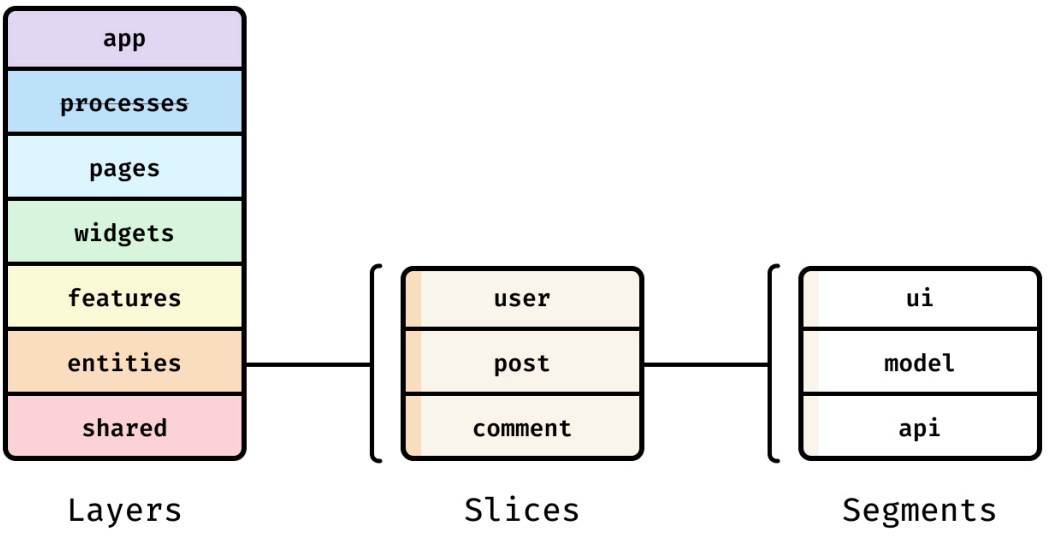
Feature Sliced Design 아키텍처 컨벤션

▼ 1. 개요

1.1 핵심 원칙

- **기능 중심 구조화**: 비즈니스 기능(feature)을 중심으로 코드를 구조화
- **책임 분리**: 각 계층(layer)은 명확한 책임을 가짐
- **단방향 의존성**: 상위 계층에서 하위 계층으로만 의존성이 흐름
- **명시적 내보내기**: 각 모듈은 index.ts 파일을 통해 외부에 공개할 컴포넌트, 함수, 타입 등만 명확하게 내보냄
- **고립된 슬라이스**: 슬라이스 간 직접 의존성 지양

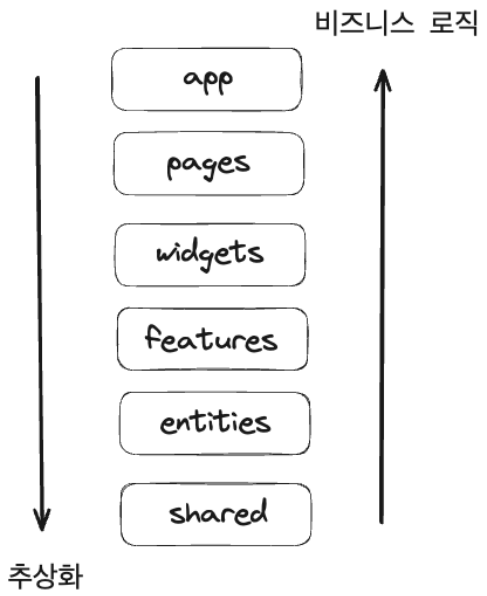
1.2 FSD 구성요소



1.3 FSD 계층

Layers

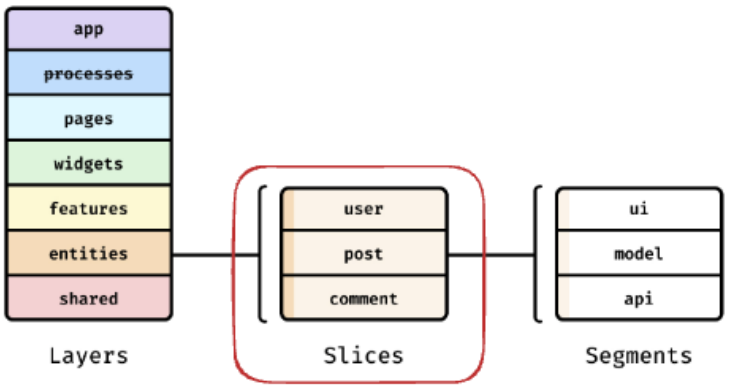
> FSD의 첫 번째 계층
>
> 추상화와 비즈니스 로직 수준에 따라 수직적으로 구성



- 레이어 간 의존성 규칙
  - 상위 레이어는 하위 레이어에 의존 가능
  - 하위 레이어는 상위 레이어에 의존 불가능
- 레이어 특성
  - 하위 레이어로 갈수록 추상화가 심화됨
  - 상위 레이어로 갈수록 비즈니스 로직이 심화됨

Slices

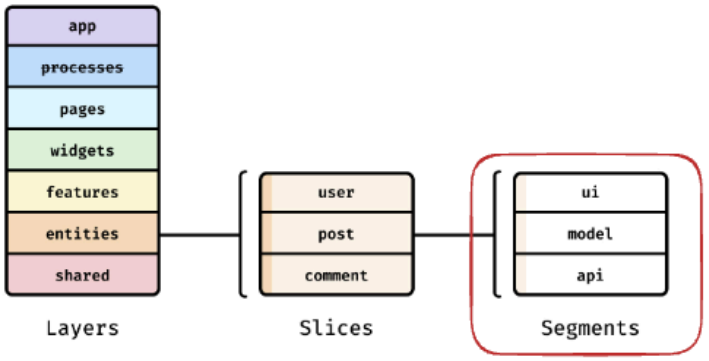
> 슬라이스는 FSD에서 각 계층(layer) 내의 비즈니스 도메인별 분할을 의미함.
>
> 슬라이스의 이름이 가져야 할 특별한 규칙은 없으며, 비즈니스 도메인에 따라 명명.



- 슬라이스 이름은 명사 단수형으로 사용
- 슬라이스 간 직접 의존성 최소화
- 각 슬라이스는 명확한 경계와 책임을 가져야 함
- 모든 계층(layer)에 동일한 슬라이스 구조 적용
- 비즈니스 도메인별로 구분하여 명확한 경계 설정
- app 레이어: 애플리케이션 전체적인 부분과 연관된 코드를 포함, 보통 슬라이스 미사용
- shared 레이어: 깊은 추상화를 가지고 비즈니스 로직이 없어 슬라이스 미사용
- 모든 코드는 ui, model, api 등의 세그먼트에 배치해야 함

Segments

> FSD의 마지막 계층에 있는 요소
> 특정 도메인 내에서 기술적 결과를 달성하기 위한 코드 분류



주요 세그먼트 유형

- `ui/` : UI 컴포넌트만 포함하며, 비즈니스 로직은 최소화
- `model/` : 상태 관리, 비즈니스 로직, 타입 정의 포함
- `api/` : API 요청 및 데이터 변환 로직
- `lib/` : 도메인 특화 유틸리티 함수
- `config/` : 도메인 특화 상수 및 설정

2. 계층(Layers)

> 프로젝트의 루트 구조는 다음과 같이 구성

```
src/
├── app/              # 애플리케이션 초기화 및 글로벌 설정
├── pages/            # 페이지 컴포넌트
├── widgets/          # 복합적인 UI 블록
├── features/         # 사용자 액션과 비즈니스 로직
├── entities/         # 비즈니스 엔티티
└── shared/           # 공유 유틸리티 및 UI 키트
```

2.1 app

> 애플리케이션 로직이 초기화되는 곳. 프로바이더, 라우터, 전역 스타일, 전역 타입 선언 등 정의

```
app/
├── providers/        # 애플리케이션 프로바이더 (React Query, Auth, Theme 등)
├── styles/           # 글로벌 스타일
├── routes/           # 라우트 정의
├── types/            # 애플리케이션 전역 타입
└── index.ts          # 진입점
```

책임:

- 애플리케이션 초기화
- 전역 프로바이더 설정

- 라우팅 구성
- 글로벌 스타일 및 테마
- 환경 변수 및 앱 설정

2.2 pages

> 하위 레이어들을 조합하여 완전한 기능을 제공하는 레이어

```
pages/
├── mypage/
│   ├── customer/
│   └── company/
├── bid/              # 입찰 관련 페이지
└── index.ts          # 페이지 public API
```

책임:

- 라우트에 매핑되는 진입점 제공
- 복잡한 로직은 포함하지 않고 위젯, 피처, 엔티티 조합
- 페이지별 레이아웃 구성
- URL 파라미터 처리

2.3 widgets

> 여러 피처와 엔티티를 조합한 독립적인 UI 블록

```
widgets/
├── header/           # 헤더 위젯
├── sidebar/          # 사이드바 위젯
├── layouts/          #
│   ├── CustomerLayout/
│   │   ├── index.tsx
│   │   └── styles.module.css
│   └── CompanyLayout/
│       ├── index.tsx
│       └── styles.module.css
```

```
├── orderForm/        # 주문 폼 위젯
│   ├── ui/           # 위젯 UI 컴포넌트
│   ├── model.ts      # 위젯 로직
│   └── index.ts      # 위젯 public API
└── index.ts          # 위젯 public API
```

책임:

- 복합적인 UI 블록 구성
- 여러 피처와 엔티티 조합
- 독립적인 비즈니스 로직 캡슐화
- 재사용 가능한 UI 패턴 제공

2.4 features

사용자 액션과 관련된 비즈니스 로직을 담당합니다.

```
features/
├── auth/             # 인증 관련 기능
│   ├── signin/
│   ├── signup/
│   └── index.ts      # 인증 public API
├── order/            # 주문 관련 기능
│   ├── createOrder/
│   ├── cancelOrder/
│   └── index.ts      # 주문 public API
└── index.ts          # 피처 public API
```

각 피처 내부 구조:

```
features/auth/signin/
├── ui/               # UI 컴포넌트
├── model/            # 비즈니스 로직 (스토어, 커스텀 훅, 타입 등)
├── api/              # API 요청
├── lib/              # 유틸리티 함수
│   ├── index.ts
│   ├── types.tx
```

```
│   └── utils.ts
└── index.ts          # signin 피처 public API
```

책임:

- 사용자 액션(유스케이스) 구현
- 액션 관련 비즈니스 로직 캡슐화
- 도메인 특화 UI 컴포넌트 제공
- 피처별 상태 관리

2.5 entities

비즈니스 엔티티(예: 유저, 주문, 입찰 등)를 표현합니다.

```
entities/
├── user/             # 사용자 엔티티
│   ├── ui/           # UI 컴포넌트
│   ├── model/        # 도메인 모델
│   ├── api/          # API 요청
│   └── index.ts      # 엔티티 public API
├── order/            # 주문 엔티티
├── bid/              # 입찰 엔티티
└── index.ts          # 엔티티 public API
```

책임:

- 비즈니스 엔티티 구조 및 타입 정의
- 엔티티 관련 UI 컴포넌트 제공
- 엔티티 CRUD 작업 캡슐화
- 엔티티별 상태 관리 및 캐싱

2.6 shared

> 애플리케이션 전반에서 공유되는 인프라 코드, UI 키트, 유틸리티 등을 포함.

```
shared/
├── api/            # API 클라이언트 및 유틸리티
├── ui/             # UI 컴포넌트 키트
├── lib/            # 유틸리티 함수
├── config/         # 구성 상수
├── model/          # 비즈니스 로직 (스토어, 커스텀 훅 등)
└── index.ts        # shared public API
```

책임:

- 기본 UI 컴포넌트 키트 제공
- 공통 유틸리티 및 헬퍼 함수
- 인프라 추상화(API 클라이언트, 로깅 등)
- 전역 타입 및 상수 정의

3. Public API 원칙

3.1 Public API의 목적

> 공개 API는 각 모듈(슬라이스나 세그먼트)이 외부에서 사용할 수 있는 것들을 명확하게 선언하는 역할을 합니다. 이는 "이 모듈 안에서 외부에 보여주고 싶은 것만 공개하겠다"는 약속임.

3.2 Public API 규칙

- **원칙 1**: 앱의 Slice와 Segment들은 Public API index 파일에 정의된 기능과 컴포넌트만 사용한다
- **원칙 2**: Public API에 정의되지 않은 내부적인 부분은 격리된 것으로 간주하여 그 자신의 Slice나 Segment만 여기에 접근할 수 있다

3.3 캡슐화를 통한 이점

- **캡슐화(Encapsulation) 확보**: 불필요하게 노출할 필요가 없는 부분은 격리
- **안전한 리팩토링**: 특정 기능이나 컴포넌트의 내부 코드를 외부에서 수정할 필요 없이 마음 편히 리팩토링 가능

- **명확한 인터페이스**: 외부에서는 index.ts 파일만 사용하므로 슬라이스 안의 구조를 직접 알 필요 없음

3.4 Index.ts 파일 구조 예시

```
// features/auth/signin/index.ts
export { SigninForm } from './ui/SigninForm';
export { useSignin } from './model/useSignin';
export type { SigninCredentials } from './model/types';

// 내부 구현은 외부에 노출하지 않음
// - ./lib/validation.ts
// - ./api/signinApi.ts (private)
```

4. 레이어 간 구분 가이드

4.1 실제 적용 시 구분 기준

> widgets과 features, entities의 구분이 모호해지는 경우를 방지하기 위한 구체적인 가이드

widgets vs features vs entities 구분법

- **widgets** → pages 컴포넌트에서 조합하여 사용하기 위한 거의 완성된 독립 기능들
- **features** → widgets를 구성하기 위한 비즈니스 로직의 구체적인 표현 기능들 (결제, 재생, 좋아요, 환불 등)
- **entities** → features에서 구체적인 동작이 부여되기 전인 비즈니스 주체들 (유저, 비디오, 라이크버튼 등)
- **shared** → 특정 비즈니스 로직에 속하지 않는 재사용 대상들 (AxiosInstance, types, ButtonBase, layout 등)

실제 예시: Fork 버튼 구현

```
// 계층별 역할 분담 예시
<Shared.Button
  onClick={forkFeature.api.fork}
  icon={shared.icon.fork}
```

```
  data={forkEntity.model.forkCount}
/>
```

1. **fork에 대한 행위**: `features`에 구현
2. **fork에 대한 entities 컴포넌트**: `entities`에서 데이터 제공
3. **UI를 담당하는 Button 컴포넌트**: `shared`에서 제공

5. 단계적 적용 전략

5.1 초심자 권장 레이어

> FSD 적용 시 처음에는 ✅ 표시된 4개 레이어만 사용하는 것을 권장

1. **app** ✅ - 애플리케이션 초기화
2. **pages** ✅ - 페이지 컴포넌트
3. **features** ✅ - 핵심 비즈니스 기능
4. **shared** ✅ - 공통 컴포넌트 및 유틸리티

5.2 점진적 도입 방법

1. **기존 UI 분류**: 모든 UI를 widgets와 pages에 일단 배분
2. **로직 분리**: features와 entities를 분리하여 page와 widget을 순수하게 합성된 레이어로 변경
3. **정확성 향상**: 점진적으로 분리의 정확성을 늘려가기

5.3 적용 시기

- **작은 규모나 운영기간이 짧은 프로젝트**: 모놀리스 형식으로 충분
- **중규모 이상이거나 확장 예정인 프로젝트**: FSD 적용 권장
- **기존 프로젝트**: 점진적으로 마이그레이션 가능