

react router

react는 url에 따른 페이지를 제어하기 위해 router 라이브러리를 사용합니다.
우리가 사용할 router는 react-router 입니다.

설치를 위해 다음 명령어를 사용합니다.

```
npm i react-router-dom
```

react-router에서 제공하는 라이브러리는 react-router-dom, react-router-native 가 있습니다.
react-router-dom 은 웹에서 사용되는 라이브러리이고
react-router-native는 앱에서 사용되는 라이브러리 입니다.

react router

이제 route를 구성하기 위해 index.js 파일의 코드를 변경시켜보겠습니다.

```
ReactDOM.render(  
  <React.StrictMode>  
    <App />  
  </React.StrictMode>,  
  document.getElementById('root')  
);
```



```
import { BrowserRouter, Routes, Route } from 'react-router-dom'  
  
ReactDOM.render(  
  <React.StrictMode>  
    <BrowserRouter>  
      <Routes>  
        <Route path="/" element={<App />}>  
          <Route path="dashboard" element={<Dashboard></Dashboard>}></Route>  
        </Route>  
      </Routes>  
    </BrowserRouter>  
  </React.StrictMode>,  
  document.getElementById('root')  
);
```

react router - BrowserRouter, Routes, Route

- BrowserRouter
 - BrowserRouter 는 React의 화면을 url과 매칭시키기 위해 html5 history api 를 사용합니다. 즉 history api 와 url을 연결하기 위해 사용합니다.
- Routes
 - Routes는 Route를 감싸고 있습니다. route 구성 설정영역을 의미합니다.
- Route
 - url과 해당 url에 매칭해야하는 element를 설정하는 기능을 합니다.
 - nesting 구조로 작성이 가능합니다.

'/' 경로에 App을 매칭시켜서 App을 최상위 컴포넌트로 등록시켰습니다.

그리고 그 하위엔 dashboard 경로와 <Dashboard> 컴포넌트를 매칭시켰습니다.

이 경우엔 App 컴포넌트 안에서 Dashboard 컴포넌트를 사용하는 경우로 App.js 안에 Dashboard 컴포넌트를 볼 수 있는 영역을 지정해줘야 합니다.

react router - Outlet

```
import logo from './logo.svg';
import './App.css';
import { Outlet } from 'react-router-dom';

function App() {
  return <>
    <Outlet></Outlet>
  </>
}

export default App;
```

App.js의 코드입니다.

route 구조중 app-route의 하위 route들을 보여주기 위해선 Outlet을 사용해야 합니다.
그럼 route에 등록한 컴포넌트들이 Outlet 영역에 나타나게 됩니다.

react router - Link

```
import logo from './logo.svg';
import './App.css';
import { Outlet, Link } from 'react-router-dom';

function App() {
  return <>
    <nav>
      <ul className="link-list">
        <li className="link-item"><Link to="dashboard">dashboard</Link></li>
      </ul>
    </nav>
    <Outlet></Outlet>
  </>
}

export default App;
```

Routes에 설정한 route로 페이지 이동을 하기 위해선 Link가 사용됩니다.

Link to에 상대경로를 입력함으로써 현재 경로에 대한 상대경로를 합쳐서 해당 url로 이동하게 됩니다.

지금은 Link에 dashboard url이 설정되어 있으므로 '/dashboard'로 이동하게 되지만 '/dashboard'는 '/'의 하위경로로 등록되어 있으므로 App 컴포넌트의 Outlet에 Dashboard 컴포넌트를 보여주게 됩니다.

react router - useNavigate

```
import { useNavigate } from 'react-router-dom';

.......

const nav = useNavigate()

const moveToDetailPage = (id) => {
  nav(`/ex-router/${id}`)
}
```

페이지 이동은 Link를 이용해서 이동할 수 있지만 특정 UI를 이용해서 페이지를 이동하거나, 어떤 로직의 결과로 페이지가 강제로 이동되어야 하는 경우가 있습니다.

이럴 때 useNavigate를 이용해서 페이지를 이동시킬 수 있습니다.

위의 코드처럼 useNavigate를 react-router-dom에서 가져온 후 nav함수를 생성해주고 필요한 곳에서 nav함수를 이용해서 이동하려는 경로를 매개변수로 전달해주면 해당 페이지로 이동할 수 있습니다.

react router - useParams, useLocation

만약 url params, queryString을 사용하는 경우라면 해당값에 접근하기 위한 함수로 useParams, useLocation을 사용하면 됩니다.

```
// router 설정
<Route path="ex-router/:id" element={<RouterDetail></RouterDetail>}></Route>

// RouterDetail에서 사용
import { useLocation, useParams } from 'react-router'

const location = useLocation()
const { id } = useParams()
```

router를 설정하는 부분에서 :id를 통해 url parameter를 받게 했습니다. 이렇게 설정된 상태에서 해당 값을 받아오기 위해선 react-router에서 제공하는 useParams()를 사용하면 됩니다.

또한 url 정보에 접근을 원한다면 useLocation을 사용하면 url의 정보를 가져올 수 있으며 queryString을 통해 전달된 데이터를 가져오려면 useLocation의 search 데이터를 사용하면 됩니다.

Ex - 페이지 추가

MyRouterPage 컴포넌트를 만들고 url에 등록해주세요

컴포넌트명: MyRouterPage

url: /my-router-page

화면에 h1 태그로 'This is My Router Page !!!' 가 보이게 만들어주세요

lazy loading

react는 SPA의 특성상 사이트의 첫번째 진입시 페이지들을 전부 다운받기 때문에 초기 로딩이 느립니다.
이러한 문제는 사이트 규모가 클 수록 심해지기 때문에 이런 문제를 해결하기 위한 방법이 제시됩니다.

바로 lazy-loading 입니다.

lazy-loading은 사이트 접속시 모든 페이지를 받지 않고 사이트의 특정 페이지 접속시 관련된 페이지들만 받게 함으로써
사이트를 최적화 시키는 방법입니다.

lazy-loading은 react 자체에서 제공하는 기능으로 react 라이브러리 안에서 lazy 함수와 Suspense 컴포넌트로 구현할 수 있습
니다.

lazy loading - lazy, Suspense

```
import React, { lazy, Suspense } from 'react';

const LazyRouterIndex = lazy(() => import('./pages/router/lazy-router-index'))

<Route path="lazy-ex-router" element={  
  <Suspense fallback={<div>...page Loading</div>}>  
    <LazyRouterIndex></LazyRouterIndex>  
  </Suspense>  
></Route>
```

lazy함수와 Suspense 컴포넌트를 react에서 가져옵니다.

lazy함수는 함수를 매개변수로 입력받는 함수인데 lazy-loading 할 컴포넌트를 반환하는 함수를 입력받습니다.

그렇게 LazyRouterIndex 컴포넌트를 생성하고 해당 컴포넌트를 Suspense를 이용해 Route의 element에 등록하면 됩니다.

Suspense의 fallback은 페이지 로딩시 띄워줄 컴포넌트를 입력하는 속성입니다.

이렇게 등록된 페이지는 다른 페이지들과는 다르게 해당 페이지 이동시 페이지를 다운받는걸 확인할 수 있습니다.

lazy loading

```
const LazyRouterIndex = lazy(() => import('./pages/router/lazy-router-index'))
const LazyRouterDetail = lazy(() => import('./pages/router/lazy-router-detail'))

<Route path="lazy-ex-router/*" element={
  <Suspense fallback={<div>...page Loading</div>}>
    <Routes>
      <Route path="/" element={<LazyRouterIndex></LazyRouterIndex>}></Route>
      <Route path=":id" element={<LazyRouterDetail></LazyRouterDetail>}></Route>
    </Routes>
  </Suspense>
}></Route>
```

위의 예제는 여러 컴포넌트에 lazy-loading을 적용시킨 예제입니다.

앞의 예제와는 다르게 그룹을 묶고있는 Route path에 'lazy-ex-router/*' 패턴 문자가 들어가 있습니다.

해당 url로 하위에 여러 lazy-loading을 children으로 넣고싶으면 '*' 로 url 사용범위를 적용시켜야 합니다.

style

react에선 style을 적용하기 위한 다양한 방법들이 존재합니다.

우선 다음의 3가지 방법에 대해 알아보겠습니다.

- style import
- inline
- module

style - style import

css 파일에 스타일을 만들고 해당 파일을 import 하는 방식입니다.
기존의 css 사용하는 방식과 비슷합니다.

```
// ex1-style-sheet.css
.primary {
  color: orange;
}

// ex1-style-sheet.jsx
import './ex1-style-sheet.css'

function Ex1StyleSheet() {
  return (
    <div>
      <h1 className="primary">Stylesheets</h1>
    </div>
  )
}
```

style - inline style

jsx에 style json 데이터를 넘겨주는 방식입니다. jsx 문법에 맞춰서 속성값을 사용해줘야 하기 때문에 camelCase로 만들어줘야 합니다.

```
// ex2-inline.jsx

const heading = {
  fontSize: '72px',
  color: 'blue'
}

function Ex2Inline() {
  return (
    <div>
      <h1 style={heading}>This is Inline</h1>
    </div>
  )
}
```

style - module

css를 모듈화 시켜서 사용하는 방법입니다. 모듈화 시킬 css 파일은 ‘파일명.module.css’로 이름을 짓고 해당 파일을 import 해서 각 스타일을 object 형식처럼 사용할 수 있습니다.

```
// ex3-module-style.module.css

.module_h1 {
  color: red;
  font-size: 80px
}

// ex3-module-style.jsx
import style from './ex3-module-style.module.css'

function Ex3ModuleStyle() {
  return (
    <div>
      <h1 className={style.module_h1}>This is Module style</h1>
    </div>
  )
}
```

Ex - style 추가

MyRouterPage 에 스타일을 추가해주세요

스타일을 추가해서 기존의 h1 태그 스타일을 다음과 같이 설정해주세요

color: red

font-size: 40px

lifecycle

react component에는 생명주기가 있습니다.

생명주기(lifecycle)은 컴포넌트가 실행되고, 업데이트되고, 제거될 때 각 단계에 맞춰서 발생하는 특정한 이벤트들을 이야기 합니다.

우리는 이러한 생명주기 함수들을 사용해서 컴포넌트가 실행될 때 서버에서 데이터를 받아와서 셋팅하거나, 컴포넌트가 사라질 때 이벤트 리스너에 등록된 이벤트들을 정리할 수 있습니다.

먼저 lifecycle의 종류를 알아보겠습니다.

lifecycle

- mount
 - constructor
 - getDerivedStateFromProps
 - render
 - componentDidMount
- update
 - getDerivedStateFromProps
 - shouldComponentUpdate
 - render
 - componentDidUpdate
- unmount
 - componentWillUnmount

lifecycle-mount

Constructor

컴포넌트가 생성될 때 실행됩니다. event-bind를 구현하거나 state를 초기화 하는 용도로 주로 사용됩니다.

getDerivedStateFromProps

prop가 변경될 때 발생합니다. props를 통해 state의 값을 구성하는 경우 여기서 로직을 구현해주면 됩니다.
실무에선 사용해본적이 없습니다.

render

컴포넌트에 전달된 props와 state를 읽어서 jsx를 반환하는 함수입니다.
만약 여기에 다른 컴포넌트들이 있다면 해당 컴포넌트들의 lifecycle이 시작됩니다.

componentDidMount

모든 요소들이 DOM에 render되면 발생하는 이벤트입니다.
DOM요소가 준비된 상태에서 데이터를 입력하기 때문에 대부분의 통신은 이 부분에서 이루어 집니다.

lifecycle-update

getDerivedStateFromProps

컴포넌트가 update 될 때 다시 발생합니다.

shouldComponentUpdate

컴포넌트가 update 되었을 때 re-render를 할지 결정하는 lifecycle입니다.

render

랜더링이 일어납니다.

componentDidUpdate

re-render가 완료된 후 발생하는 lifecycle입니다.

lifecycle-unmount

componentWillUnmount

컴포넌트가 종료될 때 실행되는 lifecycle 입니다. event-listener를 없애거나, subscription들을 정리하는 로직이 구현되는곳입니다.

useEffect

class component에서는 lifecycle을 통해 해당 상황에 맞는 적절한 로직을 만들 수 있습니다.
그렇다면 functional component의 경우 lifecycle을 어떻게 접근해야 할까요?

이 때 사용가능한 것이 useEffect 입니다.

useEffect는 componentDidMount, componentDidUpdate, componentWillUnmount 를 합친것과 비슷한 역할을 합니다.

useEffect - 구조

useEffect 함수의 구조는 다음과 같습니다.

```
import React, { useEffect, useState } from 'react'  
  
useEffect(callback, [dependency variable])
```

- callback 함수
 - callback 함수는 class component의 componentDidMount + componentDidUpdate와 비슷한 역할을 합니다.
초기값을 셋팅할 수 있고, 컴포넌트가 재랜더링 되면 callback 함수의 로직이 다시 실행되는 형식입니다.
- callback 함수 return
 - callback 함수는 return을 할 수 있습니다. 이 때 return 형태는 함수의 형태인데 이를 cleanup 함수라 부릅니다.
 - cleanup 함수는 class component의 componentWillUnmount와 비슷한 역할을 합니다.
 - 컴포넌트가 종료되면서 정리되어야 할 로직들을 구현합니다.
- dependency variable
 - useEffect의 두번째 매개변수로는 variable을 array형태로 받습니다.
 - 매개변수로 받은 값들은 의존값이라고 하는데 이런 의존값에 변화가 일어날 때마다 useEffect가 수행되지만, 배열에 없는 값들이 변화가 일어날 경우 useEffect가 수행되지 않습니다.

Ex - todo 만들어보기

todo-list, todo-detail 페이지를 만들어보세요.

url: /todos, /todos/:id

todo-list 페이지에선 todo의 title이 목록으로 나오고 해당 title을 누르면 detail 페이지로 이동합니다.

detail 페이지에선 title이 나오고 todo의 completed가 true 면 'done', false 면 'not yet' 이 나오도록 합니다.