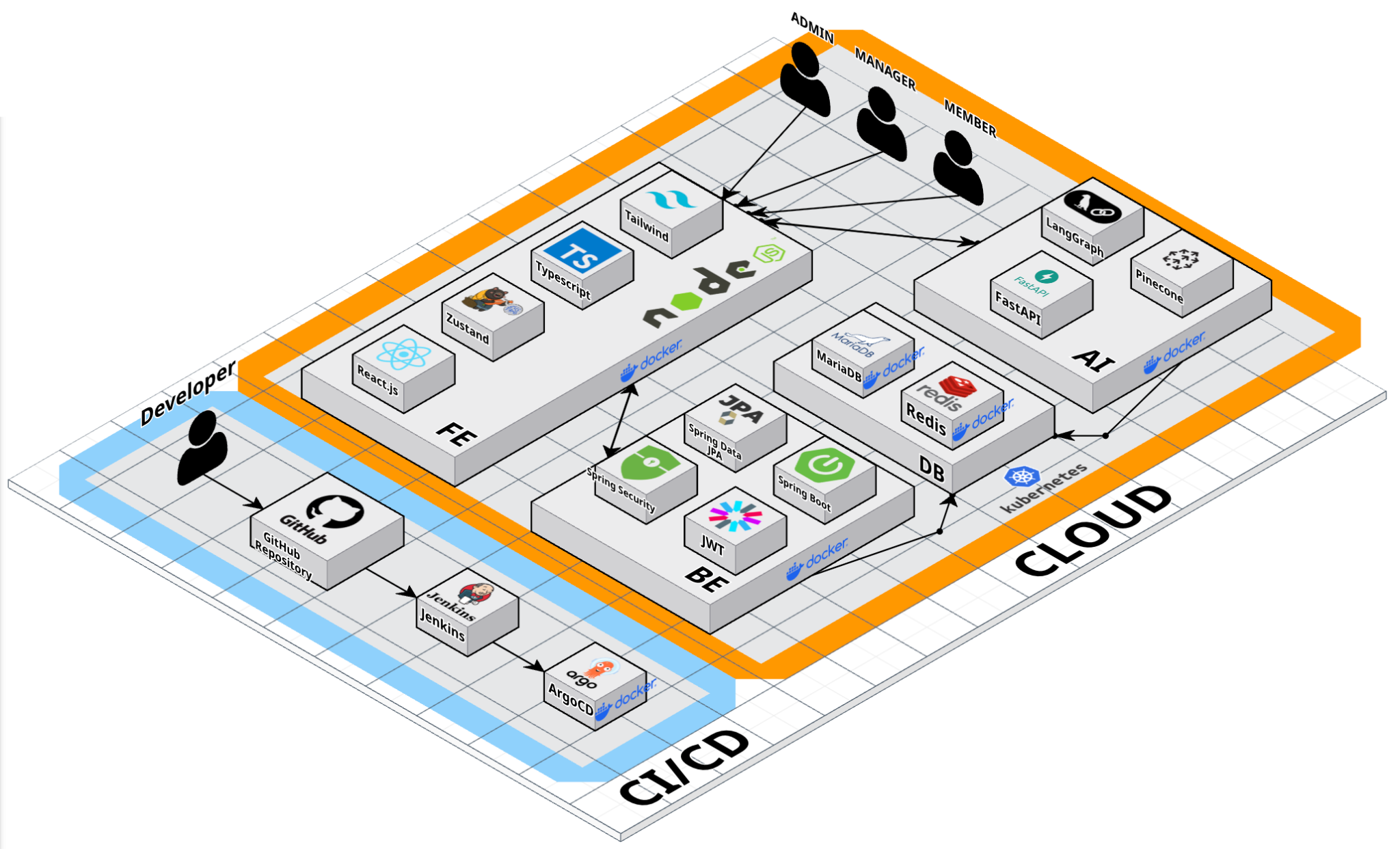
### 

### 컨테이너화 및 배포 (Docker & Kubernetes)

**Docker (컨테이너화):** 모든 애플리케이션(FE, BE, AI)은 **Dockerfile**을 통해 컨테이너 이미지로 빌드됩니다. 이를 통해 개발 환경과 운영 환경의 차이를 없애고, 어디서든 동일한 환경에서 애플리케이션을 실행할 수 있습니다.

**Kubernetes (배포 및 운영):** 컨테이너화된 애플리케이션은 **Kubernetes**를 통해 운영 환경에 배포되고 관리됩니다. Kubernetes는 오토 스케일링(Auto-scaling), 자가 치유(Self-healing), 무중단 배포(Rolling Update) 등의 기능을 통해 안정적이고 확장성 있는 서비스 운영을 지원합니다.

### 시스템 아키텍처 구성도



# 개발 표준

## Git 사용 전략

안정적인 코드 관리와 효율적인 협업을 위해 **Git-flow**를 기반으로 한 브랜칭 전략을 사용합니다. 모든 커밋 메시지는 정의된 규칙을 따라야 합니다.

### 브랜칭 전략 (Git-flow 기반)

프로젝트는 항상 배포 가능한 상태를 유지하는 **main** 브랜치와 개발의 중심이 되는 **develop** 브랜치를 중심으로 운영됩니다.

**주요 브랜치 종류 및 역할**

|  |  |  |
| --- | --- | --- |
| **브랜치 종류** | **브랜치명 규칙** | **역할 및 설명** |
| Main | main | 제품으로 출시 가능한 브랜치. 오직 develop 브랜치에서 병합(Merge)합니다. |
| Develop | develop | 다음 버전 개발을 위한 통합 브랜치. 모든 feature 브랜치가 이곳으로 병합됩니다. |
| Feature | feature/[기능명] | 단일 기능 개발을 위한 브랜치. develop 브랜치에서 생성하며, 기능명은 영어 소문자로 작성합니다. (예: feature/login-api) |
| Hotfix | hotfix/[수정내용] | main 브랜치의 긴급 버그 수정을 위한 브랜치. main에서 생성하며, 수정 후 main과 develop에 모두 병합합니다. |

**기본 작업 흐름**

* 1. develop 브랜치에서 새로운 기능 개발을 위한 feature 브랜치를 생성합니다.
  2. 기능 개발을 완료한 후, 커밋 규칙에 맞춰 코드를 커밋합니다.
  3. 개발이 완료되면 feature 브랜치를 develop 브랜치로 병합(Merge)하기 위한 Pull Request(PR)를 생성합니다.
  4. 코드 리뷰 및 CI 통과 후 develop 브랜치에 병합하고, 완료된 feature 브랜치는 삭제합니다.
  5. develop 브랜치에서 충분한 테스트가 완료되면, main 브랜치로 병합하여 새 버전을 배포합니다.

### 커밋 메시지 규칙

프로젝트의 모든 커밋 메시지는 가독성과 추적성을 높이기 위해 아래 규칙을 엄격히 따릅니다.

* **기본 형식**
  + [커밋 유형] : [커밋 내용]
* **작성 규칙**
  + 커밋 유형과 내용은 콜론(:)으로 구분하며, 콜론 뒤에는 **한 칸**을 띄웁니다.
  + 커밋 유형은 **영문 대문자**로 시작합니다. (예: Feat, Fix)
  + 커밋 내용은 **한글**로 명료하게 작성하며, 문장의 끝에 마침표를 찍지 않습니다.
* **커밋 유형**

|  |  |  |
| --- | --- | --- |
| **유형** | **설명** | **예시** |
| Feat | 새로운 기능 추가 | Feat: 로그인 API 엔드포인트 추가 |
| Fix | 버그 수정 | Fix: 특정 조건에서 발생하는 메모리 누수 해결 |
| Refactor | 기능 변경 없는 코드 구조 개선, 리팩토링 | Refactor: 중복 코드 제거 및 함수 모듈화 |
| Chore | 빌드, 설정, 문서 등 소스 코드와 직접 관련 없는 작업 | Chore: .gitignore 파일에 로그 파일 추가 |
| Style | 코드 포맷팅, 세미콜론 누락 등 스타일 관련 수정 | Style: ESLint 규칙에 맞게 코드 포맷팅 |
| Docs | 문서(README.md 등) 수정 | Docs: API 명세서 내용 보강 |
| Init | 프로젝트 초기 설정 | Init: 프로젝트 기본 구조 및 설정 파일 추가 |

### 

## 명명 규칙 (Naming Convention)

프로젝트 내 모든 리소스의 이름은 역할과 의도를 명확히 드러내야 합니다. 언어별 표준을 최대한 따르며, 프로젝트 전체의 일관성을 유지합니다.

### 케이스 스타일 (Case Styles)

#### 

|  |  |  |
| --- | --- | --- |
| **케이스 스타일** | **형식** | **주요 사용처** |
| PascalCase | EachWordIsCapitalized | 클래스(Class), 컴포넌트(Component) |
| camelCase | firstWordIsLower | 변수(Variable), 함수/메서드(Function/Method), 속성(Property) |
| UPPER\_SNAKE\_CASE | ALL\_CAPS\_WITH\_UNDERSCORES | 상수(Constant) |
| kebab-case | words-are-separated-by-hyphens | URL 경로, 프론트엔드 파일명 |
| snake\_case | words\_are\_separated\_by\_underscores | 데이터베이스 테이블 및 컬럼명 |

#### 

### 공통 명명 규칙

### 

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **구분** | **백엔드 (Java/Spring)** | **프론트엔드 (TS/React)** | **AI (Python)** | **데이터베이스** |
| 패키지/폴더 | camelCase | kebab-case | snake\_case | - |
| 클래스/컴포넌트 | PascalCase | PascalCase | PascalCase | - |
| 함수/메서드 | camelCase | camelCase | snake\_case | - |
| 변수 | camelCase | camelCase | snake\_case | - |
| 상수 | UPPER\_SNAKE\_CASE | UPPER\_SNAKE\_CASE | UPPER\_SNAKE\_CASE | - |
| 파일 | PascalCase | kebab-case | snake\_case | snake\_case |
| URL 경로 | kebab-case | kebab-case | kebab-case | - |
| 테이블/컬럼 | - | - | - | snake\_case |

### 

## 코딩 스타일 (Coding Style)

### 

코드의 가독성은 유지보수 비용과 직결됩니다. 팀 전체가 일관된 스타일을 유지하기 위해 린터(Linter)와 포맷터(Formatter) 사용을 적극 권장합니다.

### 들여쓰기 및 공백

* 들여쓰기(Indent): 탭(Tab) 대신 스페이스(Space)를 사용합니다.
  + Java, Python: 4 스페이스
  + TypeScript, React, YAML: 2 스페이스
* 공백(Space): 연산자(+, -, =, == 등) 양옆에는 공백을 추가하여 가독성을 높입니다.

### 모듈/파일 라인 수

* 하나의 파일은 300 라인, 하나의 함수/메서드는 50 라인을 넘지 않도록 작성하는 것을 권장합니다.
* 파일이나 함수가 너무 길어지면 단일 책임 원칙(SRP)에 따라 적절히 분리해야 합니다.

## 주석 처리 표준

### 주석의 기본 원칙

1. **코드는 스스로를 설명해야 한다 (Self-documenting Code)**: 좋은 변수명, 함수명을 사용하여 코드를 최대한 명확하게 작성하는 것이 최우선입니다. 주석은 코드만으로 표현하기 어려운 보조적인 정보를 제공하기 위해 사용됩니다.
2. **주석은 '왜(Why)'를 설명한다**: 주석은 코드가 '무엇을(What)' 하는지보다, '왜(Why)' 그런 방식으로 작성되었는지, 그 배경과 의도를 설명해야 합니다.
3. **주석은 코드와 함께 유지보수한다**: 코드가 변경될 때 주석도 함께 수정해야 합니다. 오래되어 실제 코드와 다른 내용을 담고 있는 주석은 없는 것보다 해롭습니다.

### 주석의 종류 및 형식

프로젝트에서는 목적에 따라 **문서화 주석**과 **구현 주석**을 구분하여 사용합니다.

### 문서화 주석

API, 클래스, 메서드, 함수 등 다른 개발자가 호출하여 사용하거나 반드시 알아야 하는 인터페이스에 대한 설명을 위해 작성합니다. 각 언어별 표준 도구로 API 문서를 생성할 수 있도록 정해진 형식을 따릅니다.

* **Java (JavaDoc)**
  + 클래스, 인터페이스, 메서드 상단에 /\*\* ... \*/ 형식으로 작성합니다.

|  |
| --- |
| /\*\*  \* 사용자 정보를 관리하는 서비스 클래스.  \* 사용자 생성, 조회, 수정, 삭제 기능을 제공합니다.  \* @author 홍길동  \* @since 2025-07-03  \*/ public class UserService {  /\*\*  \* 사용자 ID를 기반으로 사용자 정보를 조회합니다.  \*  \* @param userId 조회할 사용자의 ID (Long 타입)  \* @return 조회된 사용자 정보 DTO  \* @throws UserNotFoundException 사용자를 찾을 수 없을 때 발생  \*/  public UserDto getUserById(Long userId) throws UserNotFoundException {  // ... 구현 코드  } } |

* **TypeScript/React (JSDoc)**
  + 컴포넌트, 함수, 클래스 상단에 /\*\* ... \*/ 형식으로 작성합니다.

|  |
| --- |
| /\*\*  \* 사용자의 프로필 정보를 표시하는 카드 컴포넌트입니다.  \* @param {object} props - 컴포넌트 속성  \* @param {string} props.userId - 정보를 조회할 사용자의 ID  \* @returns {JSX.Element} 사용자 프로필 카드 UI  \*/ const UserProfileCard = ({ userId }: { userId: string }): JSX.Element => {  // ... 구현 코드 }; |

* **Python (Docstrings)**
  + 모듈, 클래스, 함수 최상단에 """ ... """ 형식으로 작성합니다. (Google Python Style 권장)

|  |
| --- |
| """사용자 관련 비즈니스 로직을 처리하는 모듈입니다."""  def get\_user(user\_id: int) -> User:  """사용자 ID로 사용자 정보를 가져옵니다.   Args:  user\_id: 조회할 사용자의 ID.   Returns:  조회된 User 객체를 반환합니다.   Raises:  UserNotFoundError: 해당 ID의 사용자를 찾을 수 없을 때 발생합니다.  """  # ... 구현 코드 |

### 구현 주석

함수 또는 메서드 내부의 복잡한 로직을 설명하거나, 특정 코드 라인에 대한 부가 설명이 필요할 때 사용합니다.

* **복잡한 로직 설명**: 특정 알고리즘이나 비즈니스 규칙이 코드만으로 이해하기 어려울 때 작성합니다.

|  |
| --- |
| // 사용자 포인트 계산: 기본 포인트 + (구매 금액 \* 0.01) - 사용 포인트 // VIP 등급의 경우, 구매 금액에 0.02를 곱함 (정책 ID: P-123) double bonusRate = user.isVip() ? 0.02 : 0.01; int finalPoint = user.getBasePoint() + (purchaseAmount \* bonusRate) - usedPoint; |

* **결정의 배경 설명**: 여러 대안 중 특정 방식을 선택한 이유를 명시합니다.

|  |
| --- |
| # A\_API는 타임아웃이 잦아 안정성이 높은 B\_API를 우선 사용하도록 처리함. response = call\_b\_api(request) |

* **태그(Tag) 주석**: 코드에 추가적인 작업이나 주의가 필요함을 명시하기 위해 표준화된 태그를 사용합니다.

|  |
| --- |
| // TODO:: 앞으로 구현해야 할 기능이나 수정이 필요할 때 // TODO: 현재는 단일 파일만 지원, 추후 다중 파일 업로드 기능 추가 필요 // FIXME:: 명확한 버그가 있어 반드시 수정되어야 할 때 // FIXME: 특정 조건에서 NullPointerException 발생 가능성이 있음 // HACK:: 깔끔하지 않은 임시방편 코드일 때 (예: 라이브러리 버그 회피) // HACK: Chart.js v2.x 버그로 인해 강제로 캔버스를 리셋함 // NOTE:: 특별히 주의해야 하거나 중요한 참고사항이 있을 때 // NOTE: 이 값은 외부 시스템과 연동되므로 임의로 변경하면 안 됨 |

### 지양해야 할 주석

1. **명백한 코드 설명 주석**: 코드를 보면 누구나 알 수 있는 내용을 굳이 주석으로 달지 않습니다.
   * **나쁜 예**: user.setAge(10); // 사용자의 나이를 10으로 설정
2. **주석 처리된 코드**: 사용하지 않는 코드는 Git을 통해 이력을 관리할 수 있으므로, 과감히 삭제합니다. 주석으로 남겨진 코드는 가독성을 해치고 혼란을 유발합니다.
3. **장식용 주석**: 별표(\*)나 해시(#) 등으로 코드 블록 주위를 감싸는 주석은 불필요한 소음(noise)을 만듭니다.

|  |
| --- |
| // **\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\****\*\*\**\* // **\*\*\*\*\***\* 사용자 로그인 처리 로직 \***\*\*\*\*\*** // **\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\****\*\*\**\* |

### 

### 

### 

### 

### 

### 

## API 공통 표준

모든 컴포넌트(FE, BE, AI) 간의 원활한 통신을 위해 일관된 API 표준을 정의합니다.

### RESTful API 설계 원칙

* 자원(Resource): URI는 리소스를 표현하며, 명사를 사용하고 복수형을 권장합니다. (예: /users, /products)
* 행위(Verb): 자원에 대한 행위는 HTTP Method(GET, POST, PUT, PATCH, DELETE)로 표현합니다.
* 표현(Representation): GET /users/1과 같이 특정 자원은 경로 변수(Path Variable)로 표현합니다. 정렬, 필터링 등은 쿼리 파라미터(Query Parameter)를 사용합니다. (예: /users?status=active)

### 공통 응답(Response) 형식

모든 API 응답은 아래와 같은 표준 구조(JSON)를 따릅니다.

* **성공 시**

|  |
| --- |
| {  "success": true,  "data": { "userId": 1, "name": "홍길동" }, // 객체 또는 배열  "error": null } |

* **실패 시**

|  |
| --- |
| {  "success": false,  "data": null,  "error": {  "code": "AUTH\_001",  "message": "인증 정보가 유효하지 않습니다."  } } |

### 공통 에러 코드 정의

에러 코드는 도메인\_코드 형식으로 정의하여 원인 파악을 용이하게 합니다.

|  |  |  |
| --- | --- | --- |
| 에러 코드 | HTTP 상태 | 메시지 |
| COMMON\_001 | 400 Bad Request | 잘못된 요청입니다. |
| AUTH\_001 | 401 Unauthorized | 인증 정보가 유효하지 않습니다. |
| AUTH\_002 | 403 Forbidden | 접근 권한이 없습니다. |
| USER\_001 | 404 Not Found | 해당 사용자를 찾을 수 없습니다. |
| SERVER\_001 | 500 Internal Server Error | 서버 내부 오류가 발생했습니다. |

### 

### 

### 

## 공통 예외 처리 및 로깅 표준

### 

* 예외 처리 (Exception Handling)
  + 예측 가능한 예외는 비즈니스 로직 내에서 처리하고, 예측 불가능한 예외는 글로벌 예외 핸들러(@RestControllerAdvice 등)를 통해 공통 처리합니다.
  + 글로벌 예외 핸들러는 위에서 정의한 공통 응답 형식에 맞춰 에러를 반환해야 합니다.
* 로깅 (Logging)
  + 로그 레벨을 명확히 구분하여 사용합니다.
    - ERROR: 심각한 시스템 오류, 처리되지 않은 예외
    - WARN: 잠재적 위험이 있지만, 시스템 동작에는 문제가 없는 경우
    - INFO: 주요 비즈니스 흐름, 상태 변경 등 의미 있는 정보
    - DEBUG: 개발 단계에서 상세한 디버깅을 위한 정보 (운영 환경에서는 비활성화)
  + 로그 형식: 로그는 JSON 형식으로 구조화하여 추적 및 분석이 용이하도록 합니다. (예: timestamp, level, service, message 등 포함)

### 

# 프론트엔드(FE) 개발 표준

## 

## 프로젝트 구조

본 프로젝트는 재사용성과 유지보수성을 극대화하기 위해 **기능 기반의 모듈형 구조**를 따릅니다. 디렉터리 구조는 기능적 관심사에 따라 명확하게 분리하여 응집도를 높이고, 다른 모듈과의 결합도를 낮추는 것을 목표로 합니다.

### 최상위 디렉터리 구조

최상위에는 src 디렉터리를 두며, 그 아래에 애플리케이션의 핵심 기능을 구성하는 하위 디렉터리들을 정의합니다.

|  |
| --- |
| src ├── apis // API 연동 및 데이터 통신 영역 ├── assets // 이미지, 폰트 등 정적 에셋 영역 ├── components // 재사용 가능한 UI 컴포넌트 영역 ├── constants // 프로젝트 전역에서 사용되는 상수 영역 ├── hooks // 커스텀 훅(Custom Hooks) 영역 ├── pages // 라우팅의 단위가 되는 페이지 컴포넌트 영역 ├── store // 전역 상태 관리(Zustand) 영역 ├── styles // 전역 스타일 및 Tailwind CSS 설정 영역 ├── types // TypeScript 타입 및 인터페이스 정의 영역 └── utils // 프로젝트 전반에서 사용되는 유틸리티 함수 영역 |

* **pages**: 애플리케이션의 각 페이지를 정의하며, 라우팅의 기본 단위입니다. 여러 컴포넌트와 비즈니스 로직을 조합하여 하나의 완전한 페이지를 구성합니다.
* **components**: 특정 도메인이나 페이지에 종속되지 않는 공용 컴포넌트와 특정 페이지 그룹에서만 사용되는 컴포넌트를 관리합니다.
* **apis**: 서버와의 데이터 통신(Axios 인스턴스, API 호출 함수 등)에 관련된 모든 로직을 관리합니다.
* **store**: Zustand를 활용한 전역 상태(Store)를 정의하고 관리합니다.
* 그 외 디렉터리는 프로젝트의 공통 기반 기능을 지원하는 역할을 합니다.

### 컴포넌트 (Components) 디렉터리 구조

컴포넌트 디렉터리는 재사용성 범위에 따라 다음과 같이 세분화하여 관리합니다.

|  |
| --- |
| src/components ├── common // 애플리케이션 전역에서 사용되는 공통 컴포넌트 ├── domain // 특정 도메인(기능)과 관련된 컴포넌트의 집합 └── layout // Header, Footer, Sidebar 등 페이지 레이아웃을 구성하는 컴포넌트 |

* **common**: Button, Input, Modal 등 프로젝트의 어느 곳에서나 재사용될 수 있는 가장 기본적인 UI 단위입니다. 특정 비즈니스 로직을 포함해서는 안 됩니다.
* **domain**: 특정 도메인(예: user, team, evaluation)과 강하게 결합된 컴포넌트들을 그룹화합니다. 예를 들어 team 도메인과 관련된 TeamCard, TeamMember 등의 컴포넌트가 여기에 위치합니다.
* **layout**: 애플리케이션의 전체적인 구조를 형성하는 컴포넌트입니다. 페이지 콘텐츠를 감싸는 역할을 하며, 공통적인 UI 셸(Shell)을 제공합니다.

### API 디렉터리 구조

서버와의 통신 로직은 apis 디렉터리에서 중앙 관리하여 일관된 API 요청 패턴을 유지합니다.

|  |
| --- |
| src/apis ├── core // Axios 인스턴스 생성, 인터셉터 등 핵심 설정 ├── [domain] // 도메인별 API 호출 함수 └── index.ts // 각 도메인 API를 모아 내보내는(export) 파일 |

* **core**: Axios 인스턴스를 생성하고, baseURL이나 headers와 같은 기본 설정을 정의합니다. 또한, 요청/응답 인터셉터를 두어 인증 토큰 처리나 공통 에러 처리를 수행합니다.
* **[domain]**: 도메인별로 API 요청 함수를 분리하여 관리합니다. 예를 들어 userAPI.ts 파일에는 사용자 정보 조회, 수정 등과 관련된 함수들이 포함됩니다.

## 상태 관리 표준 (Zustand)

### 

본 프로젝트는 Zustand를 사용하여 간결하고 효율적인 전역 상태 관리를 지향합니다. 상태 로직과 UI를 분리하여 코드의 예측 가능성을 높이고, 테스트 용이성을 확보하는 것을 목표로 합니다.

### Store의 기본 원칙

* **최소화 원칙**: 전역 상태는 여러 컴포넌트에서 **반드시 공유해야 하는 상태**에만 사용합니다. 특정 컴포넌트 내부에서만 사용되는 상태는 useState나 useReducer를 사용하여 지역 상태로 관리합니다.
* **단일 책임 원칙**: 하나의 Store는 **하나의 도메인**과 관련된 상태와 액션만을 관리합니다. 예를 들어, userStore는 사용자 정보, 로그인 상태 등과 관련된 것만 책임집니다.
* **불변성 유지**: 상태(State)를 업데이트할 때는 반드시 immer를 사용하거나, 전개 연산자(...)를 활용하여 기존 상태를 직접 수정하지 않고 새로운 객체를 반환해야 합니다.

### Store 구조

모든 Store 파일은 src/store 디렉터리 내에 위치하며, 도메인별로 파일을 분리하여 관리합니다. 각 Store는 **상태(State), 액션(Actions), 계산된 상태(Computed State)** 세 부분으로 명확하게 구분하여 정의합니다.

|  |
| --- |
| // src/store/userStore.ts import { create } from 'zustand'; import { immer } from 'zustand/middleware/immer';  // 1. 상태(State)와 액션(Actions)의 타입을 먼저 정의합니다. interface UserState {  user: User | null;  isLoggedIn: boolean;  actions: {  setUser: (user: User) => void;  clearUser: () => void;  }; }  // 2. create 함수를 사용하여 Store를 생성합니다. const useUserStore = create(  immer<UserState>((set) => ({  // 2.1. 초기 상태 (Initial State)  user: null,  isLoggedIn: false,    // 2.2. 상태를 변경하는 액션 (Actions)  actions: {  setUser: (user) =>  set((state) => {  state.user = user;  state.isLoggedIn = true;  }),  clearUser: () =>  set((state) => {  state.user = null;  state.isLoggedIn = false;  }),  },  })) );  // 3. 커스텀 훅을 만들어 UI 컴포넌트에서의 사용을 단순화합니다. export const useUser = () => useUserStore((state) => state.user); export const useIsLoggedIn = () => useUserStore((state) => state.isLoggedIn); export const useUserActions = () => useUserStore((state) => state.actions); |

* **State**: 애플리케이션의 원본 데이터 소스입니다. user, isLoggedIn과 같이 순수한 데이터 객체로 구성됩니다.
* **Actions**: set 함수를 호출하여 상태를 변경하는 함수들의 집합입니다. 모든 상태 변경은 액션을 통해서만 이루어져야 합니다. 비동기 로직(예: API 호출) 또한 액션 내에서 처리합니다.
* **Computed State (커스텀 훅)**: 상태를 기반으로 파생된 값을 계산하는 부분입니다. 컴포넌트에서는 useUser와 같은 커스텀 훅을 사용하여 필요한 데이터만 선택(select)하여 렌더링 최적화를 도모합니다. actions는 별도의 훅(useUserActions)으로 분리하여, 액션만 필요한 컴포넌트의 불필요한 리렌더링을 방지합니다.

### Store 사용 규칙

* **컴포넌트 내 사용법**: 컴포넌트에서는 위에서 정의한 커스텀 훅을 사용하여 상태를 구독하고 액션을 호출합니다.

|  |
| --- |
| // 예시: UserProfile.tsx import { useUser, useUserActions } from '@/store/userStore';  function UserProfile() {  const user = useUser();  const { setUser } = useUserActions();   // ... } |

* **액션과 상태의 분리**: 상태를 변경할 필요가 없는 컴포넌트에서는 useUserActions 훅만 사용하여 불필요한 렌더링을 방지합니다. 반대로, 상태를 읽기만 하는 컴포넌트에서는 useUser 훅만 사용합니다.

## 컴포넌트 개발 표준 (React)

### 

본 프로젝트는 일관되고 예측 가능한 UI 개발을 위해 컴포넌트 기반 방법론을 따릅니다. 컴포넌트는 **재사용성, 명확성, 독립성**을 핵심 가치로 삼으며, 이를 통해 유지보수 비용을 최소화하는 것을 목표로 합니다.

### 컴포넌트 작성 원칙

* **단일 책임 원칙 (Single Responsibility Principle)**: 하나의 컴포넌트는 하나의 기능 또는 UI 조각에 대한 책임만 가집니다. 만약 컴포넌트가 너무 많은 역할을 수행한다면, 더 작은 여러 개의 컴포넌트로 분리해야 합니다.
* **표현과 로직의 분리 (Presentational and Container)**: 가능하면 UI를 그리는 역할(Presentational)과 데이터를 불러오고 상태를 처리하는 로직(Container)을 분리합니다. 로직은 페이지 컴포넌트나 커스텀 훅(use...)으로 위임하고, 하위 컴포넌트는 전달받은 props를 기반으로 UI를 렌더링하는 데 집중합니다.
* **Props의 불변성**: 컴포넌트는 전달받은 props를 직접 수정해서는 안 됩니다. props는 읽기 전용(Read-only)으로 취급하며, props의 변경이 필요할 경우 부모 컴포넌트로부터 새로운 props를 받거나 상태 변경 함수를 props로 전달받아 실행해야 합니다.

### 명명 규칙

* **컴포넌트 파일명**: PascalCase를 사용하여 작성하며, 확장자는 .tsx를 사용합니다. (예: TeamMemberList.tsx)
* **컴포넌트 함수명**: 파일명과 동일하게 PascalCase를 사용합니다. (예: function TeamMemberList() { ... })
* **이벤트 핸들러 함수명**: handle 접두사 뒤에 이벤트 이름을 붙여 camelCase로 작성합니다. (예: handleClick, handleInputChange, handleSubmit)

### 컴포넌트 파일 구조

컴포넌트 파일 내부는 가독성과 일관성을 위해 다음 순서를 따릅니다.

1. **Imports**: 라이브러리 및 다른 모듈을 가져옵니다.
2. **Type/Interface Definitions**: 컴포넌트에서 사용할 Props나 내부 타입들을 정의합니다.
3. **Component Function**: 컴포넌트의 메인 로직과 JSX를 포함합니다.
4. **Export**: 컴포넌트를 외부에서 사용할 수 있도록 export 합니다.

|  |
| --- |
| // 1. Imports import { useState } from 'react'; import { useUserActions } from '@/store/userStore'; import { Button } from '@/components/common';  // 2. Type/Interface Definitions interface UserProfileProps {  userId: string; }  // 3. Component Function function UserProfile({ userId }: UserProfileProps) {  // Hooks  const [isLoading, setIsLoading] = useState(false);  const { fetchUser } = useUserActions();   // Event Handlers  const handleRefreshClick = () => {  setIsLoading(true);  fetchUser(userId).finally(() => setIsLoading(false));  };    // Effect Hooks  useEffect(() => {  fetchUser(userId);  }, [userId, fetchUser]);    // Return (JSX)  return (  <**div**>  {/\* ... \*/}  <**Button** onClick={handleRefreshClick} disabled={isLoading}>  Refresh  </**Button**>  </**div**>  ); }  // 4. Export export default UserProfile; |

### 훅 사용 규칙

* **커스텀 훅 활용**: 두 개 이상의 컴포넌트에서 반복되는 로직(예: API 호출, 폼 상태 관리, 윈도우 이벤트 처리 등)은 반드시 커스텀 훅으로 분리하여 src/hooks 디렉터리에서 관리합니다. 커스텀 훅의 이름은 use 접두사를 사용해야 합니다.
* **useEffect 의존성 배열**: useEffect의 의존성 배열(deps)은 반드시 명시해야 합니다. 배열에는 해당 Effect가 의존하는 모든 값을 포함시켜, 의도치 않은 동작이나 불필요한 재실행을 방지해야 합니다. eslint-plugin-react-hooks 규칙을 준수하여 의존성 배열을 관리합니다.

## API 연동 표준 (Axios)

본 프로젝트는 서버와의 데이터 통신을 위해 Axios 라이브러리를 사용합니다. API 요청 로직을 중앙에서 관리하여 **코드의 일관성을 유지**하고, **유지보수를 용이**하게 하며, **공통 로직을 효율적으로 처리**하는 것을 목표로 합니다.

### Axios 인스턴스 관리

* **중앙화된 인스턴스 생성**: 모든 API 요청은 src/apis/core/instance.ts에 생성된 **단일 Axios 인스턴스**를 통해 이루어져야 합니다. 이를 통해 baseURL, timeout, 기본 headers 등 공통 설정을 한 곳에서 관리할 수 있습니다.

|  |
| --- |
| // src/apis/core/instance.ts import axios from 'axios';  const instance = axios.create({  baseURL: process.env.REACT\_APP\_API\_URL, // .env 파일에서 API 기본 URL을 가져옵니다.  timeout: 5000, // 5초 이상 응답이 없으면 타임아웃 처리합니다.  headers: {  'Content-Type': 'application/json',  }, });  export default instance; |

* **인터셉터(Interceptors) 활용**: 요청(Request) 및 응답(Response) 인터셉터를 사용하여 모든 API 통신에 필요한 공통 로직을 처리합니다.
  + **요청 인터셉터**: API 요청이 서버로 전송되기 전에 실행됩니다. 주로 인증 토큰(e.g., JWT)을 Authorization 헤더에 추가하는 역할을 합니다.
  + **응답 인터셉터**: 서버로부터 응답을 받은 후 실행됩니다. 특정 에러 코드에 따른 공통 에러 처리(e.g., 401 Unauthorized 시 로그인 페이지로 리다이렉트)나 데이터 형식 변환 등의 역할을 수행합니다.

### API 함수 구조

* **도메인 기반 분리**: API 요청 함수들은 src/apis 디렉터리 아래에 **도메인별**로 파일을 분리하여 관리합니다. (예: userAPI.ts, teamAPI.ts)
* **명확한 함수 시그니처**: 각 API 함수는 **비동기 함수(async/await)** 로 작성하며, 매개변수와 반환 값에 대한 타입을 명확하게 정의해야 합니다. 이를 통해 API 사용 시 예측 가능성과 안정성을 높입니다.

|  |
| --- |
| // src/apis/userAPI.ts import instance from './core/instance'; import { User, UserUpdateRequest } from '@/types/user'; // 타입 정의 파일에서 타입을 가져옵니다.  /\*\*  \* 사용자 정보를 조회하는 API  \* @param userId 조회할 사용자의 ID  \* @returns 사용자 정보 객체  \*/ export const getUser = async (userId: string): Promise<User> => {  const { data } = await instance.get<User>(`/users/${userId}`);  return data; };  /\*\*  \* 사용자 정보를 수정하는 API  \* @param userId 수정할 사용자의 ID  \* @param userData 수정할 사용자 정보  \* @returns 수정된 사용자 정보 객체  \*/ export const updateUser = async (userId: string, userData: UserUpdateRequest): Promise<User> => {  const { data } = await instance.patch<User>(`/users/${userId}`, userData);  return data; }; |

### 에러 처리 표준

* **컴포넌트 레벨 에러 처리**: API 호출은 try...catch 블록으로 감싸서 처리하는 것을 원칙으로 합니다. UI와 직접적으로 관련된 에러(예: 폼 유효성 검사 실패)는 해당 컴포넌트나 페이지에서 직접 처리하여 사용자에게 피드백(e.g., 토스트 메시지, 에러 문구)을 제공해야 합니다.
* **전역 에러 처리**: 인증 실패(401), 서버 내부 오류(500) 등 애플리케이션 전반에 영향을 미치는 공통 에러는 **응답 인터셉터**에서 일괄적으로 처리합니다.

## 스타일링 표준 (Tailwind CSS)

### 

본 프로젝트는 Tailwind CSS를 사용하여 **빠르고 일관된** UI 개발을 지향합니다. 유틸리티 우선(Utility-First) 접근 방식을 통해 디자인 시스템을 코드에 직접 적용하고, 스타일링의 재사용성과 유지보수성을 높이는 것을 목표로 합니다.

### 기본 원칙 (Utility-First)

* **유틸리티 클래스 직접 사용**: 스타일링은 미리 정의된 유틸리티 클래스(p-4, flex, text-blue-500 등)를 JSX에 직접 적용하는 것을 원칙으로 합니다. 별도의 CSS 파일을 작성하는 것을 최소화하고, 마크업과 스타일을 한 곳에서 관리하여 응집도를 높입니다.
* **추상화 최소화**: 꼭 필요한 경우가 아니라면, 여러 유틸리티 클래스를 하나의 커스텀 클래스로 묶는(@apply) 것을 지양합니다. 이는 유틸리티 우선의 장점을 해치고 불필요한 추상화 계층을 만들 수 있습니다. 재사용이 필요한 UI는 스타일이 아닌 **컴포넌트 단위**로 분리하여 해결합니다.

### 설정 및 테마 관리 ([tailwind.config.js](http://tailwind.config.js))

* **중앙화된 테마 정의**: 프로젝트에서 사용될 모든 디자인 토큰(색상, 간격, 폰트 사이즈, 화면 분기점 등)은 tailwind.config.js 파일의 theme 객체 내에서 중앙 관리되어야 합니다. 디자인 시스템과의 일관성을 유지하기 위해 임의의 값을 직접 사용(e.g., top-[13px])하는 것을 금지합니다.

|  |
| --- |
| // tailwind.config.js module.exports = {  theme: {  extend: {  colors: {  primary: '#007BFF',  secondary: '#6C757D',  // ... 프로젝트 맞춤 색상 정의  },  spacing: {  '128': '32rem',  // ... 프로젝트 맞춤 간격 정의  },  },  },  // ... }; |

* **테마 값 사용**: 설정 파일에 정의된 테마 값은 text-primary, p-128과 같이 클래스 이름으로 직접 사용합니다.

### 커스텀 컴포넌트 클래스 (@apply)

* **제한적 사용**: @apply 지시어는 버튼, 카드, 폼 입력 필드와 같이 **반복적으로 사용되며 일관된 스타일이 반드시 필요한** UI 패턴에 한해서만 제한적으로 사용합니다. 이는 여러 곳에서 동일한 유틸리티 조합이 반복되는 것을 방지하기 위함입니다.
* **위치**: @apply를 사용한 커스텀 클래스는 src/styles/index.css 파일 내에서 관리합니다.

|  |
| --- |
| /\* src/styles/index.css \*/ @tailwind base; @tailwind components; @tailwind utilities;  @layer components {  .btn-primary {  @apply py-2 px-4 bg-primary text-white font-semibold rounded-lg shadow-md hover:bg-blue-700;  } } |

### 반응형 디자인 (Responsive Design)

* **모바일 우선 접근**: 모든 스타일은 모바일 화면을 기본으로 작성하며, sm, md, lg, xl과 같은 반응형 분기점(breakpoint)을 사용하여 더 큰 화면에 대한 스타일을 점진적으로 추가합니다.

|  |
| --- |
| <**div** className="w-full md:w-1/2 lg:w-1/3">  {/\* 모바일에서는 너비 100%, **md** 이상에서는 50%, lg 이상에서는 33.3% \*/} </**div**> |

### 동적 클래스 사용 시 주의사항

* **클래스 이름 전체 포함**: Tailwind CSS의 JIT 컴파일러는 소스 코드를 스캔하여 사용된 클래스를 감지합니다. 따라서 클래스 이름을 동적으로 조합해서는 안 됩니다.

|  |
| --- |
| // ❌ 잘못된 사용법 (작동하지 않음) const color = 'blue'; <div className={`text-${color}-500`} />  // ✅ 올바른 사용법 (전체 클래스 이름을 코드에 명시) const colorClasses = {  blue: 'text-blue-500',  red: 'text-red-500', }; <div className={colorClasses[color]} /> |

## 타입스크립트 (TypeScript) 사용 표준

### 

본 프로젝트는 코드의 **안정성과 예측 가능성**을 높이기 위해 TypeScript를 적극적으로 사용합니다. 컴파일 시점에 타입을 검사하여 런타임 에러를 사전에 방지하고, 명시적인 타입을 통해 코드의 가독성과 유지보수성을 향상시키는 것을 목표로 합니다.

### 기본 원칙

* **타입 안정성 극대화 (Type Safety)**: 모든 변수, 함수 매개변수, 반환 값에는 명시적으로 타입을 정의하는 것을 원칙으로 합니다. 타입 추론이 명확한 경우를 제외하고는, 의도를 명확히 하기 위해 타입을 선언합니다.
* **any 타입 사용 지양**: any 타입은 TypeScript의 정적 타입 검사를 무력화시키므로 사용을 **엄격히 금지**합니다. 타입을 알 수 없는 경우에는 any 대신 unknown을 사용한 뒤, 타입 가드(Type Guard)나 타입 단언(Type Assertion)을 통해 타입을 좁혀서 사용해야 합니다.
* **가독성과 명확성**: 타입 이름은 해당 데이터가 무엇을 나타내는지 명확하게 알 수 있도록 PascalCase로 작성합니다. (예: TeamMember, EvaluationSheet)

### 타입 정의 및 관리

* **interface vs type**:
  + **interface**: 객체의 구조를 정의할 때 사용을 권장합니다. extends를 통한 확장이 명료하고, 선언 병합(declaration merging)이 가능하여 유연한 구조를 만들 수 있습니다. API 응답 데이터, 객체 형태의 props 등을 정의하는 데 주로 사용합니다.
  + **type**: 원시 타입(Primitive), 유니언(|), 인터섹션(&), 튜플 등 더 복잡하고 다양한 형태의 타입을 정의할 때 사용합니다.
* **중앙화된 타입 관리**: 프로젝트 전반에서 재사용되는 타입은 src/types 디렉터리 내에 **도메인별**로 파일을 분리하여 관리합니다. (예: user.ts, api.ts). 특정 컴포넌트에서만 사용되는 일회성 타입의 경우, 해당 컴포넌트 파일 상단에 정의할 수 있습니다.

|  |
| --- |
| // src/types/user.ts export interface User {  id: number;  name: string;  email: string;  role: 'ADMIN' | 'USER'; // 유니언 타입을 활용한 역할 정의 }  // API 요청 시 부분적 업데이트를 위한 타입 export type UserUpdateRequest = Partial<Omit<User, 'id' | 'email'>>; |

### API 데이터 타입 정의

* **요청/응답 타입 필수**: 모든 API 연동 함수는 요청(Request) 데이터와 응답(Response) 데이터에 대한 명시적인 타입을 가져야 합니다. 이는 백엔드와의 데이터 계약을 명확히 하고, 데이터 불일치로 인한 버그를 예방합니다.
* **백엔드와 타입 공유**: 가능한 경우, 백엔드에서 사용하는 DTO(Data Transfer Object)와 프론트엔드의 타입을 일치시켜 데이터 구조의 일관성을 유지합니다.

### 유틸리티 타입 활용

* TypeScript에서 제공하는 유틸리티 타입을 적극적으로 활용하여 타입 정의의 중복을 줄이고 재사용성을 높입니다.
  + **Partial<T>**: T의 모든 프로퍼티를 선택적으로 만듭니다. (수정 폼 등에서 활용)
  + **Pick<T, K>**: T에서 K 프로퍼티만 선택하여 새로운 타입을 만듭니다.
  + **Omit<T, K>**: T에서 K 프로퍼티를 제외하고 새로운 타입을 만듭니다.
  + **Readonly<T>**: T의 모든 프로퍼티를 읽기 전용으로 만듭니다.

# 백엔드(BE) 개발 표준

## 

## 프로젝트 구조 (패키지 구조)

### 

본 프로젝트는 도메인 기반의 계층형 아키텍처(Layered Architecture)를 따릅니다. 패키지 구조는 기능적 관심사에 따라 모듈화하여 응집도를 높이고, 다른 모듈과의 결합도를 낮추는 것을 목표로 합니다.

### 최상위 패키지 구조

최상위 패키지는 skala.skoro로 하며, 그 아래에 domain과 **global** 두 개의 핵심 패키지를 둡니다.

|  |
| --- |
| skala └── skoro  ├── domain // 비즈니스 로직을 포함하는 핵심 도메인 영역  └── global // 프로젝트 전역에서 사용되는 공통 기능 영역 |

* **domain**: 애플리케이션의 핵심 비즈니스 로직이 위치하는 곳입니다. 각 하위 패키지는 독립적인 도메인(예: employee, kpi, evaluation)을 나타냅니다.
* **global**: 특정 도메인에 종속되지 않고 프로젝트 전반에서 사용되는 공통 기능(예: 보안 설정, 예외 처리, 전역 설정)을 관리합니다.

### 도메인 패키지 (domain) 구조

각 도메인 패키지는 기능에 따라 다음과 같은 표준 하위 패키지 구조를 가집니다. 예를 들어, kpi 도메인은 아래와 같은 구조를 따릅니다.

|  |
| --- |
| skala.skoro.domain └── kpi  ├── controller // API의 엔드포인트(Endpoint)를 정의하는 클래스  ├── service // 비즈니스 로직을 처리하는 서비스 클래스  ├── repository // 데이터베이스에 접근하는 인터페이스 (Spring Data JPA)  ├── entity // 데이터베이스 테이블과 매핑되는 JPA 엔티티 클래스  └── dto // 계층 간 데이터 전송을 위한 객체 (Data Transfer Object) |

* **controller**: HTTP 요청을 수신하고, 적절한 서비스 메서드를 호출한 뒤, 그 결과를 HTTP 응답으로 반환하는 역할만 수행합니다. **비즈니스 로직을 포함해서는 안 됩니다.**
* **service**: 핵심 비즈니스 로직을 구현합니다. 여러 Repository를 조합하여 하나의 트랜잭션 안에서 데이터를 처리하며, 필요한 경우 다른 도메인의 Service를 호출할 수 있습니다.
* **repository**: Spring Data JPA의 JpaRepository를 상속받아 데이터베이스와의 CRUD(Create, Read, Update, Delete) 작업을 담당합니다. 복잡한 쿼리는 @Query 어노테이션을 사용하여 정의합니다.
* **entity**: 데이터베이스의 테이블을 나타내는 순수한 객체입니다. **절대로 요청(Request)이나 응답(Response) DTO로 사용해서는 안 됩니다.**
* **dto**: 각 계층(Controller, Service, Repository) 간에 데이터를 전달하기 위한 객체입니다. Request, Response와 같이 목적에 따라 명확하게 이름을 구분하여 사용합니다.

### 글로벌 패키지 (global) 구조

프로젝트 전역에서 필요한 공통 기능들은 global 패키지 아래에 기능별로 묶어 관리합니다.

|  |
| --- |
| skala.skoro.global ├── auth // JWT 토큰 처리, UserDetails 등 인증/인가 관련 로직 ├── config // SecurityConfig, SwaggerConfig, RedisConfig 등 각종 설정 클래스 ├── exception // CustomException,GlobalExceptionHandler 등 전역 예외 처리 └── util // 날짜 처리, 정규식 등 재사용 가능한 유틸리티 클래스 (필요시 생성) |

## Controller 개발 표준

### 

Controller는 클라이언트의 HTTP 요청을 가장 먼저 수신하는 계층입니다. Controller의 역할은 요청을 적절한 Service 계층으로 전달하고, 그 결과를 클라이언트에게 응답하는 것에만 집중해야 합니다.

### 

### 기본 구조 및 책임

**@RestController: 모든 Controller 클래스에는 @RestController 어노테이션을 사용하여 각 메서드의 반환 값이 HTTP 응답 본문(Response Body)에 직접 작성되도록 합니다.**

* **@RequestMapping**: 클래스 레벨에는 @RequestMapping을 사용하여 해당 Controller가 처리할 공통 URL 접두사(prefix)를 지정합니다.

|  |
| --- |
| @Tag(name = "팀 KPI") @RestController @RequestMapping("/team-kpis") // 공통 URL 접두사 @RequiredArgsConstructor public class TeamKpiController {  // ... } |

* **의존성 주입**: private final 키워드와 @RequiredArgsConstructor 롬복(Lombok) 어노테이션을 사용하여 생성자 주입(Constructor Injection) 방식으로 Service 객체를 주입받습니다.
* **역할 제한**: Controller는 **요청/응답 처리, 데이터 유효성 검사, 인증/인가 확인** 외의 비즈니스 로직을 포함해서는 안 됩니다. 모든 비즈니스 로직은 **Service 계층에 위임**해야 합니다.

### 

### API 엔드포인트 개발 표준

**메서드 매핑: HTTP Method(GET, POST, PUT, DELETE 등)에 따라 @GetMapping, @PostMapping, @PutMapping 등의 어노테이션을 명확하게 사용합니다.**

* **URL 네이밍**: URL은 리소스를 표현하는 명사 형태로 작성하며, 동사 사용을 지양합니다. (예: /users (O), /getUsers (X))
* **파라미터 처리**:
  + **경로 변수**: @PathVariable을 사용하여 URL 경로의 일부를 변수로 받습니다. (예: /users/{userId})
  + **쿼리 파라미터**: @RequestParam을 사용하여 URL의 쿼리 스트링을 받습니다.
  + **요청 본문**: @RequestBody를 사용하여 POST, PUT 요청의 본문(JSON 등)을 DTO 객체로 받습니다.
* **인증 정보**: 현재 로그인한 사용자 정보가 필요한 경우, @AuthenticationPrincipal 어노테이션을 사용하여 CustomUserDetails 객체를 직접 주입받습니다.

### 응답(Response) 처리 표준

**ResponseEntity 활용: 모든 응답은 ResponseEntity 객체로 감싸서 반환하는 것을 원칙으로 합니다. 이를 통해 HTTP 상태 코드, 헤더, 본문을 명확하게 제어할 수 있습니다.**

* **성공 응답**:
  + 조회(GET) 성공 시: ResponseEntity.ok(body)를 사용하여 상태 코드 200 OK와 함께 응답 본문을 반환합니다.
  + 생성(POST) 성공 시: ResponseEntity.status(HttpStatus.CREATED).body(body) 와 같이 상태 코드 201 Created를 반환하는 것을 권장합니다.
  + 내용 없는 응답: ResponseEntity.ok().build() 또는 ResponseEntity.noContent().build()를 사용하여 본문 없이 상태 코드만 반환합니다.
* **예외 처리**: Controller에서 try-catch 블록을 사용하는 대신, 예외는 @RestControllerAdvice를 이용한 전역 예외 처리기(GlobalExceptionHandler)에서 처리합니다. 서비스 계층에서 발생한 CustomException은 자동으로 적절한 HTTP 상태 코드와 오류 메시지로 변환됩니다.

### API 문서화

**Swagger 사용: 모든 Controller 클래스와 API 메서드에는 Swagger 어노테이션을 사용하여 API 명세를 명확하게 문서화합니다.**

* + **@Tag**: Controller 클래스에 API 그룹의 이름을 지정합니다.
  + **@Operation**: 각 API 메서드에 대한 상세한 설명을 작성합니다.

|  |
| --- |
| @Tag(name = "동료 평가") // API 그룹 정의 @RestController @RequestMapping("/peer-evaluation") @RequiredArgsConstructor public class PeerEvaluationController {   private final PeerEvaluationService peerEvaluationService;   @Operation(summary = "동료 평가 리스트 조회") // API 기능 요약  @GetMapping  public List<PeerEvaluationStatusResponse> getPeerEvaluationStatusList(@RequestParam String empNo, @RequestParam Long periodId) {  return peerEvaluationService.getPeerEvaluationStatusList(empNo, periodId);  }  // ... } |

## Service 개발 표준

### 

Service 계층은 애플리케이션의 **핵심 비즈니스 로직을 담당**하는 가장 중요한 부분입니다. Controller로부터 전달받은 데이터를 가공하고, Repository를 통해 데이터베이스와 상호작용하며, 비즈니스 규칙에 맞는 결과를 만들어내는 책임을 가집니다.

### 기본 구조 및 책임

* **@Service**: 모든 Service 클래스에는 @Service 어노테이션을 사용하여 해당 클래스가 비즈니스 로직을 처리하는 서비스 컴포넌트임을 명시합니다.
* **@Transactional**: 클래스 레벨에 @Transactional을 선언하여 해당 클래스의 모든 public 메서드가 하나의 트랜잭션 단위로 동작하도록 하는 것을 원칙으로 합니다. 이를 통해 데이터의 일관성과 무결성을 보장합니다.
  + **조회 전용 메서드**: 데이터 변경이 없는 조회(Read) 기능의 메서드에는 @Transactional(readOnly = true)를 사용하여 성능을 최적화합니다.
* **의존성 주입**: Controller와 동일하게 private final과 @RequiredArgsConstructor를 사용하여 다른 Service나 Repository를 주입받습니다.

|  |
| --- |
| @Service @Transactional // 클래스 내 모든 public 메서드에 트랜잭션 적용 @RequiredArgsConstructor public class TeamEvaluationService {   private final EmployeeService employeeService;  private final TeamEvaluationRepository teamEvaluationRepository;  // ...   @Transactional(readOnly = true) // 데이터 변경이 없는 조회 메서드에 최적화  public List<TeamEvaluationDetailResponse> findTeamEvaluationsByYear(String empNo) {  // ...  } } |

### 비즈니스 로직 구현

* **단일 책임 원칙 (SRP)**: 메서드는 가능한 한 하나의 명확한 기능만 수행하도록 작성합니다. 예를 들어, createPeriod 메서드는 기간 생성 로직만 담당하고, 생성된 기간에 대한 팀 평가를 만드는 것은 teamEvaluationService.createAllTeamEvaluations와 같이 다른 서비스에 위임합니다.
* **데이터 처리**: 데이터베이스에서 조회한 Entity 객체는 Service 계층 안에서만 사용되는 것을 원칙으로 합니다. Controller로 반환할 때는 반드시 DTO로 변환하여 전달해야 합니다. 이는 Entity의 영속성 상태와 관계없이 계층 간의 결합도를 낮추기 위함입니다.
* **도메인 간 상호작용**: 다른 도메인의 기능이 필요할 경우, 해당 도메인의 Repository를 직접 호출하는 대신 **Service를 호출하는 것을 원칙**으로 합니다. 이는 다른 도메인의 비즈니스 로직과 트랜잭션을 존중하기 위함입니다.
  + **예시**: PeriodService는 팀 평가를 생성하기 위해 TeamEvaluationRepository를 직접 주입받는 대신, TeamEvaluationService를 주입받아 해당 서비스를 호출합니다.

### 예외 처리

* **CustomException 활용**: 비즈니스 규칙에 어긋나는 예외 상황이 발생했을 경우(예: 존재하지 않는 데이터 조회), GlobalExceptionHandler가 처리할 수 있도록 미리 정의된 CustomException을 발생시킵니다.
* **오류 코드 사용**: CustomException을 발생시킬 때는, ErrorCode 열거형(enum)에 정의된 명확한 오류 코드를 사용합니다.

|  |
| --- |
| // TeamEvaluationService.java public TeamEvaluation findTeamEvaluationByEmployeeAndPeriodId(Employee employee, Long periodId) {  return teamEvaluationRepository.findByTeamAndPeriodId(employee.getTeam(), periodId)  .orElseThrow(() -> new CustomException(TEAM\_EVALUATION\_DOES\_NOT\_EXIST)); // 명시적인 예외 발생 } |

### 명명 규칙

* **조회 메서드**: find... 또는 get...으로 시작하는 이름을 사용합니다. (예: findTeamEvaluationsByYear)
* **생성 메서드**: create... 또는 save...로 시작하는 이름을 사용합니다. (예: createPeriod)
* **수정 메서드**: update...로 시작하는 이름을 사용합니다. (예: updatePeriod)
* **삭제 메서드**: delete... 또는 remove...로 시작하는 이름을 사용합니다.

## 데이터 접근 계층 표준 (Spring Data JPA)

### 

데이터 접근 계층은 서비스 계층과 데이터베이스 사이의 상호작용을 책임지며, Spring Data JPA를 사용하여 구현합니다. 이 계층은 **Entity, Repository, DTO** 세 가지 주요 구성 요소에 대한 명확한 규칙을 가집니다.

### Entity 작성 규칙

Entity는 데이터베이스 테이블과 직접 매핑되는 핵심 도메인 객체입니다. 데이터베이스의 스키마를 표현하는 것 외의 비즈니스 로직은 포함하지 않는 것을 원칙으로 합니다.

* **@Entity**: 모든 엔티티 클래스에는 @Entity 어노테이션을 붙여 JPA가 관리하는 객체임을 명시합니다.
* **@Table**: 클래스 이름과 테이블 이름이 다를 경우, @Table(name = "...")을 사용하여 매핑될 테이블 이름을 명시적으로 지정합니다.
* **기본 생성자**: @NoArgsConstructor(access = AccessLevel.PROTECTED)를 사용하여, 외부에서 무분별한 객체 생성을 막는 protected 접근 제어자의 기본 생성자를 추가합니다.
* **ID 생성 전략**: Primary Key에는 @Id를 붙이고, 데이터베이스의 Auto Increment 전략을 따르기 위해 @GeneratedValue(strategy = GenerationType.IDENTITY)를 사용합니다.
* **연관 관계 매핑**:
  + 다른 엔티티와의 연관 관계는 @ManyToOne, @OneToMany 등을 사용하여 설정합니다.
  + 성능 최적화를 위해 모든 연관 관계의 fetch 타입은 FetchType.LAZY (지연 로딩)를 기본으로 사용합니다. N+1 문제를 방지하기 위해, 여러 엔티티를 함께 조회해야 할 경우 Repository에서 fetch join을 사용합니다.
* **상속**: 생성일, 수정일과 같은 공통 필드는 @MappedSuperclass로 정의된 BaseEntity를 상속받아 관리합니다.

|  |
| --- |
| @Getter @NoArgsConstructor(access = AccessLevel.PROTECTED) @Entity @Table(name = "tasks") public class Task extends BaseEntity {  @Id  @GeneratedValue(strategy = GenerationType.IDENTITY)  @Column(name = "task\_id")  private Long id;   // ... 다른 컬럼들 ...   @ManyToOne(fetch = FetchType.LAZY) // 지연 로딩 사용  @JoinColumn(name = "team\_kpi\_id")  private TeamKpi teamKpi;   @ManyToOne(fetch = FetchType.LAZY) // 지연 로딩 사용  @JoinColumn(name = "emp\_no")  private Employee employee; } |

#### 

### Repository 작성 규칙

Repository는 데이터베이스에 대한 접근을 추상화하는 인터페이스입니다.

* **JpaRepository 상속**: 모든 Repository 인터페이스는 JpaRepository<EntityType, IdType>를 상속받아 기본적인 CRUD 메서드를 자동으로 제공받습니다.
* **쿼리 메서드**: 간단한 조회 쿼리는 Spring Data JPA의 쿼리 메서드(Query Method) 규칙에 따라 메서드 이름을 작성하여 구현합니다. (예: findByTeamAndYearOrderByWeightDesc)
* **@Query 어노테이션**: 복잡한 조회나 Join이 필요한 쿼리는 JPQL(Java Persistence Query Language)을 사용하여 @Query 어노테이션으로 직접 작성합니다. 이는 쿼리의 가독성을 높이고, 컴파일 시점에 쿼리 문법 오류를 잡을 수 있게 해줍니다.

|  |
| --- |
| public interface TaskRepository extends JpaRepository<Task, Long> {  // 쿼리 메서드 예시  List<Task> findByTeamKpi(TeamKpi teamKpi);   // @Query 예시  @Query("""  SELECT DISTINCT t.employee  FROM Task t  WHERE t.teamKpi.id = :teamKpiId  """)  List<Employee> findEmployeesByTeamKpiId(@Param("teamKpiId") Long teamKpiId); } |

### DTO 정의 표준

DTO(Data Transfer Object)는 계층 간 데이터 전송을 위해 사용하는 객체입니다. **절대로 Entity를 Controller나 View 계층으로 직접 노출해서는 안 되며**, 항상 DTO로 변환하여 전달해야 합니다.

* **목적에 따른 분리**: DTO는 목적에 따라 명확히 구분하여 작성합니다.
  + **...Request**: 클라이언트의 요청 데이터를 담는 DTO (예: LoginRequest)
  + **...Response**: 클라이언트에게 보낼 응답 데이터를 담는 DTO (예: TeamKpiDetailResponse)
* **변환 책임**:
  + **Entity → DTO**: 주로 DTO 클래스 내부에 from(Entity) 또는 of(Entity, ...)와 같은 정적 팩토리 메서드(static factory method)를 두어 변환을 책임집니다. 이는 변환 로직의 응집도를 높여줍니다.
  + **DTO → Entity**: DTO에서 Entity를 생성해야 할 경우, Entity 클래스 내부에 정적 팩토리 메서드 from(DTO) 또는 of(DTO)를 만듭니다.

|  |
| --- |
| // EmployeeSummaryResponse DTO 예시 @Getter @Builder @AllArgsConstructor public class EmployeeSummaryResponse {  private String empNo;  private String empName;  private String profileImage;   // Entity를 DTO로 변환하는 정적 팩토리 메서드  public static EmployeeSummaryResponse from(Employee employee) {  return EmployeeSummaryResponse.builder()  .empNo(employee.getEmpNo())  .empName(employee.getEmpName())  .profileImage(employee.getProfileImage())  .build();  } } |

## 인증/인가 표준 (Spring Security & JWT)

### 

애플리케이션의 보안은 Spring Security를 기반으로 하며, 인증 방식은 상태 비저장(Stateless)을 유지하기 위해 JWT(JSON Web Token)를 사용합니다.

### 기본 아키텍처

* **Spring Security**: 전체 보안 프레임워크의 기반으로, HTTP 요청에 대한 필터 체인, URL 기반 접근 제어, 메서드 레벨 보안을 관리합니다. 모든 설정은 SecurityConfig 클래스에서 중앙 관리합니다.
* **JWT**: 사용자의 인증 상태를 나타내는 토큰으로, **Access Token**과 **Refresh Token** 두 가지 종류를 사용합니다. 토큰의 생성, 유효성 검사, 정보 추출은 JwtProvider 클래스가 전담합니다.
* **Stateless 세션 정책**: 서버에 세션을 저장하지 않는 SessionCreationPolicy.STATELESS를 사용하여, 모든 요청은 헤더에 포함된 JWT를 통해 인증됩니다.

### 인증 (Authentication) 흐름

1. **로그인**: 클라이언트는 사번(empNo)과 비밀번호를 /auth/login 엔드포인트로 전송합니다.
2. **토큰 발급**: AuthService는 사용자 정보를 확인한 후, JwtProvider를 통해 accessToken과 refreshToken을 생성하여 클라이언트에게 반환합니다.
3. **RefreshToken 저장**: 발급된 refreshToken은 서버의 Redis에 저장하여 토큰 재발급 요청 시 검증에 사용됩니다.
4. **요청 인증**: 이후 클라이언트의 모든 요청은 HTTP 헤더에 Authorization: Bearer <accessToken> 형식으로 토큰을 포함해야 합니다.
5. **필터 기반 검증**: JwtAuthenticationFilter는 매 요청을 가로채 토큰의 유효성을 검사하고, 유효한 경우 SecurityContextHolder에 인증 정보를 설정하여 해당 요청이 인증된 사용자의 요청임을 시스템에 알립니다.

### 인가 (Authorization) 방식

사용자의 역할(Role)에 따라 접근 권한을 제어하며, 두 가지 레벨에서 통제됩니다.

* **역할 계층 (Role Hierarchy)**: ADMIN > MANAGER > MEMBER 순서의 명확한 역할 계층을 정의합니다. 상위 역할은 하위 역할의 모든 권한을 가집니다. (예: ADMIN은 MANAGER와 MEMBER의 모든 API를 호출할 수 있습니다.)
* **메서드 기반 인가**: @EnableMethodSecurity를 활성화하고, Controller의 각 메서드에 @PreAuthorize("hasRole('ROLE\_NAME')") 어노테이션을 사용하여 세밀한 접근 제어를 구현합니다. 이는 특정 API에 특정 역할의 사용자만 접근할 수 있도록 보장하는 가장 명확하고 강력한 방법입니다.

### 주요 컴포넌트 및 책임

* **JwtProvider**: 토큰 생성 및 검증 로직을 완전히 캡슐화합니다. 토큰에 관련된 모든 명세(비밀 키, 유효 시간 등)는 이 클래스에서 관리됩니다.
* **CustomUserDetailsService**: 사번(empNo)을 사용하여 데이터베이스에서 Employee 정보를 조회하고, Spring Security가 이해할 수 있는 UserDetails 객체로 변환하는 역할을 합니다.
* **Role Enum**: ADMIN, MANAGER, MEMBER와 같이 시스템의 모든 사용자 역할을 열거형으로 정의하여, 문자열 오타 등으로 인한 실수를 방지하고 코드의 명확성을 높입니다.

## 

## DTO 사용 표준

DTO(Data Transfer Object)는 각 계층 간, 특히 클라이언트와 애플리케이션 간의 데이터 전송을 위해 설계된 객체입니다. DTO를 올바르게 사용하는 것은 시스템의 보안과 유지보수성을 위해 매우 중요합니다.

### 핵심 원칙: Entity 보호

* **Entity 직접 사용 금지**: **어떤 경우에도 Controller에서 Entity 객체를 직접 반환하거나 요청 파라미터로 받아서는 안 됩니다.** 이는 데이터베이스 스키마와 내부 구조를 외부에 노출시켜 보안에 취약하고, 작은 변경이 클라이언트에 큰 영향을 미치는 강한 결합을 유발합니다.
* **변환 필수**: Service 계층은 Controller로 데이터를 반환하기 전에, 반드시 Entity를 **Response DTO로 변환**해야 합니다. 마찬가지로 Controller는 클라이언트의 요청을 **Request DTO로** 받아 Service에 전달해야 합니다.

### DTO 명명 및 작성 규칙

* **목적 기반 명명**: DTO 클래스의 이름은 그 목적을 명확히 나타내야 합니다.
  + **...Request**: 클라이언트에서 서버로 들어오는 요청 데이터를 담는 DTO (예: PeriodCreateRequest)
  + **...Response**: 서버가 클라이언트로 보내는 응답 데이터를 담는 DTO (예: TeamKpiDetailResponse)
* **불변성(Immutability)**: DTO는 데이터 전송 목적을 가지므로, 생성된 이후에는 상태가 변경되지 않는 **불변 객체**로 만드는 것을 권장합니다.
  + Java 14 이상에서는 record 타입을 사용하면 간결하게 불변 DTO를 만들 수 있습니다.
  + 현재 프로젝트에서는 private final 필드와 @AllArgsConstructor를 사용하여 불변성을 유지하고 있습니다. @Setter는 사용을 지양합니다.
* **유효성 검사**: Request DTO의 필드에는 jakarta.validation.constraints (예: @NotNull, @Size, @Email) 어노테이션을 사용하여 데이터 유효성 검사 규칙을 명시합니다. 이 검사는 Controller에서 @Valid 어노테이션을 통해 자동으로 수행됩니다.

### 변환 로직

* **Entity → DTO 변환**: DTO 클래스 내에 정적 팩토리 메서드(Static Factory Method)인 from() 또는 of()를 구현하여 변환 로직의 책임을 DTO 스스로가 갖도록 합니다. 이는 코드의 응집도를 높이고, 변환 로직을 재사용하기 용이하게 만듭니다.

|  |
| --- |
| // TeamEvaluationStatusResponse.java DTO 예시 @Getter @Builder public class TeamEvaluationStatusResponse {  // ... 필드 ...   public static TeamEvaluationStatusResponse from(TeamEvaluation teamEvaluation) {  Period period = teamEvaluation.getPeriod();  return TeamEvaluationStatusResponse.builder()  .teamEvaluationId(teamEvaluation.getId())  .status(teamEvaluation.getStatus())  .startDate(period.getStartDate())  .endDate(period.getEndDate())  // ...  .build();  } } |

* **Request DTO → Entity 변환**: Request DTO를 기반으로 Entity를 생성해야 할 경우, **Entity 클래스 내에 of() 또는 from() 정적 팩토리 메서드**를 만듭니다. 이는 Entity의 생성 로직을 Entity 내부로 캡슐화하는 효과를 가집니다.

|  |
| --- |
| // Period.java Entity 예시 @Entity public class Period extends BaseEntity {  // ... 필드 ...   public static Period of(PeriodCreateRequest request, Integer orderInYear, String periodName) {  return Period.builder()  .year(request.getStartDate().getYear())  .periodName(periodName)  // ...  .build();  } } |

# AI 백엔드 개발 표준

## 프로젝트 구조

### 

본 프로젝트는 기능 중심의 계층형 아키텍처(Feature-Driven Layered Architecture)를 따릅니다. 전체 구조는 각 기능 모듈의 독립성과 재사용성을 높여 응집도를 강화하고, 모듈 간의 결합도를 낮추어 유연하고 확장 가능한 시스템을 구축하는 것을 목표로 합니다.

### 최상위 디렉터리 구조

프로젝트의 최상위 디렉터리는 기능적 책임에 따라 명확하게 분리된 패키지(디렉터리)들로 구성됩니다.

|  |
| --- |
| skoro-ai/ ├── agents/ # AI 에이전트: 핵심 AI 기능 모듈 ├── routers/ # API 계층: 외부 요청을 수신하는 엔드포인트 ├── services/ # 서비스 계층: 핵심 비즈니스 로직 오케스트레이션 ├── schemas/ # 데이터 모델 계층: 데이터 전송 객체 (DTO) ├── config/ # 설정 계층: 애플리케이션 전역 설정 ├── shared/ # 공통 계층: 재사용 가능한 유틸리티 모듈 ├── main.py # 애플리케이션 진입점 └── ... |

* **agents**: AI 모델, Langchain 등을 활용한 핵심 AI 기능의 실제 구현체가 위치하는 영역입니다.
* **routers, services, schemas**: FastAPI 애플리케이션의 웹 계층을 구성하며, 외부 요청을 처리하고 비즈니스 로직을 연결하는 역할을 합니다.
* **config, shared**: 특정 기능에 종속되지 않고 프로젝트 전반에서 사용되는 공통 기능을 관리합니다.

### 핵심 기능 계층 구조

애플리케이션의 핵심 기능은 routers, services, agents, schemas의 4개 계층이 유기적으로 연동하여 구현됩니다.

|  |
| --- |
| [Client] <--> routers <--> services <--> agents  ^ ^  | |  schemas schemas |

* **routers/ (Controller 계층)**
  + **역할**: API의 엔드포인트(Endpoint)를 정의하며, 클라이언트의 HTTP 요청을 가장 먼저 수신하는 관문(Gateway)입니다.
  + **책임**: 수신한 요청을 schemas를 통해 유효성 검증 후, 적절한 services의 메서드를 호출하고 그 결과를 HTTP 응답으로 반환하는 역할만 수행합니다. **직접적인 비즈니스 로직을 포함해서는 안 됩니다.**
  + **명명 규칙**: {기능}\_router.py (예: chat\_router.py)
* **services/ (Service 계층)**
  + **역할**: 비즈니스 로직의 흐름을 제어하고 조정(Orchestration)합니다.
  + **책임**: routers로부터 전달받은 데이터를 바탕으로, 실제 AI 기능 수행을 위해 agents의 특정 모듈을 호출하고, 여러 기능을 조합하여 최종 결과를 만들어냅니다. 데이터베이스 연동이나 트랜잭션 관리도 이 계층에서 이루어질 수 있습니다.
  + **명명 규칙**: {기능}\_service.py (예: chat\_service.py)
* **agents/ (AI Agent 계층)**
  + **역할**: **실질적인 AI 핵심 로직**을 캡슐화한 독립적인 기능 모듈입니다.
  + **책임**: 각 하위 디렉터리는 chatbot, evaluation 등과 같이 하나의 독립된 AI 기능을 나타냅니다. Langchain 체인, LLM 호출, 프롬프트 관리, 벡터 DB 검색 등 AI와 관련된 복잡한 로직은 모두 이 계층에 구현됩니다. services 계층은 이 에이전트를 호출하여 AI 기능을 사용합니다.
* **schemas/ (DTO 계층)**
  + **역할**: 계층 간, 특히 routers와 services 사이에서 데이터를 전달하기 위한 데이터 전송 객체(Data Transfer Object)를 정의합니다.
  + **책임**: Pydantic 모델을 사용하여 API의 요청(Request)과 응답(Response)의 명세를 정의합니다. 이를 통해 데이터 유효성 검사, 타입 안정성, API 문서 자동화의 이점을 얻습니다. **절대로 DB 모델(Entity)을 직접 노출해서는 안 됩니다.**
  + **명명 규칙**: 기능별로 파일을 분리하며, 목적에 따라 클래스명을 명확하게 구분합니다. (예: ChatRequest, ChatResponse)

### 공통 모듈 구조

프로젝트 전역에서 필요한 공통 기능들은 다음과 같이 기능별로 분리하여 관리합니다.

* **config/**: settings.py 등을 두어 **애플리케이션의 모든 설정**을 관리합니다. BaseSettings를 활용해 환경 변수로부터 API 키, DB 정보 등을 안전하게 로드합니다.
* **shared/**: 특정 도메인에 속하지 않는 **공용 유틸리티 모듈**이 위치합니다. 날짜 처리, 로깅 설정, 외부 API 래퍼 등 재사용 가능한 코드를 관리합니다.

## API 개발 표준 (FastAPI)

### 

본 프로젝트의 모든 API는 RESTful 디자인 원칙을 준수하여, 일관되고 예측 가능한 방식으로 동작하도록 설계합니다. FastAPI의 기능을 최대한 활용하여 타입 안정성, 자동 문서화, 높은 성능을 보장하는 것을 목표로 합니다.

### 라우팅 (Routing) 및 컨트롤러

routers 패키지는 API의 엔드포인트를 정의하는 컨트롤러 계층의 역할을 수행합니다.

* **APIRouter 사용**: 기능별 모듈화를 위해 반드시 APIRouter를 사용하여 라우터를 정의합니다. 각 라우터 파일은 chat\_router.py, evaluation\_router.py와 같이 명확한 기능 단위로 분리합니다.
* **태그(Tag) 명시**: main.py에서 라우터를 포함(include\_router)할 때, tags 속성을 명시하여 Swagger UI 문서에서 API를 기능별로 그룹화해야 합니다.

|  |
| --- |
| # main.py from routers import chat\_router, evaluation\_router  app.include\_router(chat\_router.router, prefix="/api/v1", tags=["Chat"]) app.include\_router(evaluation\_router.router, prefix="/api/v1", tags=["Evaluation"]) |

* **경로(Path) 명명 규칙**: API 경로는 소문자와 하이픈(-)을 사용하여 리소스를 명확하게 표현합니다. (예: /goal-achievement, /peer-talk)
* **책임 분리**: 라우터(컨트롤러)는 HTTP 요청/응답 처리, 경로 지정, 요청 데이터 검증 및 서비스 계층 호출의 역할만 담당합니다. **직접적인 비즈니스 로직, DB 접근, 외부 API 호출 코드를 포함해서는 안 됩니다.**

### 데이터 유효성 및 DTO (Schema)

schemas 패키지는 Pydantic 모델을 사용하여 계층 간 데이터 전송 객체(DTO)를 정의합니다.

* **Pydantic 모델 사용**: 모든 API 요청(Request Body)과 응답(Response)은 schemas에 정의된 Pydantic 모델을 사용해야 합니다. 이를 통해 런타임에 자동으로 데이터 유효성을 검사하고, 타입 힌트 기반의 코드 완성을 지원받습니다.
* **response\_model 명시**: 모든 엔드포인트에는 @router.post, @router.get 등의 데코레이터에 response\_model을 **반드시 명시**해야 합니다. 이는 API의 출력 형식을 고정하여 클라이언트와의 계약을 명확히 하고, 의도치 않은 내부 데이터(예: DB 모델의 모든 필드) 유출을 방지합니다.

|  |
| --- |
| # routers/chat\_router.py from schemas.chat import ChatRequest, ChatResponse  @router.post("/chat", response\_model=ChatResponse) def handle\_chat(request: ChatRequest):  # ...  return response |

* **목적에 따른 DTO 분리**: DTO는 목적에 따라 명확하게 분리합니다. 예를 들어, UserCreateRequest, UserResponse, UserUpdateRequest와 같이 생성, 조회, 수정 용도에 따라 별도의 클래스로 정의하여 각 API의 요구사항을 명확하게 표현합니다.

### 의존성 주입

FastAPI의 의존성 주입 시스템(Depends)을 적극적으로 활용하여 계층 간 결합도를 낮추고 코드의 재사용성과 테스트 용이성을 높입니다.

* **서비스 주입**: 라우터는 비즈니스 로직을 처리하기 위해 필요한 서비스 클래스를 Depends를 통해 주입받습니다.
* **데이터베이스 세션 주입**: db.py에 정의된 데이터베이스 세션 생성 함수(get\_db)를 Depends를 통해 각 엔드포인트에 주입합니다. 이를 통해 모든 요청이 독립적인 DB 세션을 갖도록 보장하고, 테스트 시 Mock DB 세션으로 쉽게 교체할 수 있습니다.

|  |
| --- |
| # routers/evaluation\_router.py from sqlalchemy.orm import Session from fastapi import Depends from db import get\_db from services.evaluation\_service import EvaluationService  @router.post("/evaluate") def run\_evaluation(  request: EvaluationRequest,  db: Session = Depends(get\_db),  eval\_service: EvaluationService = Depends() # 서비스 주입 ):  result = eval\_service.evaluate(db=db, data=request)  return result |

### 비동기 처리

데이터베이스 조회, 외부 API 호출 등 I/O 바운드 작업이 포함된 엔드포인트와 서비스 메서드는 async def를 사용하여 비동기로 작성하는 것을 원칙으로 합니다.

* **일관성 유지**: 하나의 요청 처리 흐름에서 async와 동기(sync) 코드를 혼용하는 것은 성능 저하 및 교착 상태(deadlock)를 유발할 수 있습니다. 라우터에서 서비스, 에이전트에 이르기까지 I/O 작업이 필요한 부분은 일관되게 async로 구현합니다.

## LangGraph 활용 표준

### 

본 표준은 LangGraph 프레임워크를 사용하여 LLM 기반 AI 기능을 개발할 때, 코드의 **일관성, 재사용성, 확장성, 안정성**을 확보하는 것을 목표로 합니다.  
LangGraph는 LangChain 위에서 동작하는 상태 기반(State-based) 워크플로우 프레임워크로, 복잡한 멀티에이전트 대화 흐름이나 조건 분기, 반복 처리, 기억(Memory) 처리 등을 명확하고 선언적으로 구현할 수 있는 장점이 있습니다.

AI 관련 로직은 agents 디렉터리 내에서 본 표준을 준수하여 구현되어야 하며, 그래프 기반으로 설계된 모든 기능은 **Graph 노드, 상태 전이, 그리고 LLM 함수 블록을 명시적으로 정의**해야 합니다.

### 그래프(Graph) 기반 워크플로우 구성 원칙

* **상태(State) 정의 표준**

#### 타입 안전성 확보 (Standard)

모든 상태는 TypedDict를 상속받아 정의하며, 병렬 처리와 상태 충돌을 방지하기 위해 Annotated 타입을 적극 활용합니다. 이는 런타임 오류를 컴파일 타임에 미리 감지하고, 코드의 의도를 명확히 하는 데 필수적입니다

|  |
| --- |
| # agents/evaluation/modules/module\_06\_4p\_evaluation/agent.py from typing import Annotated, List, Literal, TypedDict, Dict, Optional, Any import operator  from langgraph.graph import StateGraph, START, END  class Module6AgentState(TypedDict):  """모듈 6 (4P BARS 평가) 상태 - 병렬 처리 완전 지원"""    *# 병렬 누적 필드*  messages: Annotated[List[str], operator.add]    *# 읽기 전용 기본 정보*  report\_type: Literal["quarterly", "annual"]  team\_id: int  period\_id: int  emp\_no: str  feedback\_report\_id: Optional[int]  final\_evaluation\_report\_id: Optional[int]  raw\_evaluation\_criteria: str  *# 병렬 업데이트 가능한 딕셔너리 필드들*  evaluation\_criteria: Annotated[Dict[str, str], lambda x, y: {\*\*x, \*\*y}]  evaluation\_results: Annotated[Dict[str, Dict], lambda x, y: {\*\*x, \*\*y}]  integrated\_data: Annotated[Dict[str, Any], lambda x, y: {\*\*x, \*\*y}] |

* **디버깅 및 추적 (Debugging & Tracing)**
  + **원칙**: 디버깅하거나 성능을 분석할 때는 LangSmith와 같은 추적 도구를 적극적으로 활용합니다. 그래프의 각 단계를 시각적으로 확인하고, 입출력 값과 지연 시간을 분석하여 병목 지점을 찾을 수 있습니다.
  + **적용**: pyproject.toml에 langchain-teddynote 같은 라이브러리를 추가하거나, 환경변수 설정을 통해 LangSmith 연동을 활성화합니다.

### 그래프 구성 및 노드 연결 표준

#### 

그래프 구조는 워크플로우의 복잡도에 따라 순차 처리 또는 병렬 처리를 선택적으로 적용합니다. 4P 평가 모듈처럼 독립적인 평가가 가능한 경우에만 병렬 처리를 적용하고, 단순한 워크플로우는 순차 처리를 기본으로 합니다.

|  |
| --- |
| # agents/evaluation/modules/module\_06\_4p\_evaluation/agent.py  def create\_module6\_graph\_efficient():  """모듈 6 그래프"""  module6 = StateGraph(Module6AgentState)  *# 노드 정의 (initialize만 있음, parse 제거)*  module6.add\_node("initialize\_criteria",  initialize\_evaluation\_criteria\_agent)  module6.add\_node("passionate\_evaluation", passionate\_evaluation\_submodule)  module6.add\_node("proactive\_evaluation", proactive\_evaluation\_submodule)  module6.add\_node("professional\_evaluation",  professional\_evaluation\_submodule)  module6.add\_node("people\_evaluation", people\_evaluation\_submodule)  module6.add\_node("bars\_integration", bars\_integration\_submodule)  module6.add\_node("quarterly\_format\_and\_save",  quarterly\_format\_and\_save\_submodule)  module6.add\_node("annual\_format\_and\_save", annual\_format\_and\_save\_submodule)  *# 흐름 (initialize → 바로 평가들)*  module6.add\_edge(START, "initialize\_criteria")  module6.add\_edge("initialize\_criteria", "passionate\_evaluation")  module6.add\_edge("initialize\_criteria", "proactive\_evaluation")  module6.add\_edge("initialize\_criteria", "professional\_evaluation")  module6.add\_edge("initialize\_criteria", "people\_evaluation")  module6.add\_edge("passionate\_evaluation", "bars\_integration")  module6.add\_edge("proactive\_evaluation", "bars\_integration")  module6.add\_edge("professional\_evaluation", "bars\_integration")  module6.add\_edge("people\_evaluation", "bars\_integration")  def decide\_save\_path(state):  return (  "quarterly\_format\_and\_save"  if state["report\_type"] == "quarterly"  else "annual\_format\_and\_save"  )  module6.add\_conditional\_edges( "bars\_integration", decide\_save\_path,  { "quarterly\_format\_and\_save":  "quarterly\_format\_and\_save",  "annual\_format\_and\_save":  "annual\_format\_and\_save", }, )  module6.add\_edge("quarterly\_format\_and\_save", END)  module6.add\_edge("annual\_format\_and\_save", END)  return module6.compile() |

### 프롬프트 관리 표준

프롬프트는 AI의 행동을 결정하는 설계도입니다. 따라서 명확하고, 재사용 가능하며, 관리가 용이해야 합니다.

* **프롬프트 템플릿화 (Standard)**
  + **원칙**: 모든 프롬프트는 ChatPromptTemplate을 사용하여 템플릿으로 만듭니다. 하드코딩된 문자열(f-string) 사용을 지양하고, input\_variables를 명시하여 프롬프트가 어떤 입력을 필요로 하는지 명확히 해야 합니다.
* **메시지 역할 분리 (Standard)**
  + **원칙**: 채팅 모델(Chat\*)을 사용할 때는 SystemMessage, HumanMessage, AIMessage 등 역할에 맞는 메시지 타입을 명시적으로 사용해야 합니다. 이는 모델이 대화의 맥락과 각 발언의 역할을 더 잘 이해하도록 돕습니다.

|  |
| --- |
| from langchain\_core.prompts import ChatPromptTemplate  prompt = ChatPromptTemplate.from\_messages([  ("system", "당신은 IT 기술 전문 컨설턴트입니다."),  ("human", "{user\_input}"),  ("ai", "이전 답변: {previous\_answer}"), # Few-shot 예시 ]) |

### 모델 출력 처리 표준

LLM의 출력을 안정적으로 제어하고 애플리케이션에서 사용하기 쉬운 형태로 가공하는 것은 매우 중요합니다.

* **구조화된 출력 (Structured Output)**
  + **원칙**: LLM이 JSON과 같이 정해진 스키마의 결과물을 반환하도록 해야 할 때는, **Pydantic 모델과 Function/Tool Calling**을 사용하는 것을 표준으로 합니다. 이는 단순 파서보다 훨씬 안정적으로 동작합니다.
  + **예시**: 사용자 리뷰 텍스트에서 긍정/부정 여부와 핵심 키워드를 추출하는 경우

|  |
| --- |
| # schemas/evaluation.py from pydantic import BaseModel, Field from typing import List  class ReviewAnalysis(BaseModel):  """사용자 리뷰 분석 결과"""  sentiment: str = Field(description="리뷰의 감성. 'positive' 또는 'negative' 중 하나.")  keywords: List[str] = Field(description="리뷰의 핵심 키워드 리스트 (최대 5개)")  score: int = Field(description="리뷰 내용 기반의 평점 (1-10점)")  # agents/evaluation/analyzer.py from langchain\_openai import ChatOpenAI from schemas.evaluation import ReviewAnalysis # Pydantic 모델 임포트  # 1. 모델에 Pydantic 스키마를 도구(tool)로 바인딩 llm = ChatOpenAI(model="gpt-4o") structured\_llm = llm.with\_structured\_output(ReviewAnalysis)  # 2. 체인 구성 # 프롬프트는 분석할 텍스트를 입력받도록 설계 analysis\_chain = prompt | structured\_llm  # 3. 실행 review\_text = "이 제품은 배송이 정말 빨랐고 품질도 만족스러워요." result: ReviewAnalysis = analysis\_chain.invoke({"review\_text": review\_text}) # result.sentiment -> 'positive' # result.score -> 10 |

### 에이전트 및 메모리 활용 표준

* **사용자 정의 도구 (Custom Tool)**
  + **원칙**: 에이전트가 사용할 도구를 정의할 때는 @tool 데코레이터를 사용합니다. 이때 도구 함수의 **Docstring**은 LLM이 도구의 기능과 인자를 이해하는 데 결정적인 역할을 하므로, **반드시 명확하고 상세하게 작성**해야 합니다.

|  |
| --- |
| from langchain\_core.tools import tool  @tool def search\_internal\_document(query: str) -> str:  """  사내 문서 데이터베이스에서 관련 문서를 검색할 때 사용합니다.  사용자가 제품 기능, 정책, 내부 절차에 대해 질문할 때 이 도구를 호출하세요.  'query' 인자에는 사용자의 질문을 핵심 키워드 위주로 요약하여 전달해야 합니다.  """  # ... 문서 검색 로직 구현 ...  return "검색 결과입니다..." |

* **메모리 관리 및 영속성**
  + **원칙**: 사용자와의 연속적인 대화 기록을 관리해야 할 경우, 목적에 맞는 메모리 모듈을 선택합니다. 단순 대화는 ConversationBufferMemory, 긴 대화는 ConversationSummaryBufferMemory를 사용합니다.
  + **영속성**: 사용자 세션이 유지되어야 하는 운영 환경에서는 인메모리(In-memory) 방식이 아닌, Redis나 데이터베이스와 연동된 ChatMessageHistory를 구현하여 메모리를 영속적으로 관리해야 합니다. 이는 services 계층에서 사용자 ID 또는 세션 ID를 기준으로 메모리를 로드하고 저장하는 로직으로 구현됩니다.

## 벡터 DB (Pinecone) 연동 표준

### 

본 프로젝트는 RAG(검색 증강 생성) 기능의 핵심 요소인 벡터 검색을 위해 **Pinecone**을 표준 벡터 데이터베이스로 사용합니다. 이 표준은 Pinecone을 일관되고 효율적으로 사용하여 데이터의 색인 및 검색 성능을 최적화하는 것을 목표로 합니다. Pinecone 관련 로직은 agents와 shared 계층에서 관리됩니다.

### 기본 원칙 및 구조

* **역할 분리**:
  + **shared/vector\_store/**: Pinecone 클라이언트 초기화, 인덱스 생성 및 상태 확인 등 저수준(low-level)의 공통 연동 로직은 shared 패키지 내에 배치하여 재사용성을 높입니다.
  + **agents/{agent\_name}/**: 특정 AI 기능(에이전트)에 종속적인 데이터의 **색인(Indexing) 및 검색(Retrieval)** 로직은 해당 에이전트 디렉터리 내에서 구현합니다. 이를 통해 각 에이전트는 독립적인 벡터 저장소 로직을 가질 수 있습니다.
* **설정 중앙 관리**:
  + Pinecone API 키, 환경(Environment), 인덱스 이름 등 모든 설정 값은 config/settings.py를 통해 중앙에서 관리되어야 합니다. 코드 내에 설정 값을 하드코딩하는 것을 엄격히 금지합니다.

### 데이터 색인 표준

문서나 데이터를 벡터로 변환하여 Pinecone에 저장(upsert)하는 과정은 다음 표준을 따릅니다.

* **1. 문서 분할 (Splitting)**
  + **표준 분할기**: 텍스트 분할에는 RecursiveCharacterTextSplitter를 사용하는 것을 표준으로 합니다. 이 분할기는 문단, 문장 등 의미 단위로 텍스트를 유연하게 분할하여 컨텍스트 손실을 최소화합니다.
  + **청크 크기 및 겹침**: chunk\_size와 chunk\_overlap은 문서의 특성에 맞게 조정하되, 프로젝트의 기본값은 chunk\_size=1000, chunk\_overlap=200으로 설정하여 일관성을 유지합니다.
* **2. 임베딩 (Embedding)**
  + **표준 모델**: 프로젝트의 표준 임베딩 모델은 text-embedding-3-small (OpenAI)을 사용합니다. 모델 정보 역시 config/settings.py에서 관리하여, 필요시 쉽게 다른 모델로 교체할 수 있도록 합니다.
* **3. 메타데이터 (Metadata) 구조화**
  + **원칙**: 모든 벡터에는 검색 효율성과 결과의 신뢰도를 높이기 위해 **반드시 구조화된 메타데이터를 포함**해야 합니다. 이는 필터링 검색의 기반이 되며, RAG 답변의 출처를 추적하는 데 필수적입니다.
  + **표준 메타데이터 필드**:
    - source (str): 문서의 출처 (예: 파일명, URL, DB 테이블명)
    - doc\_id (str): 문서의 고유 식별자
    - created\_at (float): 색인 생성 시점의 Unix timestamp
    - author (str, optional): 문서 작성자
    - category (str, optional): 문서 분류 (예: "정책", "기술", "인사")

|  |
| --- |
| # agents/chatbot/indexer.py from langchain\_text\_splitters import RecursiveCharacterTextSplitter from langchain\_openai import OpenAIEmbeddings from langchain\_pinecone import PineconeVectorStore import time  # ... 문서 로드 ... text\_splitter = RecursiveCharacterTextSplitter(chunk\_size=1000, chunk\_overlap=200) docs = text\_splitter.split\_documents(documents)  # 메타데이터 추가 for doc in docs:  doc.metadata = {  "source": "internal\_policy\_v1.2.pdf",  "doc\_id": "policy-001",  "created\_at": time.time(),  "category": "인사"  } embeddings = OpenAIEmbeddings(model="text-embedding-3-small") PineconeVectorStore.from\_documents(  docs, embeddings, index\_name="skoro-ai-prod" ) |

* **배치 처리 (Batching)**
  + **원칙**: 대량의 문서를 색인할 때는 from\_documents 메서드를 사용하거나, 수동으로 배치(batch)를 구성하여 add\_texts를 호출해야 합니다. 단일 문서를 반복적으로 upsert하는 방식은 성능 저하를 유발하므로 지양합니다.

### 데이터 검색 표준

* **리트리버(Retriever) 사용**
  + **원칙**: Pinecone에서의 벡터 검색은 Langchain의 Retriever 인터페이스를 통해 추상화되어야 합니다. vectorstore.as\_retriever()를 사용하여 리트리버를 생성하는 것을 표준으로 합니다. 이는 LCEL 체인과의 통합을 용이하게 합니다.
* **메타데이터 필터링 검색**
  + **원칙**: 검색의 정확도와 속도를 높이기 위해, 가능한 경우 항상 **메타데이터 필터링**을 사용해야 합니다. 사용자의 의도나 현재 컨텍스트에 따라 검색 범위를 좁히는 것은 RAG 성능 향상의 핵심입니다.
  + **예시**: 특정 카테고리의 문서 내에서만 검색

|  |
| --- |
| # agents/chatbot/vectorstore.py from langchain\_pinecone import PineconeVectorStore from langchain\_openai import OpenAIEmbeddings  embeddings = OpenAIEmbeddings(model="text-embedding-3-small") vectorstore = PineconeVectorStore.from\_existing\_index(  index\_name="skoro-ai-prod", embedding=embeddings )  # '인사' 카테고리 문서만 검색하도록 필터링된 리트리버 생성 retriever = vectorstore.as\_retriever(  search\_kwargs={  "k": 5, # 검색할 문서 개수  "filter": {"category": "인사"}  } )  # 이 리트리버는 RAG 체인에 연결되어 사용됨 # relevant\_docs = retriever.invoke("연차 사용 규정이 어떻게 되나요?") |

* **검색 파라미터 관리**
  + k (검색 개수): 검색할 문서의 개수(k)는 LLM에 제공할 컨텍스트의 양을 결정합니다. 기본값은 5로 설정하되, 질문의 유형이나 토큰 제한에 따라 동적으로 조정할 수 있도록 구현합니다.

## 파이썬 코딩 스타일

### 

본 프로젝트는 코드의 **가독성, 일관성, 유지보수성**을 최우선 가치로 삼습니다. 모든 파이썬 코드는 PEP 8 스타일 가이드를 준수하며, 아래에 명시된 규칙과 도구를 통해 일관된 코드 스타일을 유지해야 합니다.

### 코드 포매터 (Code Formatter)

* **표준 도구**: **Black**
* **원칙**: 모든 파이썬 코드는 예외 없이 Black 포매터를 사용하여 형식을 통일합니다. 이는 코드 리뷰 시 스타일과 관련된 불필요한 논쟁을 없애고, 개발자가 로직에만 집중할 수 있도록 돕습니다.
* **적용**: pyproject.toml에 Black 설정을 명시하며, Git의 pre-commit 훅(hook)을 설정하여 커밋 이전에 모든 코드가 자동으로 포맷팅되도록 강제하는 것을 권장합니다.

|  |
| --- |
| # pyproject.toml [tool.black] line-length = 88 target-version = ['py311'] |

### 린터 (Linter)

* **표준 도구**: **Ruff**
* **원칙**: Ruff를 사용하여 잠재적인 버그, 안티 패턴, 스타일 오류를 정적으로 분석하고 해결합니다. Ruff는 매우 빠르며, Flake8, isort 등 여러 도구의 기능을 통합하고 있어 효율적인 코드 품질 관리가 가능합니다.
* **적용**: pyproject.toml에 Ruff 규칙을 설정합니다. 특히 isort 규칙을 활성화하여 임포트 순서가 자동으로 정렬되도록 합니다.

|  |
| --- |
| # pyproject.toml [tool.ruff] line-length = 88 select = ["E", "W", "F", "I", "C", "B"] # Flake8, isort 등 기본 규칙 활성화 ignore = [] |

### 명명 규칙

* **snake\_case (소문자와 밑줄)**
  + **대상**: 변수(variables), 함수(functions), 메서드(methods), 모듈(modules), 패키지(packages)
  + **예시**: user\_input, calculate\_score, chat\_service.py
* **PascalCase (단어의 첫 글자를 대문자로)**
  + **대상**: 클래스(Classes)
  + **예시**: ChatService, EvaluationRouter, KpiResultSchema
* **UPPER\_SNAKE\_CASE (전체 대문자와 밑줄)**
  + **대상**: 상수(Constants)
  + **예시**: DEFAULT\_MODEL\_NAME, PINECONE\_API\_KEY

### 타입 힌트

* **원칙**: 프로젝트의 **모든 코드에는 타입 힌트를 반드시 명시**해야 합니다. 이는 코드의 명확성을 비약적으로 향상시키고, 정적 분석 도구(Mypy, Pyright)를 통해 런타임 이전에 타입 관련 오류를 발견할 수 있게 해줍니다.
* **적용**:
  + 함수와 메서드의 모든 인자(arguments)와 반환 값(return value)에 타입을 명시합니다.
  + 변수를 선언할 때 타입을 명시하여 의도를 분명히 합니다.

|  |
| --- |
| # services/chat\_service.py from schemas.chat import ChatRequest, ChatResponse from agents.chatbot import ChatbotAgent  class ChatService:  def \_\_init\_\_(self, agent: ChatbotAgent):  self.agent = agent   def get\_chat\_response(self, request: ChatRequest) -> ChatResponse:  # 변수 선언 시 타입 명시  user\_message: str = request.message  response\_text: str = self.agent.invoke(user\_message)  return ChatResponse(reply=response\_text) |

### 독스트링과 주석

* **독스트링 (Docstrings)**
  + **원칙**: 모든 모듈, 함수, 클래스, 메서드의 시작 부분에는 **Google Python 스타일 가이드** 형식의 독스트링을 작성하여 해당 코드의 목적, 인자, 반환 값에 대해 명확하게 설명해야 합니다.

|  |
| --- |
| def calculate\_kpi(employee\_id: int, year: int) -> float:  """직원의 연간 KPI 달성률을 계산합니다.   Args:  employee\_id (int): 조회할 직원의 고유 ID.  year (int): 조회할 연도.   Returns:  float: 해당 연도의 KPI 달성률 (0.0 ~ 1.0 사이의 값).   Raises:  EmployeeNotFoundError: 해당 ID의 직원이 존재하지 않을 경우.  """  # ... function body ... |

* **주석 (Comments)**
  + **원칙**: 코드로 명확하게 표현되지 않는 "왜(Why)"에 해당하는 부분을 설명하기 위해 인라인 주석을 사용합니다. 코드의 동작 방식인 "어떻게(How)"를 설명하는 주석은 지양하고, 대신 코드를 더 읽기 쉽게 작성하는 것을 우선으로 합니다.

# 인프라 및 환경 표준

## Dockerfile 작성 표준

### 

본 문서는 SKoro 프로젝트의 Docker 이미지 빌드 프로세스를 표준화하여, 이미지의 경량화, 빌드 시간 최적화, 그리고 보안 강화를 목표로 합니다. 모든 Dockerfile은 아래의 표준을 준수해야 합니다.

### 멀티 스테이지 빌드 (Multi-stage Build) 사용

**빌드(Build) 환경과 런타임(Runtime) 환경을 분리**하여 최종 이미지의 크기를 최적화하고 불필요한 파일을 제거합니다.

* **Builder 스테이지:** 소스 코드를 컴파일하고 실행 가능한 아티팩트(예: .jar 파일)를 생성하는 역할을 합니다. gradle이나 maven과 같은 빌드 도구와 JDK가 포함된 이미지를 사용합니다.
  + AS builder와 같이 스테이지에 명확한 별칭을 부여하여 이후 스테이지에서 쉽게 참조할 수 있도록 합니다.

|  |
| --- |
| # 1. 빌드 스테이지 FROM gradle:8.4.0-jdk17 AS builder WORKDIR /app COPY . . RUN gradle build -x test |

* **Runtime 스테이지:** 빌드된 애플리케이션을 실행하는 데 필요한 최소한의 환경만 포함합니다. JRE(Java Runtime Environment)만 포함된 경량 이미지를 사용해야 합니다.
  + COPY --from=builder 구문을 사용하여 **Builder 스테이지에서 생성된 아티팩트만**을 런타임 스테이지로 가져옵니다. 이를 통해 빌드 도구, 소스 코드, 중간 파일 등이 최종 이미지에 포함되는 것을 방지합니다.

|  |
| --- |
| # 2. 런타임 스테이지 FROM eclipse-temurin:17-jdk WORKDIR /app COPY --from=builder /app/build/libs/\*.jar app.jar |

### 기본 이미지 선택

이미지의 목적에 맞게 공식적이고 검증된 이미지를 사용합니다.

* **버전 명시:** 이미지 태그는 latest 대신 **구체적인 버전(예: gradle:8.4.0-jdk17)을 명시**하여 빌드 환경의 일관성을 보장하고, 예기치 않은 버전 변경으로 인한 문제를 방지합니다.
* **경량 이미지 사용:** 최종 런타임 환경에서는 alpine 버전이나 eclipse-temurin과 같이 가볍고 안정적인 이미지를 사용하여 이미지 크기를 최소화하고 보안을 강화합니다.

### 명령어 작성 규칙

Dockerfile의 가독성과 유지보수성, 그리고 빌드 효율성을 높이기 위한 규칙입니다.

* **WORKDIR 사용:** RUN, COPY, CMD 등의 명령어를 실행할 작업 디렉토리를 지정하기 위해 **WORKDIR을 명시적으로 사용**합니다. 이를 통해 명령어에서 절대 경로를 사용하는 것을 피하고, Dockerfile을 더 깔끔하게 관리할 수 있습니다.
* **ENTRYPOINT 사용:** 컨테이너가 시작될 때 실행될 고정 명령어를 지정합니다.
  + 배열(Exec) 형식 (["executable", "param1", "param2"])으로 작성하여 쉘(Shell)을 거치지 않고 직접 실행되도록 합니다. 이는 sh -c와 관련된 문제를 예방할 수 있습니다.
  + 백엔드 애플리케이션의 경우, java -jar app.jar와 같이 애플리케이션을 실행하는 명령어를 ENTRYPOINT로 설정합니다.

|  |
| --- |
| ENTRYPOINT ["java", "-jar", "app.jar"] |

## Jenkinsfile 작성 표준

### 

본 문서는 SKoro 프로젝트의 CI/CD 파이프라인을 정의하는 **Jenkinsfile의 작성 표준**을 규정합니다. 파이프라인의 일관성, 가독성, 그리고 유지보수성을 높이는 것을 목표로 하며, 모든 Jenkinsfile은 아래의 표준을 준수해야 합니다.

### 선언형 파이프라인 사용

파이프라인은 **선언형(Declarative) 구문**을 사용하여 pipeline { ... } 블록 안에 정의합니다. 이는 파이프라인의 구조를 더 명확하게 표현하고, Groovy 문법에 대한 깊은 이해 없이도 쉽게 작성하고 수정할 수 있게 합니다.

|  |
| --- |
| pipeline {  agent any  environment { /\* ... \*/ }  stages { /\* ... \*/ }  post { /\* ... \*/ } } |

### 환경 변수 관리

파이프라인 전반에서 사용되는 변수는 **environment 블록** 안에 중앙에서 관리하여 가독성과 재사용성을 높입니다.

* **변수 정의:** 이미지 이름, 태그, Git 브랜치 등 파이프라인의 설정과 관련된 값들을 정의합니다.
* **네이밍 컨벤션:** 변수 이름은 \*\*대문자와 스네이크 케이스(SNAKE\_CASE)\*\*를 사용하여 명확하게 구분합니다.
* **Credential ID 분리:** GitHub, 이미지 레지스트리 등 외부 시스템에 접근하기 위한 Credential ID를 변수로 정의하여 stages 내에서 일관되게 참조합니다.

|  |
| --- |
| environment {  IMAGE\_NAME = 'skoro-backend'  IMAGE\_TAG = '1.0.0'  GIT\_BRANCH = 'main'  GIT\_CREDENTIAL\_ID = 'skala-github-id'  HARBOR\_CREDENTIAL\_ID = 'skala-image-registry-id' } |

### 스테이지 구성

빌드, 테스트, 배포 등 CI/CD 프로세스의 각 단계를 **논리적인 stage로 분리**하여 파이프라인의 흐름을 명확하게 표현합니다.

* **단일 책임 원칙:** 각 stage는 **하나의 주요 목적**(예: 'Docker 빌드 및 푸시', '인프라 리포지토리 업데이트')을 가져야 합니다.
* **script 블록 최소화:** 복잡한 로직이나 조건부 실행이 필요할 경우에만 script 블록을 사용합니다. 가능한 선언형 구문을 우선으로 사용합니다.

|  |
| --- |
| stages {  stage('Docker Build & Push') {  steps {  // ...  }  }  stage('Update Infra Repository') {  steps {  // ...  }  } } |

### 보안 및 인증

비밀번호, API 키, 토큰 등 민감한 정보는 절대로 Jenkinsfile에 하드코딩하지 않습니다. **Jenkins Credential 관리 기능**과 **withCredentials 블록**을 사용해야 합니다.

* **withCredentials 사용:** steps 내에서 민감 정보가 필요한 경우, withCredentials 블록으로 감싸서 해당 정보가 필요한 범위 내에서만 안전하게 변수로 주입합니다.
* **Docker 레지스트리 인증:** docker.withRegistry 헬퍼 메서드를 사용하여 Docker 이미지 레지스트리에 대한 인증을 처리합니다.

|  |
| --- |
| withCredentials([  usernamePassword(credentialsId: "${GIT\_CREDENTIAL\_ID}", usernameVariable: 'GIT\_USERNAME', passwordVariable: 'GIT\_PASSWORD') ]) {  // 이 블록 안에서만 GIT\_USERNAME, GIT\_PASSWORD 변수 사용 가능 } |

### GitOps 기반 배포 자동화

애플리케이션의 새 버전이 빌드되면, **관련 인프라 리포지토리(예: skoro-infra)의 설정(예: Kubernetes deployment.yaml의 이미지 태그)을 자동으로 업데이트**하는 단계를 포함합니다.

* **동적 이미지 태그 생성:** 빌드의 고유성을 보장하기 위해 기본 이미지 태그에 Jenkins의 내장 변수인 **${BUILD\_NUMBER}를 조합**하여 최종 이미지 태그를 생성합니다.
* **인프라 리포지토리 업데이트:**
  1. git clone을 통해 최신 인프라 코드를 가져옵니다.
  2. sed나 다른 스크립트 도구를 사용하여 deployment.yaml 파일의 이미지 태그를 새로 빌드된 버전으로 교체합니다.
  3. 변경 사항을 git commit 하고 git push 하여 인프라 리포지토리에 반영합니다. 이 커밋은 ArgoCD와 같은 GitOps 도구에 의해 감지되어 클러스터에 자동으로 배포됩니다.

|  |
| --- |
| // 1. 최종 이미지 태그 생성 def FINAL\_IMAGE\_TAG = "${IMAGE\_TAG}-${BUILD\_NUMBER}"  // 2. 인프라 리포지토리의 YAML 파일 수정 sh """  currentTag=\$(grep 'image:' apps/skoro-backend/deployment.yaml | awk -F ':' '{print \$NF}')  sed -i "s|\\$currentTag|${env.FINAL\_IMAGE\_TAG}|g" apps/skoro-backend/deployment.yaml  git add .  git commit -m "[AUTO] Update backend tag: ${env.FINAL\_IMAGE\_TAG}"  git push origin ${GIT\_BRANCH} """ |

### 파이프라인 후처리

파이프라인의 최종 실행 결과(성공, 실패 등)에 따라 **항상 실행되어야 하는 정리 작업이나 알림**을 post 블록에 정의합니다.

* **success 블록:** 파이프라인이 성공적으로 완료되었을 때 실행됩니다. 성공 메시지를 출력하거나 Slack 알림 등을 보낼 수 있습니다.
* **failure 블록:** 파이프라인이 실패했을 때 실행됩니다. 실패 원인을 파악하는 데 도움이 되는 로그를 남기거나 실패 알림을 보냅니다.
* **always 블록:** 성공/실패 여부와 관계없이 항상 실행되어야 하는 작업(예: 임시 파일 삭제)을 정의할 수 있습니다.

|  |
| --- |
| post {  success {  echo "✅ Backend CI Pipeline Completed!"  }  failure {  echo "❌ Pipeline Failed! Check logs."  } } |

## Kubernetes YAML 작성 표준

### 

본 문서는 AI 성과 관리 프로젝트의 쿠버네티스 리소스를 정의하는 **YAML 파일의 작성 표준**을 규정합니다. 리소스의 일관성, 가독성, 그리고 GitOps 기반의 관리 효율성을 높이는 것을 목표로 하며, 모든 쿠버네티스 매니페스트(Manifest) 파일은 아래의 표준을 준수해야 합니다.

### 리소스 구조 및 관리

애플리케이션과 리소스를 효율적으로 관리하기 위해 **디렉토리 구조와 Kustomize를 표준**으로 사용합니다.

* **애플리케이션별 디렉토리 분리:** 모든 쿠버네티스 리소스는 apps/ 디렉토리 하위에 **애플리케이션(컴포넌트)별로 독립된 디렉토리를 생성**하여 관리합니다. 예를 들어, 프론트엔드 관련 리소스는 apps/skoro-frontend/에 위치합니다.
* **리소스 파일 분리:** 각 애플리케이션 디렉토리 내에서 Deployment, Service 등 쿠버네티스 리소스는 **기능별로 개별 YAML 파일에 정의**합니다.
* **Kustomize 활용:** 각 애플리케이션 디렉토리에는 **kustomization.yaml 파일을 두어** 해당 애플리케이션을 구성하는 모든 리소스 파일들을 목록으로 관리합니다. 이는 애플리케이션 단위의 리소스 관리를 용이하게 합니다.
* **리소스 이름 (Naming Convention):** 모든 리소스의 metadata.name은 **"skoro-{컴포넌트명}" 형식**을 따릅니다 (예: skoro-frontend, skoro-backend-secret).

### 메타데이터 표준

리소스 식별과 관리를 용이하게 하기 위해 **네임스페이스, 레이블, 어노테이션을 표준화**합니다.

* **네임스페이스 (Namespace):**
  + 모든 애플리케이션 리소스(Deployment, Service, Ingress 등)는 **sk-team-10 네임스페이스**에 배포되어야 합니다.
  + ArgoCD Application CRD는 **skala-argocd 네임스페이스**에 위치합니다.
* **레이블 (Labels):**
  + **표준 식별 레이블:** Deployment와 Service 등 연관된 리소스를 연결하기 위해 **app: {애플리케이션명} 레이블을 필수**로 사용합니다 (예: app: skoro-frontend).
  + **Selector 일치:** Deployment의 spec.selector.matchLabels와 spec.template.metadata.labels는 반드시 일치해야 하며, Service의 spec.selector는 연결할 파드(Pod)의 레이블을 정확히 타겟해야 합니다.
* **어노테이션 (Annotations):**
  + **Prometheus 모니터링:** 모니터링을 위해 모든 Deployment의 template.metadata.annotations에 **Prometheus 스크래핑(scraping) 설정을 추가**해야 합니다.
    - prometheus.io/scrape: 'true'
    - prometheus.io/port: '{애플리케이션-actuator-포트}'
    - prometheus.io/path: '/actuator/prometheus'
  + **TLS 인증서 자동화:** Ingress 리소스에는 cert-manager.io/cluster-issuer: letsencrypt-prod 어노테이션을 사용하여 **Let's Encrypt로부터 TLS 인증서를 자동으로 발급**받도록 설정합니다.

### Deployment 명세 표준

애플리케이션의 배포 속성과 실행 환경을 정의하는 Deployment 리소스의 표준입니다.

* **컨테이너 이미지:**
  + **프라이빗 레지스트리 사용:** 모든 컨테이너 이미지는 **amdp-registry.skala-ai.com** 레지스트리에서 가져와야 합니다.
  + **이미지 태그:** 이미지 태그는 **{버전}-{빌드번호} 형식**을 따르며, CI/CD 파이프라인에 의해 동적으로 업데이트됩니다.
  + **이미지 Pull 정책:** imagePullPolicy는 **Always로 설정**하여 배포 시 항상 최신 버전의 이미지를 가져오도록 보장합니다.
* **환경변수 (Environment Variables):**
  + **민감 정보는 Secret 사용:** 데이터베이스 접속 정보, API 키, JWT 시크릿 등 모든 민감 정보는 \*\* 절대로 YAML 파일에 직접 작성하지 않습니다\*\*. 대신, Secret 리소스를 생성하고 env.valueFrom.secretKeyRef를 통해 컨테이너 환경변수로 주입해야 합니다.
  + **일반 설정값:** TZ(타임존), SPRING\_PROFILES\_ACTIVE 등 민감하지 않은 일반 설정값은 env.value를 통해 직접 설정할 수 있습니다.

### Service 및 Ingress 명세 표준

클러스터 내부와 외부의 네트워크 연결을 정의하는 표준입니다.

* **Service 타입:** 클러스터 내부 통신을 위한 모든 서비스는 \*\*type: ClusterIP\*\*를 사용합니다.
* **Ingress 설정:**
  + **Ingress Class:** 외부 트래픽 유입을 위해 \*\*ingressClassName: public-nginx\*\*를 명시해야 합니다.
  + **TLS (HTTPS):** Ingress의 spec.tls 설정을 통해 **HTTPS 통신을 활성화**하고, secretName에는 Cert-Manager가 생성한 TLS 시크릿 이름을 지정합니다.
  + **Path 기반 라우팅:** 단일 호스트(skoro.skala25a.project.skala-ai.com)에 대해 **URL path를 기준으로 트래픽을 적절한 백엔드 서비스로 라우팅**해야 합니다.

### GitOps - ArgoCD Application 표준

모든 애플리케이션은 **ArgoCD를 통해 GitOps 방식으로 배포**되어야 하며, 이를 위한 Application CRD(Custom Resource Definition) 작성 표준입니다.

* **애플리케이션 리소스 정의:** 각 컴포넌트(frontend, backend 등)는 **개별 Application 리소스로 정의**되어야 합니다.
* **Source of Truth (SoT):**
  + spec.source.repoURL: 매니페스트가 저장된 **Git 리포지토리 주소**를 명시합니다 (예: https://github.com/SKALA-10/SKoro-Infra.git).
  + spec.source.path: 해당 애플리케이션의 리소스들이 위치한 **apps/ 하위의 경로**를 정확히 지정해야 합니다.
  + spec.source.targetRevision: 추적할 Git 브랜치를 **main으로 설정**합니다.
* **자동 동기화 정책 (Automated Sync Policy):**
  + prune: true: Git에서 삭제된 리소스가 클러스터에서도 자동으로 제거되도록 설정합니다.
  + selfHeal: true: 클러스터의 라이브 상태가 Git에 정의된 상태와 달라졌을 경우, 자동으로 바로잡도록 설정합니다.

## 환경 변수 관리 표준

### 

본 문서는 SKoro 프로젝트의 애플리케이션 환경 구성을 위한 **환경 변수 관리 표준**을 규정합니다. 설정(Configuration)을 코드와 분리하고, 특히 민감 정보를 안전하게 관리하여 보안을 강화하며, 환경별 구성을 체계화하는 것을 목표로 합니다.

### 기본 원칙: 설정의 외부화

애플리케이션의 동작에 필요한 모든 설정값(데이터베이스 주소, API 키, 기능 플래그 등)은 **코드에 하드코딩하지 않고 외부에서 주입**하는 것을 원칙으로 합니다. 이를 위해 쿠버네티스의 Secret과 ConfigMap(필요시)을 사용합니다.

* **12-Factor App 원칙 준수:** 이 방식은 환경에 따라 달라지는 설정을 코드 변경 없이 배포할 수 있도록 하는 "The Twelve-Factor App"의 세 번째 원칙, \*\*"설정(Config)"\*\*을 따릅니다.

### 민감 정보 관리: Kubernetes Secret 사용

**API 키, 데이터베이스 비밀번호, JWT 시크릿 등 모든 종류의 민감 정보는 반드시 Kubernetes Secret 리소스를 사용하여 관리**해야 합니다.

* **YAML 파일에 직접 작성 금지:** Deployment 등 어떠한 YAML 파일에도 민감 정보를 평문으로 작성해서는 안 됩니다.
* **Secret 주입 방식:** Secret에 저장된 값은 env.valueFrom.secretKeyRef를 통해 컨테이너의 환경 변수로 주입합니다. 각 key는 Secret 리소스 내에서 해당 값과 매핑되는 키를 가리킵니다.

|  |
| --- |
| # 예시: deployment.yaml env:  - name: DB\_PASSWORD # 컨테이너 내에서 사용할 환경 변수 이름  valueFrom:  secretKeyRef:  name: skoro-backend-secret # 참조할 Secret 리소스의 이름  key: DB\_PASSWORD # Secret 리소스 내의 데이터 키 |

* **Secret 네이밍 컨벤션:** Secret의 이름은 **{애플리케이션명}-secret 형식**을 따릅니다 (예: skoro-backend-secret, skoro-ai-secret). 이는 어떤 애플리케이션을 위한 Secret인지 명확하게 식별할 수 있도록 합니다.

### 일반 설정 정보 관리

민감하지 않으면서 애플리케이션의 동작에 필요한 일반 설정값은 Deployment YAML 파일에 직접 명시할 수 있습니다.

* **env.value 사용:** 타임존(TZ), Spring Profile(SPRING\_PROFILES\_ACTIVE) 등 보안에 민감하지 않은 정적인 설정값은 env.value를 통해 직접 주입합니다.

|  |
| --- |
| # 예시: deployment.yaml env:  - name: TZ  value: Asia/Seoul  - name: SPRING\_PROFILES\_ACTIVE  value: prod  - name: JAVA\_TOOL\_OPTIONS  value: "-Duser.timezone=Asia/Seoul" |

* **ConfigMap 사용 (필요시):** 만약 일반 설정값의 종류가 많아져 Deployment 파일이 복잡해지는 경우, 관련 설정들을 모아 ConfigMap 리소스를 생성하고 envFrom.configMapRef를 통해 주입하는 것을 고려할 수 있습니다.

### 환경별 설정 관리

개발(dev), 스테이징(staging), 운영(prod) 등 여러 환경에 배포할 경우, 각 환경에 맞는 Secret과 ConfigMap을 별도로 생성하여 관리해야 합니다.

**Kustomize 오버레이 활용:** Kustomize의 overlays 기능을 사용하면 base 디렉토리에 공통 리소스를 정의하고, 각 환경(예: overlays/prod, overlays/dev)별로 달라지는 설정값(Secret, ConfigMap 등)만 패치(patch)하여 환경별 구성을 효율적으로 관리할 수 있습니다. (현재 레포지토리 구조에서는 각 애플리케이션 디렉토리가 단일 환경(prod)을 기준으로 작성되어 있으나, 다중 환경 구성 시 이 패턴을 적용합니다.)

## 

이 페이지의 끝