**리액트 아키텍처 패턴**

**소프트웨어 아키텍처와 디자인 패턴**

* **소프트웨어 아키텍처란** 소프트웨어 시스템의 구조를 정의하고 설계하는 과정입니다.
* ﻿﻿이는 시스템의 구성 요소 간의 상호작용 방식, 데이터 처리 방법, 사용자 인터페이스와의 상호작용 방식 등을 포함합니다.
* ﻿﻿아키텍처는 시스템의 성능, 안정성, 보안성 등 전반적인 품질 속성에 큰 영향을 미칩니다.
* ﻿﻿**아키텍처 패턴은** 특정 문제를 해결하기 위해 반복적으로 사용할 수 있는 설계 패턴이며, 이는 소프트웨어 아키텍처를 구성하는 핵심 요소 중 하나입니다.

﻿﻿아키텍쳐 패턴은 공통된 설계 문제를 해결하는 방법을 제시하고, 이를 통해 코드의 재사용성, 유지보수성, 확장성을 향상시킵니다.

**단순한 프로젝트 구조의 문제점**

**1. 구조화와 유지보수성의 부족**

* ﻿﻿단일 컴포넌트 패턴만 사용하여 대규모 애플리케이션을 개발한 경우, 모든 기능을 하나의 컴포넌트에 집어넣어 코드가 길어지고 복잡해집니다.
* ﻿﻿각 기능을 독립적으로 테스트하거나 변경하기 어려워집니다.
* ﻿﻿또한, UI 로직과 비즈니스 로직이 섞이면서 코드의 가독성이 저하되고 유지보수가 어려워집니다.

**2. 상태 관리의 어려움**

* ﻿﻿React 컴포넌트 간에 직접적으로 상태를 전달하거나 상위 컴포넌트에서만 상태를 관리하는 경우, Props drilling이 발생하거나 상태가 여러 컴포넌트에 분산되어 있을 수 있습니다.

﻿﻿이로 인해 데이터의 일관성 유지가 어렵고, 디버깅과 테스트가 어려워질 수 있습니다.

**단순한 프로젝트 구조의 문제점**

**3. 비즈니스 로직과 UI 로직의 혼합**

* ﻿﻿비즈니스 로직과 UI 로직 모두를 단일 컴포넌트에 포함한 경우, 비즈니스 로직과 이 로직이 혼합되어 코드의 재사용성이 저하됩니다.
* ﻿﻿동일한 로직을 여러 컴포넌트에서 중복해서 작성해야 할 수 있으며, 이는 코드의 중복과 복잡성을 증가시킵니다.
* ﻿﻿코드의 중복과 복잡성은 프로젝트의 유지보수성을 떨어뜨리고, 확장성을 제한할 수 있습니다.
* ﻿﻿또한 이러한 점들 때문에, 새로 합류한 개발자나 다른 팀원이 코드를 이해하고 수정하기 어려울 수 있습니다.
* ﻿﻿이와 같은 문제들은 애플리케이션의 복잡성과 규모가 커질수록 심각해질 수 있으므로, 적절한 아키텍쳐 패턴을 선택하는 것이 중요합니다.

﻿﻿패턴을 적용함으로써 코드의 구조화와 관리가 개선되어 유지보수성과 확장성을 높이며, 팀 내 협업도 원활하게 할 수 있습니다.

**React에서의 아키텍쳐 패턴의 중요성**

**코드 구조화와 모듈화**

* ﻿﻿컴포넌트 기반 접근 방식과 함께 아키텍쳐 패턴은 코드를 논리적으로 구조화하고 모듈화할 수 있는 방법을 제공합니다.
* ﻿﻿이는 코드의 가독성을 높이고 유지보수를 용이하게 만듭니다.

**유지보수성 및 확장성 향상**

* ﻿﻿각각의 아키텍쳐 패턴은 특정 문제를 해결하도록 설계되어 있어, 시스템이 변화하거나 성장할 때 쉽게 적용할 수 있습니다.
* ﻿﻿이는 시스템의 유지보수성을 높이고, 새로운 요구사항에 대응할 수 있는 확장성을 제공합니다.

**개발자 팀 간의 협업 효율성 증대**

* ﻿﻿일관된 설계 패턴을 사용하면 개발자 팀 간의 협업이 효율적으로 진행될 수 있습니다.

﻿﻿각자가 이해할 수 있는 구조와 코드 스타일을 제공함으로써, 개발 프로세스의 일관성을 유지할 수 있습니다.

**React에서의 아키텍쳐 패턴 종류**

* ﻿﻿**단일 컴포넌트 패턴** (Single Component Pattern)
* ﻿﻿**컨테이너 / 프레젠테이셔널 패턴** (Container / Presentational Pattern)
* ﻿﻿**НОС 패턴** (Higher-Order Component Pattern)
* ﻿﻿**Flux 아키텍처** (Flux Architecture)
* ﻿﻿**모듈형 아키텍쳐** (Modular Architecture)
* ﻿﻿**FSD 아키텍쳐** (Feature Sliced Design Architecture)
* ﻿﻿이 외에도 다양한 종류의 아키텍쳐 패턴 종류가 존재

> 여러 아키텍처와 패턴을 숙지하고 있다가, 각 프로젝트의 상황에 따라서 선택

**단일 컴포넌트 패턴 (Single Component Pattern)**

* ﻿﻿단일 컴포넌트 패턴은 React에서 컴포넌트를 매우 단순하게 사용하는 패턴을 말합니다.
* ﻿﻿이 패턴에서는 하나의 컴포넌트가 UI의 모든 부분을 담당하며, 별도의 하위 컴포넌트를 사용하지 않거나 최소한의 컴포넌트만 사용합니다.
* ﻿﻿주로 작은 규모의 애플리케이션이나 간단한 기능을 구현할 때 사용될 수 있습니다.

**문제점**

* ﻿﻿코드의 재사용성 감소
* ﻿﻿유지보수의 어려움
* ﻿﻿성능 최적화의 한계

﻿﻿테스트의 어려움

**컨테이너 / 프레젠테이셔널 패턴 (Container / Presentational Pattern)**

* ﻿﻿컨테이너 / 프레젠테이셔널 패턴은 컴포넌트의 역할에 따라 컨테이너 혹은 프레젠테이셔널 컴포넌트로 구분하여 관리하는 방법입니다.
* ﻿﻿이 패턴은 컴포넌트의 역할을 명확히 구분함으로써 코드의 재사용성, 유지보수성, 테스트 용이성을 향상시키는 데 중점을 둡니다.

**컨테이너 컴포넌트**

* ﻿﻿컨테이너 컴포넌트는 주로 상태 관리나 데이터 로직을 처리하는 역할을 맡습니다.
* ﻿﻿일반적으로 ApI와의 통신이나 상태 관리 라이브러리와의 연결을 담당합니다.

**프레젠테이셔널 컴포넌트**

* ﻿﻿프레젠테이셔널 컴포넌트는 주로 U를 표시하고 사용자와의 인터랙션을 담당합니다.

﻿﻿상위로부터 받은 props를 받아와 UI를 렌더링하며, 상태나 데이터를 직접적으로 관리하지 않습니다.

**프레젠테이셔널 컴포넌트 특성**

* ﻿﻿**UI표시:** UI를 정의하고 렌더링합니다.
* ﻿﻿**상태 관리 X**: props를 통해 받은 데이터를 표시하거나 사용자와 상호작용을 담당합니다.
* ﻿﻿**재사용성**: 독립적이고 재사용 가능한 UI 요소로 설계됩니다.

**패턴의 장점**

* ﻿﻿**코드의 분리와 구조화**: 컨테이너와 프레젠테이셔널 컴포넌트를 명확히 구분함으로써 코드의 역할과 책임을 명확히 할 수 있습니다.
* ﻿﻿**재사용성 증가**: 프레젠테이셔널 컴포넌트는 독립적이며 재사용 가능하기 때문에 다양한 기능과 컨테이너에서 사용할 수 있습니다.

﻿﻿**유지보수성 향상:** 코드의 수정이나 버그 수정이 더 쉬워지며, 특히 UI의 변화가 필요할 때 컨테이너와 프레젠테이셔널 컴포넌트를 분리함으로써 영향을 줄일 수 있습니다.

**НОС 패턴 (Higher-Order Component Pattern)**

* ﻿﻿HOC 패턴은 React에서 컴포넌트 로직을 재사용하기 위한 패턴입니다.
* ﻿﻿고차 컴포넌트는 컴포넌트를 인자로 받아 새로운 컴포넌트를 반환합니다.
* ﻿﻿이를 통해 기존 컴포넌트의 기능을 확장하거나 새로운 기능을 추가할 수 있습니다.
* ﻿﻿일반적으로 함수 이름 앞에 **with** 를 붙여 구현합니다.

**주요 용도**

* ﻿﻿**상태 관리 및 로직 추가**: HOC를 사용하여 상태 관리 로직이나 라이프사이클 메서드 등을 추가할 수 있습니다.
* ﻿﻿**인증 및 권한 관리:** 인증 여부를 확인하고 필요한 권한을 가진 사용자만 접근할 수 있도록 제어할 수 있습니다.

﻿﻿**코드 재사용성:** 여러 컴포넌트에서 동일한 기능을 재사용할 수 있어 코드 중복을 방지하고 유지보수성을 높입니다.

**HOC 패턴 사용 시 주의사항**

* ﻿﻿**컴포넌트 구조의 복잡성:** 과도한 사용은 컴포넌트의 구조를 복잡하게 만들 수 있으므로 필요한 경우에만 사용하는 것이 좋습니다.
* ﻿﻿**명확한 컴포넌트 구분:** HOC를 사용할 때는 컴포넌트의 역할과 책임을 명확히 구분해야 합니다. 너무 많은 로직을 한 번에 처리하려고 하지 않는 것이 중요합니다.

﻿﻿**컴포넌트 트리 구조 이해:** HOC를 사용할 때 컴포넌트 트리의 구조를 고려해야 합니다. HOC는 특정 컴포넌트의 래핑을 통해 구현되므로, 트리 구조에서 어떻게 동작하고 렌더링되는지 이해하는 것이 중요합니다

**Flux 아키텍처**

* ﻿﻿Flux는 Facebook에서 클라이언트 측 웹 애플리케이션을 구축하기 위해 설계된 애플리케이션 아키텍처입니다.
* ﻿﻿이 패턴은 데이터가 어떻게 애플리케이션 내에서 흐르는지를 명확히 정의하여 상태 관리를 용이하게 합니다.

**데이터 흐름**

* ﻿﻿Flux는 단방향 데이터 흐름을 따릅니다.
* ﻿﻿사용자의 상호작용은 View에서 Action으로 변환되고, Dispatcher를 통해 모든 Store에게 전달됩니다.
* ﻿﻿각각의 Store는 Dispatcher에 등록된 콜백을 통해 Action을 처리하고, 필요에 따라 상태를 변경하고 이벤트를 발생시켜 View에게 새로운 데이터를 전달합니다.

﻿﻿이러한 방식은 데이터의 예측 가능성을 높이고, 복잡한 의존성과 상호작용을 줄이는 데 유리합니다.

**Flux 구성 요소**

**Dispatcher**

* ﻿﻿애플리케이션 내에서 모든 데이터의 흐름을 중앙에서 관리합니다.
* ﻿﻿이 요소는 action을 받아 등록된 모든 store에게 전달하는 역할을 합니다.

**Stores**

* ﻿﻿애플리케이션의 상태와 데이터 로직을 포함하고 있습니다.
* ﻿﻿Store는 상태를 변경할 때 마다 View에게 알리는 역할을 하며, 이로 인해 View는 데이터가 변경될 때마다 자동으로 업데이트될 수 있습니다.

**Views (React Components)**

* ﻿﻿Store로부터 데이터를 받아와 화면에 렌더링합니다.
* ﻿﻿사용자의 상호작용을 Dispatcher를 통해 Action으로 변환하여 데이터의 변경을 요청할 수 있습니다.

**Controller-Views**

* ﻿﻿Store로부터 데이터를 받아 하위 View에 전달하는 역할을 합니다.

﻿﻿이를 통해 위계의 최상위에서 데이터의 흐름을 관리할 수 있으며, 상태의 변화가 필요한 경우 다시 자식 View에게 전달하여 UI를 업데이트합니다.

**모듈형 아키텍쳐 (Modular Architecture)**

* ﻿﻿모듈형 아키텍쳐는 소프트웨어 개발에서 코드를 모듈 단위로 분리하고 구성하여, 코드의 재사용성, 유지보수성, 확장성을 높이는 아키텍처적 접근 방식입니다.
* ﻿﻿모듈은 독립적으로 테스트하고 배포할 수 있는 단위로, 비즈니스 로직이나 기능별로 분리되어야 합니다.
* ﻿﻿일종의 객체 지향 프로그래밍을 위한 아키텍쳐로 볼 수 있습니다.

**주요 개념**

* **﻿﻿모듈**
* ﻿﻿모듈은 애플리케이션 내에서 특정 기능이나 독립적인 기능 단위를 의미합니다.
* ﻿﻿각 모듈은 자체적으로 완전한 기능을 제공하며, 다른 모듈과 독립적으로 개발, 테스트, 배포할 수 있어야 합니다.
* ﻿﻿**단일 책임 원칙**
* ﻿﻿각 모듈이 한 가지 기능 또는 책임을 가지도록 설계해야 합니다.

﻿﻿이는 코드의 응집성을 높이고, 유지보수성을 향상시키는 데 중요한 역할을 합니다.

**모듈형 아키텍쳐 사용시 주의사항**

* ﻿﻿**의존성 관리:** 각 모듈은 필요한 경우 외부에서 제공되는 인터페이스를 통해 의존성을 관리해야 합니다. 의존성 주입 패턴 등을 사용하여 결합도를 낮추는 것이 좋습니다.
* ﻿﻿**테스트와 배포:** 각 모듈은 독립적으로 테스트와 배포가 가능해야 합니다. 이를 위해 CI/CD 파이프라인을 설정하여 각 모듈을 자동으로 빌드하고 배포하는 것이 좋습니다.

﻿﻿**명명 규칙:** 모듈명과 파일명은 각 모듈의 기능을 명확히 반영하도록 명명하는 것이 중요합니다. 이는 프로젝트 전체의 가독성과 유지보수성을 높이는 데 기여합니다.

**모듈형 아키텍쳐와 OOP의 SOLID 원칙**

* ﻿﻿**단일 책임 원칙** (Single Responsibility): 각 모듈은 단일 책임 원칙에 따라 특정 기능 또는 관심 영역에 집중해야 합니다. 이는 모듈의 응집성을 높이고, 유지보수성을 향상시킵니다.
* ﻿﻿**개방-폐쇄 원칙** (Open-closed): 모듈은 확장에 열려 있어야 하고, 수정에는 닫혀 있어야 합니다. 새로운 기능이 추가될 때 기존 코드를 수정하지 않고도 확장할 수 있도록 설계해야 합니다.
* ﻿﻿**리스코프 치환 원칙** (Liskov Substitution): 모듈 간의 상속 관계에서는 하위 모듈이 상위 모듈을 대체할 수 있어야 합니다. 인터페이스와 추상화를 통해 일관된 동작을 보장하는 것이 중요합니다.
* ﻿﻿**인터페이스 분리 원칙** (Interface Segregation): 클라이언트는 자신이 필요로 하는 인터페이스만을 이용할 수 있어야 합니다. 이는 모듈 간의 결합도를 낮추고 각 모듈이 독립적으로 발전할 수 있도록 합니다.

﻿﻿**의존성 역전 원칙** (Dependency Inversion): 모듈은 추상화(인터페이스) 에 의존해야 하며, 구체적인 구현에 의존해서는 안 됩니다. 이는 모듈 간의 결합도를 줄이고 유연한 설계를 가능하게 합니다.

**FSD 아키텍쳐 (Feature Sliced Design Architecture)**

* ﻿﻿최근 등장한 FSD 아키텍쳐(Feature Sliced Design Architecture) 는 모듈형 아키텍쳐에서 변형된 아키텍쳐입니다.
* ﻿﻿주로 대규모 프로젝트에서 사용되며 특히 팀 간 협업과 유지보수성을 강화하는 데 중점을 둡니다.
* ﻿﻿이 아키텍쳐는 각기 다른 기능이나 기능 그룹을 독립적인 단위로 분리하여 개발하고 관리하는 방법을 제안합니다.

﻿﻿이 아키텍쳐는 계속 변화하는 비즈니스 요구사항에도 안정적으로 대응할 수 있도록 설계되었습니다.

**FSD 아키텍쳐의 주요 특징**

* ﻿﻿**계층적 구조**
* ﻿﻿FSD는 프로젝트를 명확하게 구분된 여러 계층으로 구성합니다.
* ﻿﻿이러한 계층 구조는 모듈화된 개발을 촉진하고 의존성을 줄이며 코드의 명확성을 제고합니다.
* ﻿﻿**슬라이스와 세그먼트**
* ﻿﻿각 계층은 슬라이스(Slice)로 나누어지며, 비즈니스 도메인에 따라 코드를 파티션화합니다.
* ﻿﻿슬라이스는 서로 다른 기능 영역을 유지하면서 코드베이스를 쉽게 탐색할 수 있도록 합니다.

﻿﻿슬라이스와 함께 각 계층은 기술적인 목적에 따라 세그먼트(Segment)로 구성됩니다.

**FSD 아키텍쳐의 계층적 구조**

* ﻿﻿**App:** 라우팅, 진입점, 전역 스타일, 프로바이더 등 애플리케이션을 실행하는 데 필요한 기본적인 요소들을 포함합니다.
* ﻿﻿**Pages:** 전체 페이지 또는 중첩된 라우팅 내의 주요 섹션을 나타냅니다.
* ﻿﻿**Widgets:**다시 말해, 사용자에게 제공하는 기능이나 화면 요소를 완전히 독립적으로 구성한 컴포넌트를 포함합니다.
* ﻿﻿**Features**: 비즈니스 가치를 제공하는 완전한 제품 기능의 재사용 가능한 구현을 포함합니다.
* ﻿﻿**Entities:** 사용자 또는 제품과 같은 핵심 비즈니스 개체를 나타냅니다.

﻿﻿**Shared:** 프로젝트 특정 세부 정보에서 독립적인 재사용 가능한 기능들을 포함합니다.

**FSD의 슬라이스와 세그먼트**

* ﻿﻿슬라이스(Slice)
* ﻿﻿슬라이스는 FSD에서 두 번째 수준의 조직적 계층입니다.
* ﻿﻿슬라이스의 주된 목적은 제품, 비즈니스 또는 애플리케이션의 의미에 따라 코드를 그룹화하는 것입니다.
* ﻿﻿슬라이스의 이름은 애플리케이션의 비즈니스 도메인에 따라 결정되므로 표준화되지 않습니다.
* ﻿﻿슬라이스 내부에서는 비즈니스와 연관된 로직을 작성할 수 있습니다.
* ﻿﻿슬라이스 내부에서는 여러 그룹의 세그먼트를 가질 수 있으나 특정 코드를 세그먼트를 만들지 않고 배치하지 않습니다.
* ﻿﻿**세그먼트(Segments)**
* ﻿﻿세그먼트는 조직적 계층의 세 번째이자 마지막 수준이며, 코드의 기술적 특성에 따라 코드를 그룹화하는 목적을 가집니다.

﻿﻿세그먼트의 이름은 ui, model, lib, api등이 될 수 있습니다.

**FSD 아키텍쳐의 장점**

* ﻿﻿**일관성 유지**: 표준화된 구조로 인해 프로젝트의 일관성이 유지되며, 새로운 팀원이 프로젝트에 합류해도 빠르게 적응할 수 있습니다.
* ﻿﻿**변화와 리팩터링에 대한 안정성**: 한 계층의 모듈이 같은 계층의 다른 모듈에 의존하지 않기 때문에 변경 사항이 다른 부분에 미치는 영향을 사전에 방지할 수 있습니다.
* ﻿﻿**로직의 제어된 재사용**: 각 계층에 따라 코드를 매우 재사용 가능하게 하거나 특정한 범위 내에서 제한하는 균형을 유지합니다.
* ﻿﻿**비즈니스와 사용자 중심의 지향**: 비즈니스 도메인과 사용자 요구에 집중하여 애플리케이션을 분할하고, 프로젝트 내에서 관련 없는 부분에 대해 완전한 이해 없이도 유용한 제품 작업을 수행할 수 있도록 합니다.
* ﻿﻿**점진적 채택**: 기존 코드베이스를 FSD로 이전할 때는 점진적으로 접근하는 것이 좋습니다. 이는 프로젝트의 특정 부분만을 수정하고 발전시키면서 새로운 구조를 구현하는 방식입니다.

> **FSD는 특히 큰 규모의 프론트엔드 프로젝트에서 효과적이며, 설계의 간결성과 유연성을 제공하여 팀의 협업과 소통을 강화합니다**.

**Controlled vs. Uncontrolled Components**

* ﻿﻿React에서 폼 관리 패턴은 사용자 입력을 다루는 방식에 따라 크게 Controlled Components® Uncontrolled Components 로 나눌 수 있습니다.

﻿﻿각각의 방식에는 고유한 장단점이 있으며, 상황에 따라 적합한 방법을 선택할 수 있습니다.

Controlled vs. Uncontrolled Components

1. **Controlled Components**

* ﻿﻿Controlled Components는 React 컴포넌트가 입력 필드의 상태를 제어하는 방식입니다.
* ﻿﻿입력 필드의 값은 컴포넌트의 상태로 관리되며, 사용자가 입력을 변경할 때마다 상태가 업데이트됩니다.

**툭징**

* ﻿﻿폼 요소의 값은 컴포넌트의 state로 관리됩니다.
* ﻿﻿값이 변경될 때마다 onChange 이벤트 핸들러를 통해 상태를 업데이트합니다.

**• 장점**

* ﻿﻿데이터 흐름이 단방향으로 명확해지므로 디버깅이 용이합니다.
* ﻿﻿폼의 유효성 검사 및 동적 변경이 쉽습니다.
* ﻿﻿상태를 중앙에서 관리할 수 있어 보다 일관된 상태 관리를 할 수 있습니다.

**단점**

* ﻿﻿폼 요소가 많을 경우 다소 코드가 장황해질 수 있습니다.

﻿﻿매 입력마다 상태 업데이트가 발생하여 성능에 영향을 줄 수 있습니다.

1**. Uncontrolled Components**

* ﻿﻿Uncontrolled Components는 폼 요소 자체가 상태를 관리하는 방식입니다.
* ﻿﻿React는 폼 요소의 값에 직접 접근하지 않으며, 필요할 때만 참조를 통해 값을 가져옵니다.
* ﻿﻿**특징**
* ﻿﻿폼 요소의 값은 컴포넌트 상태로 관리되지 않고 DOM에서 직접 관리됩니다.
* ﻿﻿React의 ref를 사용하여 폼 요소에 접근합니다.
* ﻿﻿초기값만 React를 통해 설정되고, 이후에는 폼 자체에서 값을 관리합니다.
* **장점**
* ﻿﻿초기값만 설정하고 이후 관리가 필요 없으므로 코드가 간결해집니다.
* ﻿﻿폼 요소가 많은 경우 성능상의 이점이 있을 수 있습니다.
* ﻿﻿**단점**
* ﻿﻿상태를 관리하는 데 있어 유연성이 떨어집니다.

﻿﻿폼의 유효성 검사 및 동적 변경이 어려울 수 있습니다.