컴포넌트 설계와 모노레포 전략

1. 리액트 컴포넌트 설계

❖컴포넌트의 정의

- 애플리케이션에서 독립적인 기능을 수행하는 재사용이 가능한 구성요소

❖리액트에서의 컴포넌트

- "Components let you split the UI into independent, reusable pieces, and think about each piece in isolation." – 리액트 공식페이지
- "독립성, 재사용성, 고립성을 가진 UI 조각"

❖리액트 컴포넌트 설계 원칙

- 확장성, 재사용성
- 단일 책임 원칙
- 렌더링 최적화

1. 리액트 컴포넌트 설계

❖컴포넌트 설계 단계

- 1. 화면 설계
  - 리액트는 웹애플리케이션의 UI를 책임지는 프레임워크이므로 화면 설계가 진행되어야 함
  - 산출물 : 화면 시안, 화면 설계서
- 2. 각 화면별로 상태, 액션 도출
  - 상태
    - 각 화면에서 다룰 중요한, 관리가 필요한 데이터를 도출
    - 상태로 다루지 말아야 예시
      - » 원본 데이터를 이용해 필터링한 데이터
      - » 사용자로부터 입력을 받기 위해 입력필드에 바인딩되는 상태 ---〉 하위 컴포넌트의 로컬 상태로 처리
  - 액션
    - 상태를 변경하는 기능.
    - 순수하게 상태를 변경하는 기능이 아닌 외부요소에 의존하는 작업이라면 기능 분리(예: 리액트 훅을 이용하여)가 필요함
    - 특히 백엔드 API에 의존하는 액션이라면
  - 상태와 액션이 백엔드 API에 의존한다면 백엔드 API의 요청/응답 형식을 고려하여 설계해야 함

1. 리액트 컴포넌트 설계

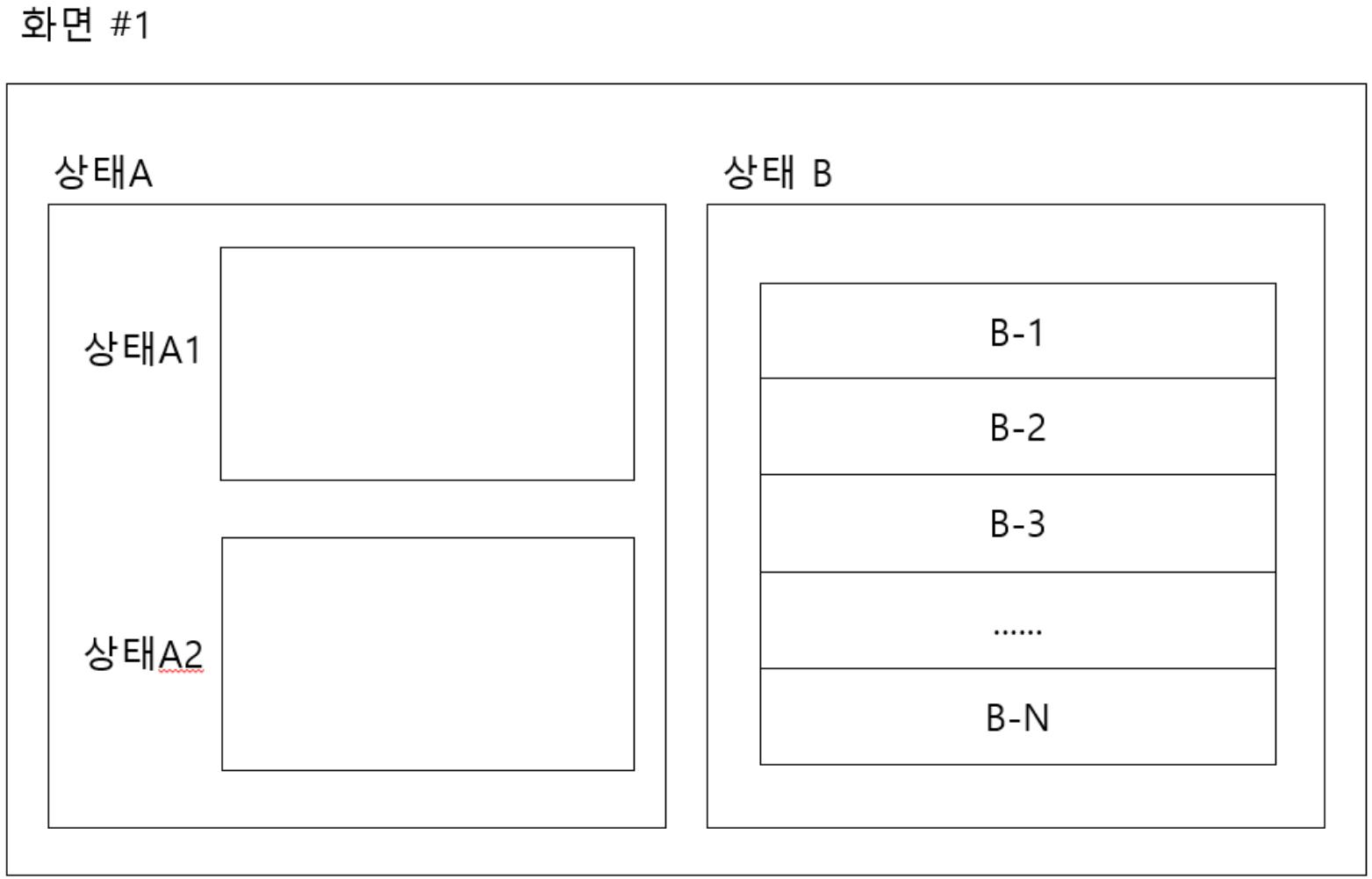
- 3. 애플리케이션 수준의 상태, 액션 도출
  - 여러 화면에서 사용하는 공통의 상태, 액션은 애플리케이션 수준의 상태 관리 기술을 적용할 수 있음
    - redux, recoil, zustand 등

- 4. 각 화면 별로 컴포넌트 분할
  - 단일 책임(Single Responsibility) 원칙을 준수하도록 분할함
    - 독립성, 재사용성, 테스트 용이성을 높일 수 있도록
    - 복잡한 기능은 고차함수, 리액트 훅으로 분리함

  - 렌더링 최적화를 고려하여 컴포넌트 분할
    - 렌더링 최적화가 필요한 화면에서만 고려함
    - 너무 작은 입자로 쪼갠 컴포넌트는 오히려 렌더링 최적화를 방해할 수 있음.
    - 데이터가 변화하는 단위까지 분할해줌

1. 리액트 컴포넌트 설계

- 렌더링 최적화를 고려한 컴포넌트 분할 예시



1. 리액트 컴포넌트 설계

- 5. 컨테이너 컴포넌트, 표현 컴포넌트 식별
  - 컨테이너 컴포넌트
    - Container Component
    - 상태, 상태 변경기능(액션)을 포함하는 컴포넌트
    - 주로 화면의 최상위 컴포넌트
      - » 한 화면이 레이아웃으로 구성되었다면 각 레이아웃의 최상위 컴포넌트
    - 자세한 UI보다는 상태, 기능에 집중
    - 2단계에서 식별한 복잡한 기능, 사이드 이펙트를 일으키는 기능은 별도의 훅으로 분리함.
  - 표현 컴포넌트
    - Presentation Component
    - 속성(props)를 전달받아 UI를 만들어내는 컴포넌트
    - 순수 컴포넌트 : Pure Component, 입력(props)이 동일하면 출력(리턴되는 JSX)이 동일함
  - 산출물 : 컴포넌트 리스트
    - 각 컴포넌트마다 필요한 상태, 액션, 속성, 사용할 훅 등을 명시해야 함
    - 무엇이 여러 화면에서 사용할 공통의 컴포넌트인지를 식별할 수 있어야 함

1. 리액트 컴포넌트 설계

- 6. 프로젝트 폴더 구조 정의
  - 정답이 없음.
    - 자주 변경되는 것 중심으로 묶어줄 것
    - 프로젝트 규모가 커지면서 언제든지 바뀔 수 있음
  - 다양한 방법
    - 파일 유형 별로 폴더 분리
      - » hooks, components, pages 와 같이 파일의 유형별로 분리함
      - » 아토믹 디자인 : 화면을 더이상 쪼갤 수 없을 때까지 분할하여 재사용성을 극대화함. 하지만 너무 많은 컴포넌트가 생성되어 오히려 복잡도를 증가시킬 수 있음
    - 기능, 라우트 경로별 폴더 정의
      - » 기능, 라우트별로 폴더를 만들고 관련 컴포넌트와 컴포넌트에서 필요로 하는 js, ts, css, test 코드를 배치함
      - » 가장 대표적인 사례 : FSD(Feature Sliced Design)
    - 프로젝트 규모를 고려하여 미리 선정하는 것이 바람직함

2. 리액트 폴더 구조

❖파일 유형별로 폴더 구성

- 예시) – 반드시 이 예시와 같은 이름의 폴더명을 사용할 필요는 없음
  - pages
  - components
  - hooks
  - contexts
  - services : 여러 컴포넌트, 훅, 컨텍스트에서 이용하는 기능을 정의

❖feature 폴더 사용

- components 폴더의 컴포넌트 중 여러 화면에서 재사용하는 컴포넌트는 별도의 feature 폴더를 생성하여 기능별로 컴포넌트를 분류할 수 있음

2. 리액트 폴더 구조

❖컴포넌트 파일명 Casing Rule

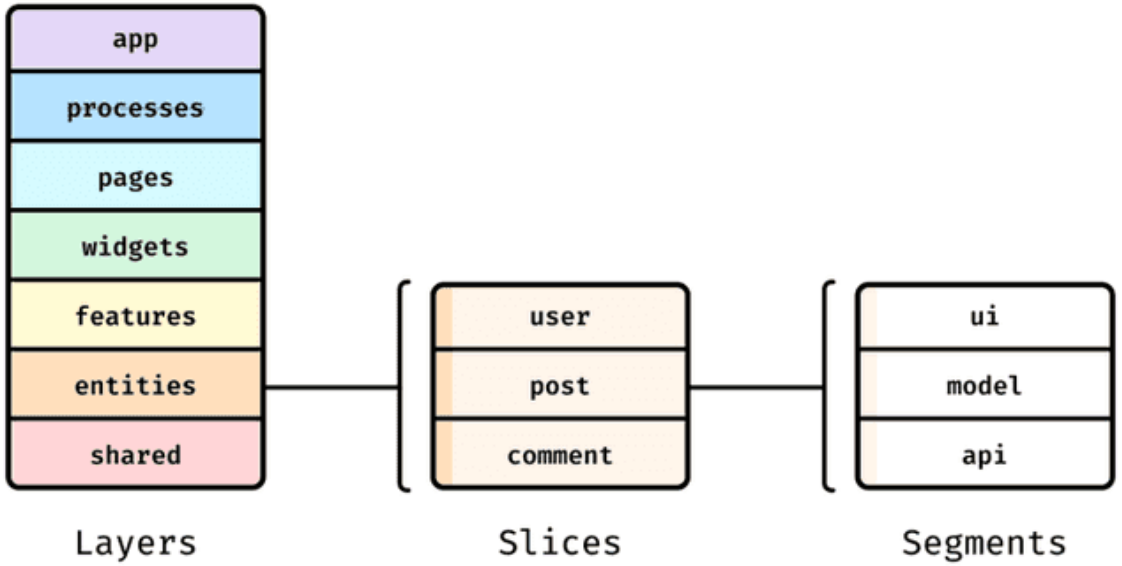
- PascalCase
  - 컴포넌트 파일 하나를 참조하여 여러개의 인스턴스로 만들 수 있으므로 java 등의 언어에 사용하는 타입처럼 간주
  - 대소문자를 구별하지 않는 개발 환경이 있을 수 있으므로 대소문자 구별 규칙을 준수할 수 있도록 강제해야 함
- kebob-case
  - 모두 소문자를 쓰며 새로운 단어가 시작하는 시점에 하이픈(-)기호를 넣어주는 방법
- 적절한 것을 선택해서 사용할 것

2. 리액트 폴더 구조

❖FSD란?

- Feature Sliced Design : 기능 분할 설계
- 최근 가장 널리 쓰이는 방법
- https://dev.to/m_midas/feature-sliced-design-the-best-frontend-architecture-4noj

❖FSD의 핵심 개념 : 레이어, 슬라이스, 세그먼트



2. 리액트 폴더 구조

❖레이어

- 프로젝트 내의 최상위 폴더, 애플리케이션 분해의 첫번째 단계
- 7개의 레이어
  - app
    - 애플리케이션 로직, Provider, Router, Global Style 등이 이 위치에서 정의됨.
    - 애플리케이션 진입점
  - processes : deprecated
    - Multi Step 사용자 등록과 같은 여러 페이지에 걸쳐서 처리하는 프로세스를 배치함
  - pages
    - 애플리케이션의 페이지를 포함함
  - widgets
    - 페이지에 사용되는 독립적인 UI 컴포넌트
  - features : 선택적
    - 비즈니스 로직을 처리하는 사용자 시나리오, 기능을 처리함
  - entities : 선택적
    - 비즈니스 엔티티. 예) 사용자, 블로그, 상품, 리뷰 등
  - shared
    - 특정 비즈니스 로직에 종속되지 않는 재사용가능한 컴포넌트, 유틸리티를 배치함

```
└── src/
    ├── app/
    ├── processes/ (deprecated)
    ├── pages/
    ├── widgets/
    ├── features/
    ├── entities/
    └── shared/
```

2. 리액트 폴더 구조

- 레이어는 상위, 하위 개념이 있어서 하위 레이어에서 상위 레이어의 기능을 이용할 수 없음
  - 계층 구조상 아래에 있으면 더 많은 영역에서 재사용될 수 있음

레이터	무엇을 이용할 수 있는가?	무엇에 의해 이용될 수 있는가?
app	processes, pages, widgets, features, entities, shared	-
processes	pages, widgets, features, entities, shared	app
pages	widgets, features, entities, shared	app, processes
widgets	features, entities, shared	app, processes, pages
features	entities, shared	app, processes, pages, widgets
entities	shared	app, processes, pages, widgets, features
shared	-	app, processes, pages, widgets, features, entities

2. 리액트 폴더 구조

❖슬라이스

- 각 레이어에서의 특정 비즈니스 엔티티를 중심으로 생성
- 슬라이스의 목표
  - 슬라이스 각각에서 다루는 값에 의해 코드를 그룹화하는 것
- 슬라이스 이름 : 비즈니스 도메인에 의해 지정
  - 예) SNS 애플리케이션 : 게시물, 사용자, Favorites, NewsFeed 등
- 밀접하게 연관된 조각들은 한 폴더에 구조적으로 그룹화할 수 있지만 다른 슬라이스와 동일한 격리 규칙을 준수해야 하며 직접적으로 공유되지 않음

```
└── src/
    ├── app/
    │   ├── providers/
    │   ├── styles/
    │   └── index.tsx
    ├── processes/
    ├── pages/
    │   ├── home/
    │   ├── profile/
    │   └── about/
    ├── widgets/
    │   ├── newsfeed/
    │   ├── catalog/
    │   ├── header/
    │   └── footer/
    ├── features/
    │   ├── user/
    │   ├── auth/
    │   ├── favorites/
    │   └── filter-users/
    ├── entities/
    │   ├── user/
    │   └── session/
    └── shared/
```

2. 리액트 폴더 구조

❖세그먼트

- 각 슬라이스는 목적에 따라 여러 세그먼트로 나눠질 수 있음
- 세그먼트의 이름은 팀내 구성원의 합의에 따라 결정될 수 있음
- 일반적으로 사용되는 세그먼트 이름
  - api : 필요한 서버 요청
  - UI : 슬라이스의 UI 컴포넌트
  - model : 비즈니스 로직, action, selector 등. 상태와의 인터랙션
  - lib : 슬라이스 내에서 사용되는 라이브러리
  - config : 슬라이스에 필요한 구성 설정 정보
  - consts : 상수

❖각 슬라이스와 세그먼트는 외부로 공개하기 위한 API가 필요한

- 각 디렉토리의 index.js, index.ts
- 이 파일을 통해 슬라이스, 세그먼트에서 필요한 기능만 외부로 공개함
  - index에 정의된 것 : 외부로 공개하는 것
  - index에 정의되지 않은 것 : 슬라이스 또는 세그먼트 내에서만 접근가능한 것

2. 리액트 폴더 구조

❖FSD가 문제를 해결하는 방법

- FSD의 과제 : 결합을 느슨하게 하고 응집력을 높이는 것
- 추상화와 다형성
  - 레이어를 통해 구현
  - 낮은 레이어 : 더 추상화되어 있음. 높은 레이어에서 재사용될 수 있음
- 캡슐화
  - 공개 API(index.js, index.ts)를 통해서 달성됨.

❖FSD를 적용하기 위해서...

- 팀의 구성원들이 아키텍처에 대한 더 높은 이해도와 기술 수준을 필요로 함

❖정답은 없음!!!

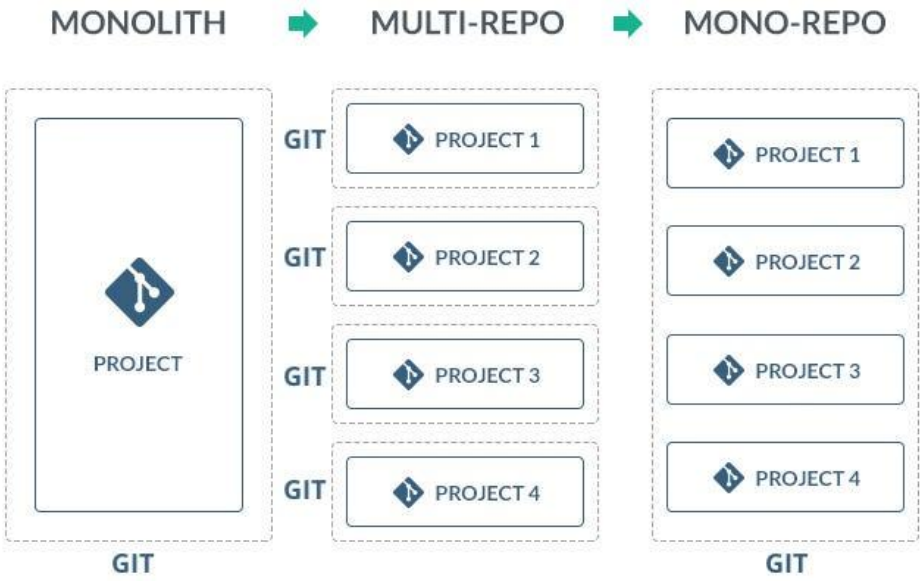
3. Repo 전략

❖Repo 전략이란?

- 애플리케이션 프로젝트를 어떤 구조의 Repository 로 관리할 것인가를 선택하는 전략
- 종류 : Multi Repo, Mono Repo

❖모놀리식 애플리케이션(단일 레포)

- 개념
  - 설계시에 모듈화가 적용되지 않은 애플리케이션
  - 강하게 결합된(Tightly Coupled) 애플리케이션 구조
- 장점
  - 단순성 : 단일 코드베이스이므로 관리가 더 간단함
  - 쉬운 배포 : 배포 절차가 단순함
- 단점
  - 확장성 : 코드베이스가 커지면 확장이 힘듬
  - 유지관리 : 배포할 때마다 오랜 빌드, 배포 시간이 소요됨
  - 영향성 : 일부분의 오류가 전체 서비스에 영향을 줄 수 있음



참조 : https://www.testingfly.com/articles/repository-architecture-monolith-vs-multirepo-vs-Monorepo

3. Repo 전략

❖모놀리식 애플리케이션의 한계

- 모듈화없이 하나의 프로젝트로 구성되면...
  - 관심사의 분리가 어려워지고
  - 설계, 리팩토링 등의 작업을 전체 애플리케이션 구조단위로 수행해야 함

❖멀티 레포(Multirepo)

- 모듈화 수행 -〉 각각의 모듈을 별도의 프로젝트화 -〉 별도의 리포지토리로 관리
- 분리된 프로젝트는 독립적인 개발, 테스트, 빌드, 배포가 가능함.
- 기능변경을 위해 애플리케이션의 전체를 수정할 필요없이 모듈이라는 부품을 교체함
- 장점
  - 독립성 : 각 프로젝트마다의 독립적인 리포지토리, 독립적인 빌드, 배포 작업 수행
  - 명확한 경계 : 각 모듈을 위한 프로젝트 간의 뚜렷한 경계를 가짐
- 단점
  - 종속성 관리 : 리포지토리 간의 종속성 관리가 복잡해짐
  - 패키지의 중복 가능성 : 여러 프로젝트에서 동일한 코드가 작성될 수 있음
  - 관리포인트가 늘어남 : 프로젝트 수만큼...

3. Repo 전략

❖모노레포

- 여러 프로젝트를 동일한 리포지토리로 관리하는 전략
- 장점, 특징
  - 새로운 프로젝트 생성을 위한 오버헤드가 없음
  - 단일화된 관리포인트, 단일화된 원자적 커밋
  - 코드 공유, 재사용이 용이함
  - 빌드 아티팩트 저장소(예:npm registry)에 배포할 필요가 없으므로 의존성 관리가 쉬움
  - 팀간 협업이 용이해짐 -개발자 이동성
- 단점
  - 대규모 프로젝트일 경우 모노레포에서 필요한 리소스를 탐색하는 데 오랜 시간이 소요됨
  - 개발자가 모노레포 전략을 사용하는데 생각의 전환이 필요함
- 주의 사항
  - 엄격한 권한 설정, 폴더기반의 소유권 설정이 요구됨
- 모노레포가 적절한 경우
  - 유사한 제품의 집합
  - 공통기능을 재사용하는 관련 프로젝트들의 집합

3. Repo 전략

❖다양한 모노레포 도구

- npm workspace
- yarn workspace
- lerna + yarn
- lerna + npm
- NX
- Turborepo

❖고려사항

- 이중에서 lerna 오픈소스 프로젝트는 NX에 인수되었음
- lerna 는 내부적으로 NX를 사용함

4. 모노레포를 위한 NX

❖NX란?

- 개발자 생산성 향상, 지속적 통합 성능 최적화, 코드 품질 유지를 위한 도구와 기법을 제공하는 강력한 오픈소스 빌드 시스템
- 빠르고 효과적인 높은 확장성을 가진 모노레포 도구

❖주요 기능

- 주요 프론트엔드 프레임워크 지원 : angular, react, vue
- SSR 프레임워크, 백엔드 프레임워크 지원 : express, nest, next
- Monorepo 지원
- E2E 테스트, 단위 테스트를 위한 도구 지원 : Jest, Cypress 등

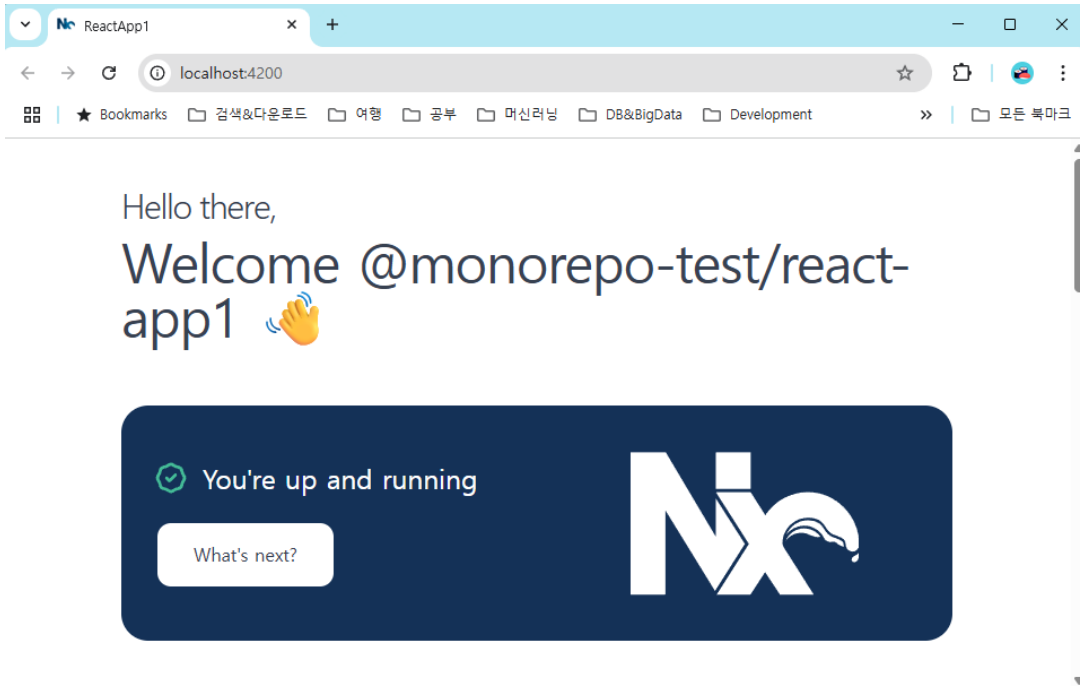
4. 모노레포를 위한 NX

❖간단한 예제 작성

- 리액트 앱을 위한 새로운 NX 워크스페이스 생성

```
npx create-nx-workspace@21 monorepo-test --preset=react-monorepo
```

```
PS>  npx create-nx-workspace@21 monorepo-test --preset=react-monorepo
Need to install the following packages:
create-nx-workspace@21.3.11
Ok to proceed? (y) y


 NX   Let's create a new workspace [https://nx.dev/getting-started/intro]

√ Application name · react-app1
√ Which bundler would you like to use? · vite
√ Which unit test runner would you like to use? · none
√ Test runner to use for end to end (E2E) tests · none
√ Default stylesheet format · css
√ Would you like to use ESLint? · Yes
√ Would you like to use Prettier for code formatting? · Yes
√ Which CI provider would you like to use? · skip
√ Would you like remote caching to make your build faster? · yes

 NX   Creating your v21.3.11 workspace.
```



- 워크스페이스 내의 생성된 프로젝트 실행
  - npx nx serve [프로젝트명
  - 예시)npx nx serve react-app1

4. 모노레포를 위한 NX

❖Generator

- NX 플러그인을 이용해 쉽게 코드, Configuration, 프로젝트, 라이브러리를 스캐폴딩할 수 있도록 하는 도구
- 사용가능한 Generator 목록 : npx nx list @nx/react

```
PS>  npx nx list @nx/react

 NX  Capabilities in @nx/react:

GENERATORS

init : Initialize the `@nx/react` plugin.
application : Create a React application.
library : Create a React library.
component : Create a React component.
redux : Create a Redux slice for a project.
storybook-configuration : Set up storybook for a React app or library.
component-story : Generate storybook story for a React component
stories : Create stories/specs for all components declared in an app or library.
hook : Create a hook.
host : Generate a host react application
remote : Generate a remote react application
cypress-component-configuration : Setup Cypress component testing for a React project
component-test : Generate a Cypress component test for a React component
setup-tailwind : Set up Tailwind configuration for a project.
setup-ssr : Set up SSR configuration for a project.
federate-module : Federate a module.

EXECUTORS/BUILDERS

module-federation-dev-server : Serve a host or remote application.
module-federation-ssr-dev-server : Serve a host application along with it's known remotes.
module-federation-static-server : Serve a host and its remotes statically.
```

4. 모노레포를 위한 NX

❖새로운 애플리케이션 추가

```
npx nx g @nx/react:application react-app2 --directory=apps/react-app2
```

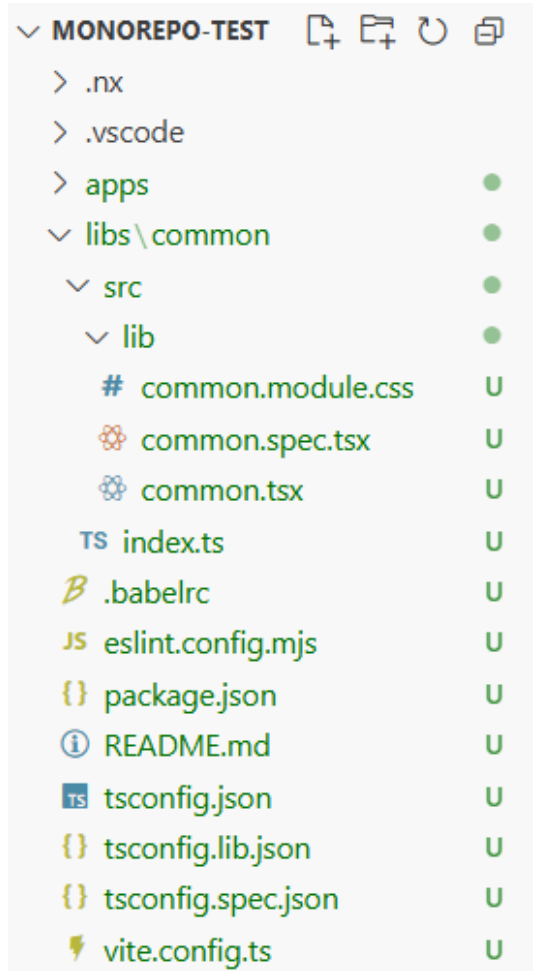
```
PS>  npx nx g @nx/react:application react-app2 --directory=apps/react-app2

 NX  Generating @nx/react:application

√ Would you like to add routing to this application? (y/N) · true
√ What unit test runner should be used? · none
√ Which E2E test runner would you like to use? · none
√ Which port would you like to use for the dev server? · 4201
CREATE apps/react-app2/index.html
CREATE apps/react-app2/public/favicon.ico
CREATE apps/react-app2/src/assets/.gitkeep
CREATE apps/react-app2/src/main.tsx
CREATE apps/react-app2/tsconfig.app.json
CREATE apps/react-app2/src/app/nx-welcome.tsx
CREATE apps/react-app2/src/app/app.module.css
CREATE apps/react-app2/src/app/app.tsx
CREATE apps/react-app2/src/styles.css
CREATE apps/react-app2/tsconfig.json
CREATE apps/react-app2/package.json
CREATE apps/react-app2/eslint.config.mjs
CREATE apps/react-app2/vite.config.ts
UPDATE tsconfig.json

 NX   👀 View Details of @monorepo-test/react-app2

Run "nx show project @monorepo-test/react-app2" to view details about this project.
```

4. 모노레포를 위한 NX

❖로컬 라이브러리 프로젝트 추가

```
npx nx g @nx/react:library common --directory=libs/common --unitTestRunner=vitest --bundler=none
```

```
NX  Generating @nx/react:library

√ What should be the project name and where should it be generated? · common @ libs/common
\CREATE libs/common/project.json
CREATE libs/common/.eslintrc.json
CREATE libs/common/README.md
CREATE libs/common/src/index.ts
CREATE libs/common/tsconfig.lib.json
CREATE libs/common/.babelrc
CREATE libs/common/tsconfig.json
UPDATE nx.json
CREATE libs/common/vite.config.ts
CREATE libs/common/tsconfig.spec.json
UPDATE package.json
CREATE vitest.workspace.ts
CREATE libs/common/src/lib/common.module.css
CREATE libs/common/src/lib/common.spec.tsx
CREATE libs/common/src/lib/common.tsx
UPDATE tsconfig.base.json

added 3 packages, and audited 997 packages in 5s
```



- 생성된 컴포넌트의 기본 이름 : Workspace + Project
  - MonorepoTestCommon 컴포넌트 이름을 Common으로 변경
  - common.tsx, common.spec.tsx 를 모두 변경

4. 모노레포를 위한 NX

❖생성된 라이브러리 프로젝트에 기본으로 생성된 컴포넌트 코드 확인

- common.tsx : 자동생성으로 만들어진 컴포넌트
- index.tsx : 내부에 생성된 컴포넌트를 공개하는 공개 API
- tsconfig.base.json : paths 속성에 라이브러리의 컴포넌트를 참조하기 위한 참조 경로가 나타남

❖만일 기존 라이브러리 프로젝트에 새로운 컴포넌트를 추가한다면?

```
//common 프로젝트에 Home 컴포넌트 추가
npx nx g @nx/react:component Home --project=common --path="libs/common/src/lib/Home"
```

```
PS>  npx nx g @nx/react:component Home --project=common --path="libs/common/src/lib/Home"

 NX  Generating @nx/react:component

√ Should this component be exported in the project? (y/N) · true
CREATE libs/common/src/lib/Home.module.css
CREATE libs/common/src/lib/Home.spec.tsx
CREATE libs/common/src/lib/Home.tsx
UPDATE libs/common/src/index.ts
```

4. 모노레포를 위한 NX

❖Home 컴포넌트 추가후 index.ts 변화

- index.ts

```
export * from './lib/Home';
export * from './lib/common';
```

- react-app1, react-app2 모두에서 Home, Common 컴포넌트 이용할 수 있음

❖Common 컴포넌트 코드 변경

- libs/common/src/lib/common.tsx
- 컴포넌트 이름을 다음과 같이 변경
  - MonorepoTestCommon --〉 Common

4. 모노레포를 위한 NX

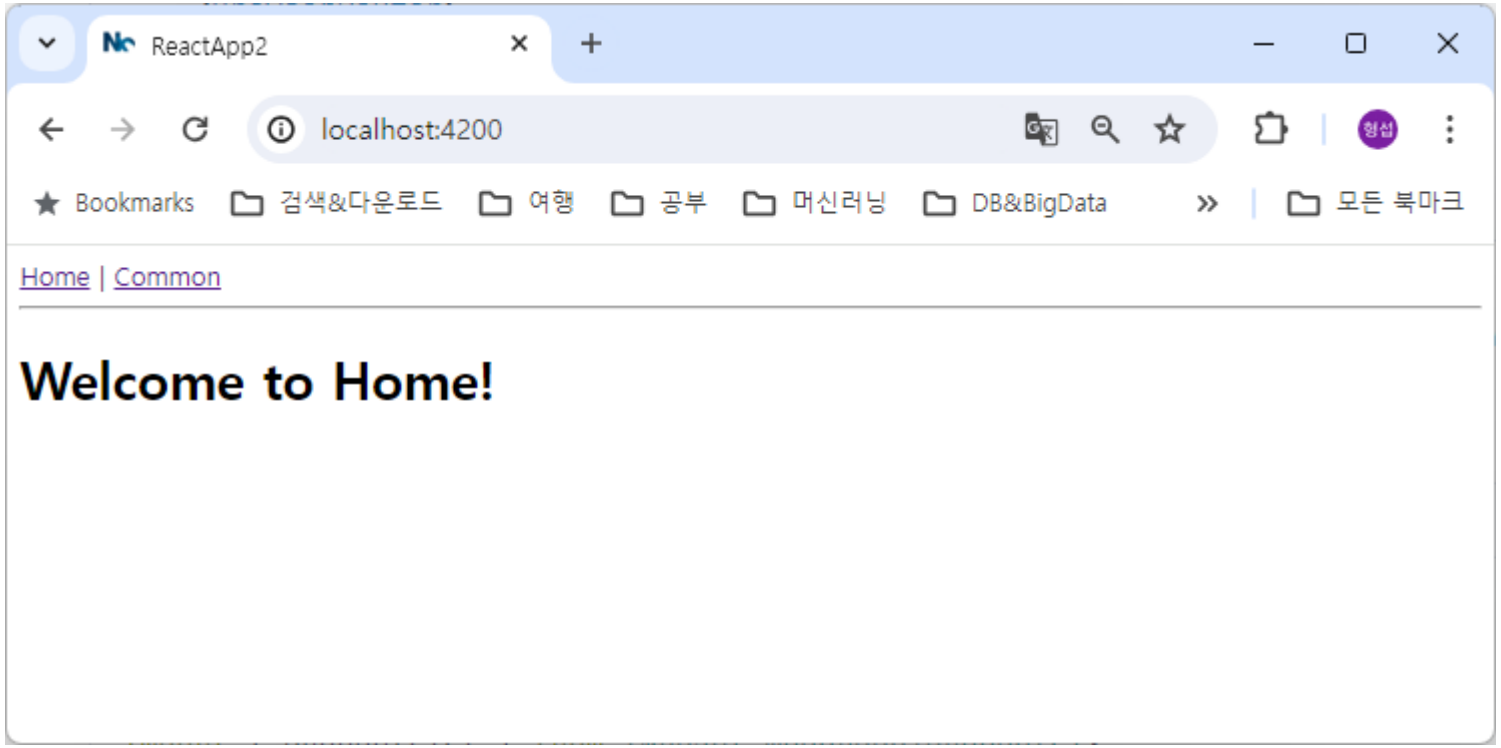
❖react-app2에서 Home, Common 컴포넌트 사용

```
import { Link, Route, Routes } from 'react-router-dom';
//자동완성이 안되므로 다음과 같이 작성
//import {} from '@monorepo-test/common' 을 먼저 작성하고 {} 안쪽을 자동완성할 것
import { Common, Home } from '@monorepo-test/common';

export function App() {
  return (
    <>
      <div>
        <Link to="/">Home</Link> | <Link to="/common">Common</Link>
      </div>
      <hr />
      <Routes>
        <Route path="/" element={<Home />}></Route>
        <Route path="/common" element={<Common />}></Route>
      </Routes>
    </>
  );
}

export default App;
```

4. 모노레포를 위한 NX

❖react-app2 실행

- npx nx serve react-app2



4. 모노레포를 위한 NX

❖react-app1에서도 Home 컴포넌트 사용

- apps/react-app1/src/app/app.tsx

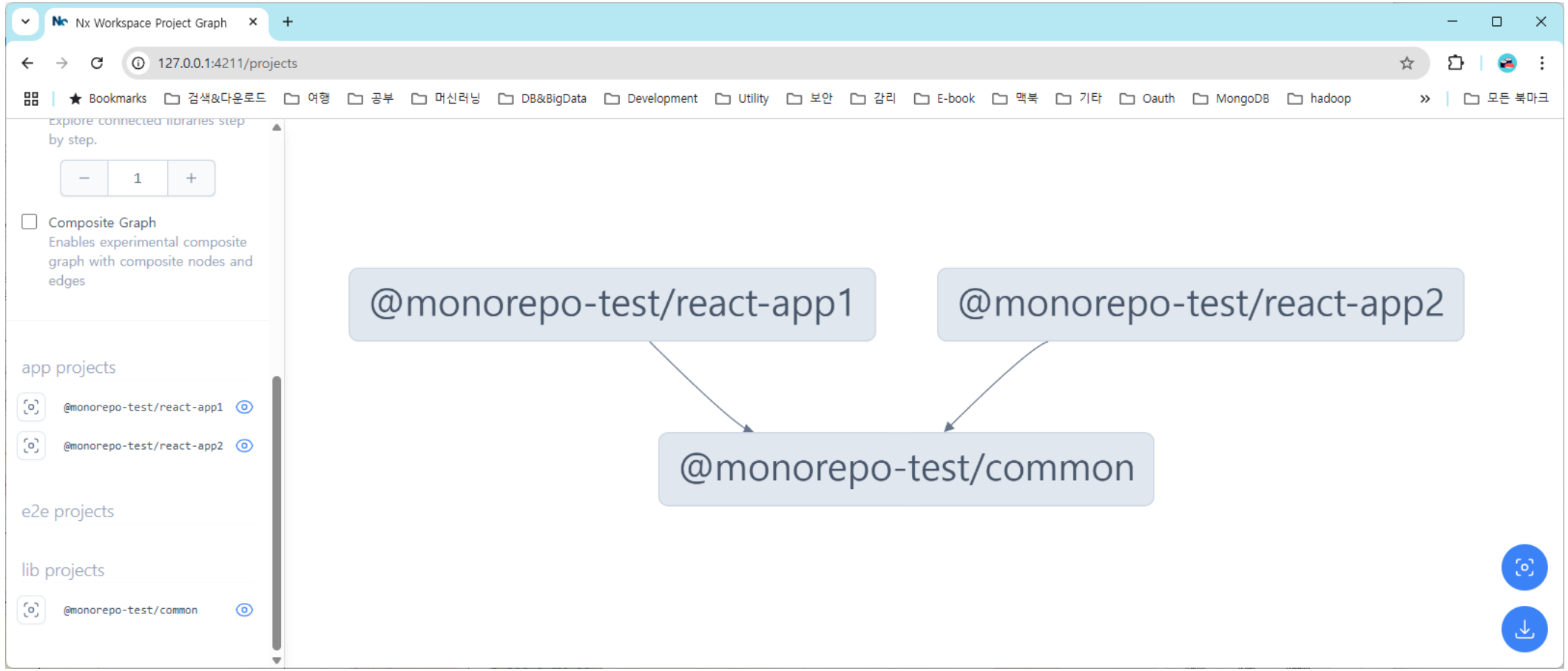
```
// eslint-disable-next-line @typescript-eslint/no-unused-vars
import { Home } from '@monorepo-test/common';

export function App() {
  return (
    <div>
      <Home />
    </div>
  );
}

export default App;
```

- 실행 결과 확인
  - npx nx serve react-app1

4. 모노레포를 위한 NX

❖프로젝트 참조 그래프 보기

- npx nx graph



4. 모노레포를 위한 NX

❖단위 테스트

- npx nx test [프로젝트명]
  - 예) npx nx test common

```
PS>  npx nx test common
(node:35968) [DEP0169] DeprecationWarning: `url.parse()` behavior is not standardized and prone to errors that have secu
rity implications. Use the WHATWG URL API instead. CVEs are not issued for `url.parse()` vulnerabilities.
(Use `node --trace-deprecation ...` to show where the warning was created)

> nx run @monorepo-test/common:test

> vitest


 RUN  v3.2.4 D:/workspace_temp/nx/monorepo-test/libs/common

 ✓ src/lib/common.spec.tsx (1 test) 18ms
 ✓ src/lib/Home.spec.tsx (1 test) 20ms

 Test Files  2 passed (2)
      Tests  2 passed (2)
   Start at  14:16:03
   Duration  17.14s (transform 255ms, setup 0ms, collect 8.24s, tests 38ms, environment 22.90s, prepare 1.42s)


———————————————————————————————————————————————————————————————————————————————————————————————

 NX   Successfully ran target test for project @monorepo-test/common (20s)

View logs and investigate cache misses at https://nx.app/runs/EoLIiWzC4S
```

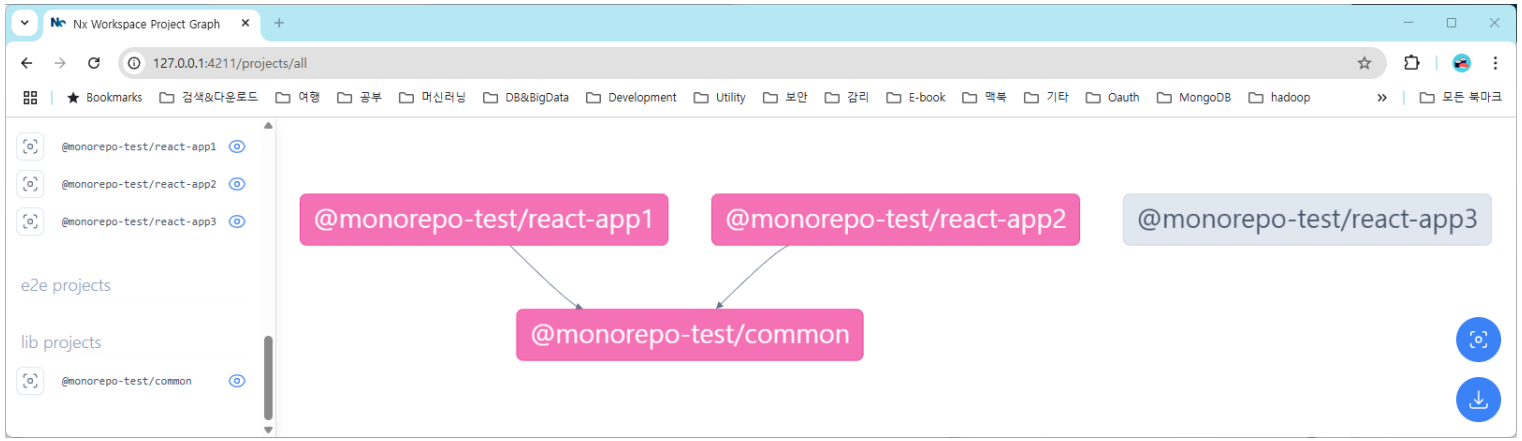
4. 모노레포를 위한 NX

❖린팅

- npx nx lint [프로젝트명]

❖변경 후 영향을 받는 프로젝트 확인

```
// 3번째 애플리케이션 추가
npx nx g @nx/react:application react-app3 --directory=apps/react-app3
// git을 사용해야만 영향을 받는 프로젝트 관계를 확인할 수 있음
git init
git add .
git commit -m 'git commit 후 변경 테스트'
// common 프로젝트의 Home 컴포넌트의 일부를 변경하고 저장만 후 다음 명령어 실행
npx nx graph --affected
```



4. 모노레포를 위한 NX

❖애플리케이션 빌드

- 명령어
  - npx nx run-many -t build
- 빌드를 여러번 수행하면 캐시가 사용됨

```
PS>  npx nx run-many -t build

(node:18036) [DEP0169] DeprecationWarning: `url.parse()` behavior is not standardized and prone to errors that have secu
rity implications. Use the WHATWG URL API instead. CVEs are not issued for `url.parse()` vulnerabilities.

   √  nx run @monorepo-test/react-app1:build (2s)
   √  nx run @monorepo-test/react-app3:build (2s)
   √  nx run @monorepo-test/react-app2:build (2s)


————————————————————————————————————————————————————————————————————————————————————————————————————————————————

 NX   Successfully ran target build for 3 projects (3s)

View logs and investigate cache misses at https://nx.app/runs/dDykepna5e
```

4. 모노레포를 위한 NX

❖이밖에도 다음과 같은 기능이 있음

- 모듈 경계 규칙으로 제약 조건 부과
  - 모듈화된 라이브러리가 결합되는 방법을 제어하는 방법을 제공함
    - 예) A모듈은 B모듈을 import할 수 있지만 반대는 허용하지 않도록 함
    - 각 프로젝트의 project.json 파일에 태그를 부여하고 .eslintrc.json 파일에 규칙을 정의함
    - https://nx.dev/getting-started/tutorials/react-monorepo-tutorial#imposing-constraints-with-module-boundary-rules
  - 예제 확인
    - https://github.com/nrwl/nx-recipes/tree/main/react-monorepo 폴더
    - https://github.com/nrwl/nx-recipes/blob/main/react-monorepo/libs/orders/package.json 파일의 tag 확인
    - https://github.com/nrwl/nx-recipes/blob/main/react-monorepo/libs/shared/ui/package.json 파일의 tag 확인
    - https://github.com/nrwl/nx-recipes/blob/main/react-monorepo/eslintrc.config.mjs파일
      - » rules의 depConstraints 내용 확인
  - 위 예제의 설정 내용
    - type:feature 태그가 지정된 리소스는 'type:feature', 'type:ui' 태그를 가진 리소스만 참조할 수 있음
    - type:ui 태그가 지정된 리소스는 'type:ui' 태그를 가진 리소스만 참조할 수 있음
    - scope:orders 태그가 지정된 리소스는 'scope:orders', 'scope:products', 'scope:shared' 태그를 가진 리소스만 참조할 수 있음

5. Module Federation

❖Module Federation이란?

- 런타임에 코드를 더 작은 배포 모듈로 분할하고 애플리케이션 간에 공유할 수 있도록 하는 기술
- 마이크로 프론트엔드 구조로 개발할 때 적용함
- Mono Repo 환경에서는 각각의 리모트 애플리케이션을 피더레이션 모듈로 빌드하고 호스트 애플리케이션에서 소비하도록 구성함

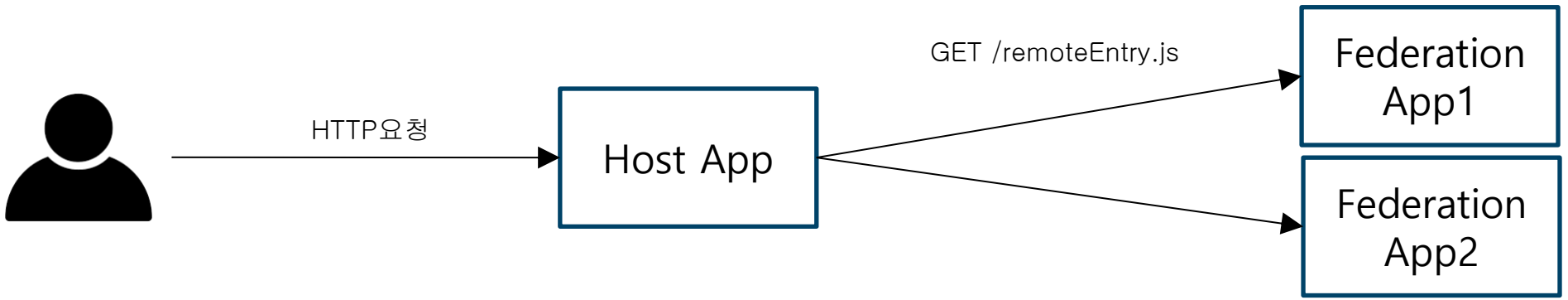
❖ 핵심 개념

- Host Application
  - 피더레이션 모듈을 런타임에 소비하는 애플리케이션
  - 전체 애플리케이션의 진입점과 라우팅을 지원하고 관리함
- Remote Application
  - 네트워크를 통해 런타임에 피더레이션 모듈을 제공하는 애플리케이션
  - 독립적으로 실행할 수도 있어서 독립적으로 개발, 디버깅 가능

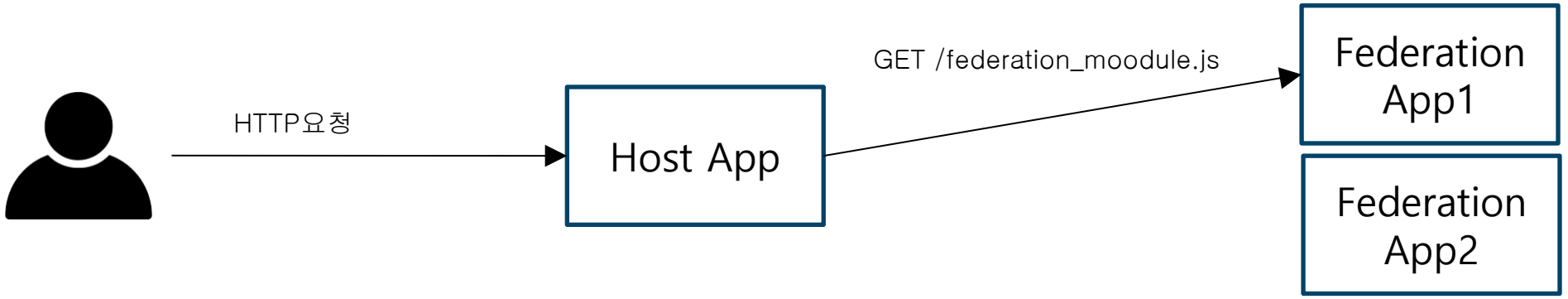
5. Module Federation

❖ NX를 통한 모듈 피더레이션 개요도

- 초기화 시에 remoteEntry 모듈만 로딩



- App1 의 라우팅 경로로 진입하면 실제 기능을 수행하는 피더레이션 모듈을 Lazy Loading



5. Module Federation

❖NX를 이용한 모듈 피더레이션 기능 실습

- hands-on-lab
  - 다음 문서를 참조하여 진행
  - https://github.com/stepanowon/nx-module-federation-demo
- 사전 준비사항
  - Node.js 24(22 버전 이상)
  - 인터넷 연결
  - nx 21버전

5. Module Federation

❖hands-on-lab 수행 후 검토

- project.json 파일

// host-app 의 project.json

```
{
  "name": "host-app",
  "$schema": "../../node_modules/nx/schemas/project-schema.json",
  "sourceRoot": "apps/host-app/src",
  "projectType": "application",
  "tags": [],
  "targets": {
    "serve": {
      "options": {
        "port": 4200
      }
    }
  }
}
```

// app1, app2 의 project.json

```
{
  "name": "app1",
  "$schema": "../../node_modules/nx/schemas/project-schema.json",
  "sourceRoot": "apps/app1/src",
  "projectType": "application",
  "tags": [],
  "targets": {
    "serve": {
      "options": {
        "port": 4201
      },
      "dependsOn": ["host-app:serve"]
    }
  }
}
```

5. Module Federation

❖hands-on-lab 수행 후 검토(이어서)

- Host Application 설정 검토

```
// apps/host-app/module-federation.config.ts 파일
  const config: ModuleFederationConfig = {
    name: 'host-app',
    remotes: ['app1', 'app2'],  // 원격 마이크로프론트엔드 앱들을 등록
  };

// host-app의 rspack.config.ts 파일에서 module-federation.config.ts 파일을 참조
```

- Remote Application 설정 검토

```
//apps/app1/module-federation.config.ts 파일
  const config: ModuleFederationConfig = {
    name: 'app1',
    exposes: {
      './**Module**': './src/remote-entry.ts',  // 외부로 노출할 모듈 정의. 외부로 노출시킬 때의 이름은 Module
    },
  };
```

5. Module Federation

❖hands-on-lab 수행 후 검토(이어서)

- Remote Entry 파일 검토

```
// apps/app1/src/remote-entry.ts 파일
// 메인 컴포넌트(App)를 외부로 공개,노출시킴
export { default } from './app/app';
```

- Host에서 Remote 모듈 동적 로딩

```
// apps/host-app/src/app/app.tsx
//Remote Application에서 노출된 모듈의 이름(Module)을 이용해 import 수행
  const App1 = React.lazy(() => import('app1/Module'));
  const App2 = React.lazy(() => import('app2/Module'));
```

5. Module Federation

❖사용된 NX 구성 설정 요소

- NX 플러그인 설정(nx.json 파일 참조)
  - @nx/module-federation 플러그인으로 모듈 페더레이션 지원
  - @nx/rspack/plugin으로 빌드 도구 설정
- 패키지 의존성 (package.json)
  - @module-federation/enhanced: 향상된 모듈 페더레이션 기능
  - @nx/module-federation: Nx 모듈 페더레이션 플러그인
  - @rspack/core: 빠른 번들러
- 아키텍처 패턴
  - Host: 다른 마이크로프론트엔드들을 조합하는 메인 앱
  - Remote: 독립적으로 개발/배포되는 마이크로프론트엔드
  - Dynamic Import: 런타임에 원격 모듈을 동적으로 로딩
  - Lazy Loading 기법 사용
  - Bootstrap Pattern: 각 앱이 독립 실행 가능한 구조

6. 정리

- ❖ 리액트 컴포넌트 설계 원칙
  - 확장성, 재사용성
  - 단일 책임 원칙
  - 렌더링 최적화
- ❖ 폴더 구조 정의
  - 파일 유형별로 분리
  - 기능, 라우트 단위 정의 : FSD
  - 프로젝트 규모를 고려해 미리 선정
- ❖ FSD : 기능 분할 설계
  - 레이어
  - 슬라이스
  - 세그먼트
- ❖ 레포전략
  - 멀티레포
  - 모노레포 : NX, lerna