React-9,13장

생성일	@2022년 5월 10일 오후 8:17
분류	React-리액트를 다루는 기술 책
주제	
날짜	

컴포넌트 스타일링

- 일반 CSS: 가장 기본적인 방식
- Sass: 자주 사용되는 CSS전처리기 중 하나, 확장된 CSS문법을 사용
- CSS Module: 스타일을 작성 할때 CSS클래스가 다른 CSS클래스의 이름과 절대 충돌하지않도록 파일마다 고유한 이름을 자동으로 생성해주는 옵션
- styled-commponents: 스타일을 JS파일에 내장시키는 방식, 스타일을 작성함과 동시에 해당 스타일이 적용된 컴포넌트를 만들수 있게해줌

일반 CSS

- 소규모 프로젝트를 개발하고 있다면 새로운 스타일링 시스템을 적용하는것이 불필요할수 있음
- 프로젝트에 이미 적용되어있는 기본 CSS시스템을 사용하는 것만으로도 충분함
- 이름짓는 규칙: 자신만의 네이밍 규칙을 만들거나 메이저 업체의 코딩 규칙 문서를 참고
  - ex) 네이버 코딩 컨벤션 등
  - BEM 네이밍: CSS방법론 중 하나, 해당 클래스가 어디에서 어떤 용도로 사용되는지 명확하게 작성하는 방식

Sass

- CSS전처리기
- 복잡한 작업을 쉽게 할수있도로 해주고, 스타일코드의 재활용성을 높여줌
- 코드의 가독성을 높여 유지 보수를 쉽게해줌
- scss문법과 sass문법의 차이점
  - sass는 {}(중괄호)와 ;(세미콜론)을 사용하지 않음
  - scss는 기존 css를 작성하는 방식과 문법이 크게 다르지않음
- $변수이름: 값; - 변수 사용
- @mixin 함수이름() {} - 재사용되는 스타일 블록을 함수처럼 사용가능
  - mixin을 호출한 위치에서 {} 안의 스타일 코드가 적용됨
- 자손 셀렉터- 포함관계를 {}의 중첨으로 표현
- &.클래스이름- <? class='클래스 클래스'/>

Scss실습은 추후 다시 해보기⇒ 제대로 동작하지 않는다

CSS Module

- CSS를 불러와서 사용할때 클래스 이름을 고유한 값, 즉 [파일 이름]_[클래스 이름]__[해시값] 형태로 자동으로 만들어서 **컴포넌트 스타일 클래스 이름이 중첩되는 현상을 방지**해주는 기술
- 설정을 따로 할 필요없이 파일이름.module.css 확장자로 파일을 저장하면 CSS Module이 적용됨
- 클래스 이름을 지을때 그 고유성에 대해 고민하지 않아도 됨
  ⇒ 서로 다른 CSS파일에 같은 클래스 이름이 있어도 난독화 과정에서 서로 다른값으로 구분됨
- 특정 클래스가 웹 페이지에서 전역적으로 사용되는 경우- :global을 앞에 붙여 글로벌 CSS임을 명시
- 고유한 클래스 이름을 사용하려면 클래스를 적용하고 싶은 JSX element에 className={style.[클래스이름]}형태로 전달

- :global을 사용해 전역적으로 선언한 클래스의 경우- 그냥 문자열로 넣어줌

- Sass도 동일한 방법으로 사용

```jsx
import React from 'react'
import styles from '../styled/CssModule.module.css';

const CssModule = () => {
  return (
    <div className={styles.wrapper}>
        안녕하세요!! <span className='something'>CSS Module!</span>
    </div>
  )
}

export default CssModule
```

```css
.wrapper{
    background-color: black;
    padding: 1rem;
    color: white;
    font-size: 2rem;
}
/*글로벌 CSS*/
:global .something{
    font-weight: bolder;
    color: cadetblue;
}
```

출력 결과

안녕하세요!! CSS Module!

클래스 이름을 두개 이상 적용

```jsx
<div className={`${styles.wrapper} ${styles.inverted}`}>
    안녕하세요!! <span className='something'>CSS Module!</span>
</div>
```

```css
.inverted{
    color: black;
    background-color: white;
    border: 1px solid black;
}
```

출력 결과

안녕하세요!! CSS Module!

- ES6문법 템플릿 리터럴을 사용해 문자열을 합해준것
  - 문자열 안에 JS 레퍼런스를 쉽게 넣어줄수 있음
- = className={[styles.wrapper, styles.inverted].join(' ')}
  - 클래스를 배열로 배치해 join()함수 사용 가능

---

classname

- CSS클래스를 조건부로 설정할때 매우 유용한 라이브러리
- yarn add classnames 설치
- 여러가지 종류의 파라미터를 조합해 CSS클래스를 설정할수 있기 때문에 컴포넌트에서 조건부로 클래스를 설정핼때 매우 편함

```
//사용법
import classNames from 'classnames';

classNames('one', 'two');
//=>'one two'
classNames('one', {two: true});
//=>'one two'
classNames('one', {two: false});
//=>'one'
classNames('one', ['two', 'three']);
//=>'one two three'

const myClass= 'hello';
classNames('one', myClass, {myCondition: true});
//=>'one hello myCondition'
```

- const myClass= 'hello';⇒ 클래스 이름의 변수화
- ... {myCondition: true}: true대신 a일때
  ⇒ let a= true;일 경우- 조건부 클래스 적용(a값에 따라 적용 여부가 바뀜)

```
const MyComponent= ({highlighted, theme})=> (
  <div className={classNames('MyComponent', {highlighted}, theme}>
    Hello
  </div>);
```

- hightlighted: props값이 존재할때만 props에 저장된 값이 class이름으로 사용
- hightlighted값이 true hightlighted클래스가 적용되고, false면 적용되지 않음
- theme으로 전달받은 문자열은 내용 그대로 클래스에 적용됨

라이브러리 적용하지 않을 경우

```
const MyComponent= ({highlighted, theme})=> {
  <div className={`MyComponet ${theme} ${highlighted ? 'highlighted' : ''}`>
    Hello
  </div>
};
```

classNames의 내장 함수-bind()

```
import React from 'react'
import styles from '../styled/CssModule.module.css';
import classNames from 'classnames/bind';

//미리 styles에서 클래스를 받아 오도록 설정함
const cx= classNames.bind(styles);
const CssModule = () => {
  return (
    <div className={cx('wrapper', 'inverted')}>
        안녕하세요!! <span className='something'>CSS Module!</span>
    </div>
  )
}

export default CssModule
```

- bind함수를 사용하면 클래스에 넣어줄 때마다 styles.[클래스이름]형태를 사용할 필요가 없음
- cx('클래스이름1','클래스이름2')형태로 사용

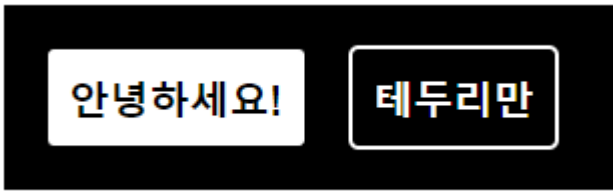
---

styled-components

- JS파일 안에 스타일을 선언하는 방식: CSS-in-JS
- yarn add styled-components로 패키지 추가
- JS파일 하나에 스타일까지 작성할수 있기때문에 스타일 파일을 따로 만들지 않아도 된다는 큰 이점이 있음

```jsx
import React from "react";
import styled, { css } from "styled-components";

const Box = styled.div`
  /*props로 넣어준 값을 직접 전달해줄수 있음 */
  background: ${(props) => props.color || "blue"};
  padding: 1rem;
  display: flex;
`;
const Button = styled.button`
  background: white;
  color: black;
  border-radius: 4px;
  padding: 0.5rem;
  display: flex;
  align-items: center;
  justify-content: center;
  box-sizing: border-box;
  font-size: 1rem;
  font-weight: bold;
  &:hover {
    background: gray;
  }

  /* inverted값이 true일때 특정 스타일을 부여함*/
  ${(props) =>
    props.inverted &&
    css`
      background: none;
      border: 2px solid white;
      color: white;
      &:hover {
        background: white;
        color: black;
      }
    `}
  &+ button {
    margin-left: 1rem;
  }
`;
const StyledComponents = () => (
  <Box color="black">
    <Button>안녕하세요!</Button>
    <Button inverted={true}>테두리만</Button>
  </Box>
);

export default StyledComponents;

```

출력 결과



- Box,Button⇒ 컴포넌트

Tagged 템플릿 리터럴

- 스타일을 작성할때 `(백틱)을 사용하여 만든 문자열에 스타일 정보를 넣음 =>사용한 문법은 Tagged 템플릿 리터럴이라고 함
- 일반 템플릿 리터럴과 다른점: 템플릿안에 JS객체나 함수를 전달할때 온전히 추출할수 있다는것

스타일링된 엘리먼트

- styled-commponents를 사용해 스타일링된 엘리먼트를 만들때 컴포넌트 파일의 상단에 styled를 불러오고 styled.태그명을 사용해 구현

```
import styled form 'styled-commponents';

const MyComponent= styled.div`
  font-size: 2rem;
`;
```

- styled.div뒤에 Tagged 템플릿 리터럴 문법을 통해 스타일을 넣어주면 해당 스타일이 적용된 div로 이루어진 리액트 컴포넌트가 생성됨
- <MyComponent/>형태로 사용할수 있음

```
const MyInput= styled('input')`
  background: gray;
`
```

```
const StyledLink= styled(Link)`
  color: blue;
`
```

- 사용해야할 태그명이 유동적이거나 특정 컴포넌트 자체에 스타일링
- 태그 타입을 styled함수의 인자로 전달하거나 아예 컴포넌트 형식의 값을 넣어줌

스타일에서 props조회

- 스타일쪽에서 컴포넌트에게 전달된 prpos값을 참조

```
background: ${(props) => props.color || "blue"};
```

- porps를 조회해서 porps.colord의 값을 사용
- color값이 주어지지 않았을때: 기본값 blue로 설정
- JSX에서 사용될때 다음과 같이 color값을 props로 넣어줄수있음
  - <Box color=”black”>...</Box>

---

props에 따른 조건부 스타일링

- porps를 사용해 서로 다른 스타일을 적용할수 있음

```
<Button>안녕하세요!</Button>
<Button inverted={true}>테두리만</Button>
```

- 스타일 코드 여러줄을 props에 따라 넣어주어야 할때 CSS를 styled-components에서 불러와야함

- CSS를 사용하지 않고 바로 문자열을 넣어도 가능함

```
${(props) =>
    props.inverted &&
    `
      background: none;
```

```
      border: 2px solid white;
      color: white;
      &:hover {
        background: white;
        color: black;
      }
    `}
```

- 해당 내용이 그저 문자열로만 취급됨
  → VS Code 확장팩에 신텍스 하이라이팅이 제대로 이루어지지않음
  → Tagged 템플릿 리터럴이 아니기 때문에 함수를 받아 사용하지 못해 해당 부분에서는 props값을 사용하지 못함
- porps를 참조한다면 반드시 CSS로 감싸주어 Tagged 템플릿 리터럴을 사용

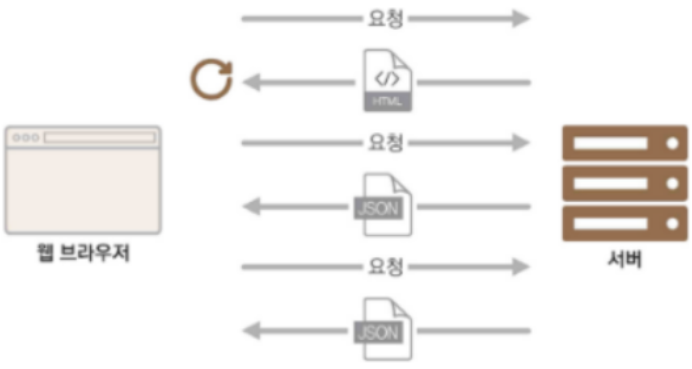
라우터로 SPA 개발하기

라우팅

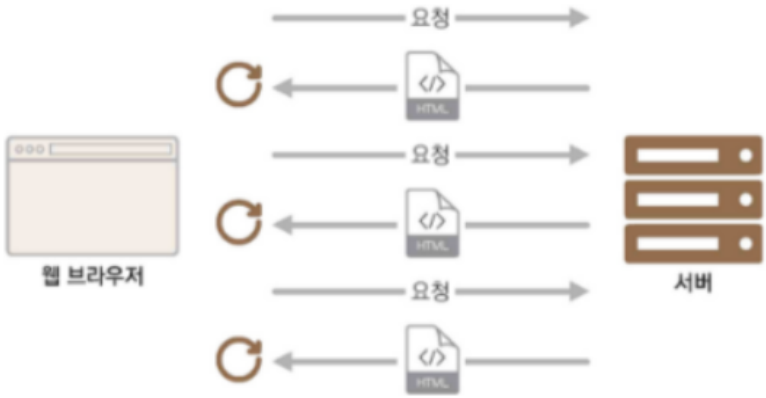
- 사용자가 요청한 URL에 따라 알맞은 페이지를 보여주는것을 의미
- 다른 주소에 다른 화면을 보여주는 것
- 라우팅 시스템: 여러 페이지로 구성된 웹 애플리캐이션을 만들때 페이지 별로 컴포넌트들을 분리해가면서 프로젝트를 관리하기 위해 필요한 것
- 리액트 라우터
  - 리액트의 라우팅 관련 라이브러리들 중에서 가장 오래됐고 가장 많이 사용되고 있음
  - 컴포넌트 기반으로 라우팅 시스ㅔㅁ을 설정
- Next.js
  - 리액트 프로젝트의 프레임워크
  - 프로젝트를 설정하는 기능,라우팅 시스템,최적화,다국어 시스템 지원,서버 사이드 렌더링 등 다양한 기능을 제공
  - 라우팅 시스템은 파일 경로 기반으로 작동함
  - 리액트 라우터의 대안으로 많이 사용

싱글 페이지 애플리케이션 (SPA)

- 하나의 페이지로 이루어진 애플리케이션
- html은 한번만 받아와서 웹 애플리케이션을 실행시킨 후, 이후에는 필요한 데이터만 받아와서 화면에 업데이트하는 것
- 기술적으로는 한 페이지만 존재하지만, 사용자가 경험하기에는 여러 페이지가 존재하는 것처럼 느낄 수 있음
- 링크를 눌러 다른 페이지로 이동할 때 서버에 다른 페이지의 html을 요청하는 것이 아닌 브라우저의 History API를 사용해 브라우저의 주소창의 값만 변경하고 기존에 페이지에 띄웠던 웹 애플리케이션을 그대로 유지하면서 라우팅 설정에 따라 또 다른 페이지를 보여주게됨
- SPA 방식
  - 뷰 렌더링을 사용자의 브라우저가 담당하도록 하고 우선 웹 애플리케이션을 브라우저에 불러와서 실행→ 사용자와의 인터랙션이 발생하면 필요한 부분만 JS를 사용해 업데이트하는 방식
- Ajax 방식
  - 새로운 데이터가 필요하다면 서버 API를 호출해 필요한 데이터만 새로 불러와 애플리케이션에서 사용
- yarn add react-router-dom 패키지 설치



- 멀티 페이지 애플리케이션



- 사용자가 다른 페이지로 이동할때 마다 새로운 html을 받아오고, 페이지를 로딩할때 마다 서버에서 서버에서 CSS,JS,이미지 파일등의 리소스를 전달받아 브라우저 화면에 보여줌
- 각 페이지마다 다른 html파일을 만들어 제공하거나 데이터에 따라 유동적인 html을 생성해주는 템플릿 엔진을 사용함
- 웹 서버라는 소프트웨어 웹 브라우저가 접속함→ 이때 전달되는 URL은 웹 서버가 관리하는 폴더에 저장되어있는 Html파일의 경로를 의미
  ⇒즉, 웹 브라우저가 웹 서버에 저장되어있는 웹 페이지를 열람,
  페이지 이동시 마다 접속,해제가 이루어짐
- 정적인 페이지들은 기존의 방식이 적합
- 사용자 인터랙션이 많고 다양한 정보를 제공하는 모던 웹 애플리케이션은 이 방식이 적합하지 않음
  - 새로운 페이지를 보여줘야 할때마다 서버 측에서 모든 준비를 한다면 그만큼 서버의 자원을 사용하는 것이고, 트래픽도 더 많이 나올수 있기 때문

프로젝트에 라우터 적용

- 라우터를 적용할때는 src/index.js파일에 react-router-dom에 내장되어있는 'BrowserRouter'라는 컴포넌트를 사용해 감싸면됨
  - 웹 애플리케이션에 html5의 History.API를 사용해 페이지를 새로 불러오지 않고도 주소를 변경하고 현재 주소의 경로에 관한 정보를 리액트 컴포넌트에서 사용할수 있도록 해줌

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

ReactDOM.render(
  <BrowerRouter>
    <App/>
  </BrowerRouter>
document.getElementById('root')
);
```

- Route 컴포넌트로 특정 경로에 원하는 컴포넌트 보여주기
  - <Route path='주소규칙' element={보여줄 컴포넌트 JSX}/>
  - Routes 컴포넌트 내부에서 Route 사용
  - Route하나에 여러개 path설정- 여러개의 path에 같은 컴포넌트 보여주기

```
<Route path='/about' element={About}/>
<Route path='/info' element={About}/>
```

- Link 컴포넌트
  - 다른 페이지로 이동하는 링크
  - 웹 페이지에서는 원래 링크를 보여줄 때 a태그를 사용하지만 리액트 라우터에서 a태그를 바로 사용하면 안됨- a태그 클릭시 페이지를 이동할대 브라우저는 페이지를 새로 불러오기 때문
  - Link컴포넌트도 a태그를 사용하지만 페이지를 새로 불러오는것을 막고 History API를 통해 브라우저 주소의 경로만 바꾸는 기능이 내장되어있음
  - <Link to='경로'> </Link>

src > ex > pages > JS Home.js > [e] default

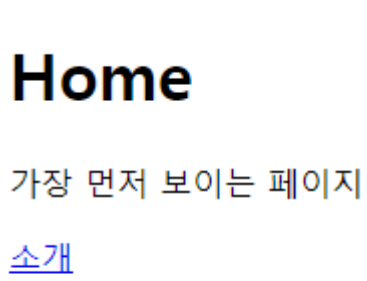
```jsx
import React from 'react';
import {Link} from 'react-router-dom';

const Home = () => {
  return (
    <div>
        <h1>Home</h1>
        <p>가장 먼저 보이는 페이지</p>
        <Link to='/about'>소개</Link>
    </div>
  )
}

export default Home;
```

c > ex > pages > JS About.js > [e] default

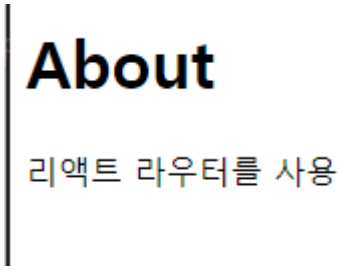
```jsx
import React from 'react'

const About = () => {
  return (
    <div>
        <h1>About</h1>
        <p>리액트 라우터를 사용</p>
    </div>
  )
}

export default About;
```

출력 결과

초기 화면



소개 링크 클릭시



URL 파라미터와 쿼리 스트링

URL 파라미터

- 주소의 경로에 유동적인 값을 넣은 형태
- 주로 ID 또는 이름을 사용해 특정 데이터 조회할때 사용
- ex) /profile/:username, /profile/velopert
  URL 파라미터가 여러개인경우- /profile/:username:field
- useParams Hook사용- URL파라미터가 저장되어있는 객체를 리턴받을수 있다.

- 이름 설정은 라우트를 설정할때 Route컴포넌트의 path props를 통해 설정

```jsx
import React from "react";
import { Route, Routes } from "react-router-dom";
import Home from "./ex/pages/Home";
import About from "./ex/pages/About";
import URLparameter from "./ex/pages/URLparameter";

function App() {
  return (
    <div>
      <Routes>
        <Route path='/' exact={true} element={<Home/>}/>
        <Route path='/about' element={<About/>}/>
        <Route path='/profiles/:username' element={<URLparameter/>}/>
      </Routes>
    </div>
  );
}

export default App;
```

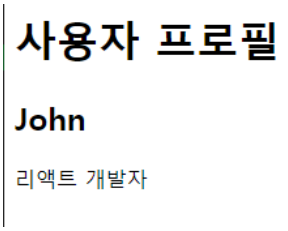
```jsx
import React from "react";
import { Link } from "react-router-dom";

const Home = () => {
  return (
    <div>
      <h1>Home</h1>
      <p>가장 먼저 보이는 페이지</p>
      <ul>
        <li>
          <Link to="/about">소개</Link>
        </li>
        <li>
          <Link to="/profiles/velopert">velopert 프로필</Link>
        </li>
        <li>
          <Link to="/profiles/gildong">gildong 프로필</Link>
        </li>
        <li>
          <Link to="/profiles/void">존재하지 않는 프로필</Link>
        </li>
      </ul>
    </div>
  );
};

export default Home;
```

```jsx
import React from 'react'
import {useParams} from 'react-router-dom';

const data={
    velopert:{
        name: 'John',
        description: '리액트 개발자'
    },
    gildong:{
        name: '홍길동',
        description: '홍길동전 주인공'
    }
}
const URLparameter = () => {
    const params= useParams();
    const profile= data[params.username];
  return (
    <div>
        <h1>사용자 프로필</h1>
        {profile ?(
        <div>
            <h2>{profile.name}</h2>
            <p>{profile.description}</p>
        </div>
        ): (
            <p>존재하지 않는 프로필 입니다.</p>
        )}
    </div>
  )
}

export default URLparameter;
```

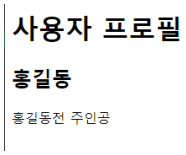
출력 결과

초기 화면

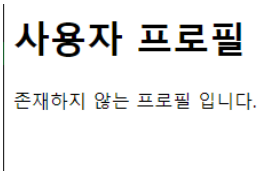
velopert 프로필 클릭시



gildong 프로필 클릭시



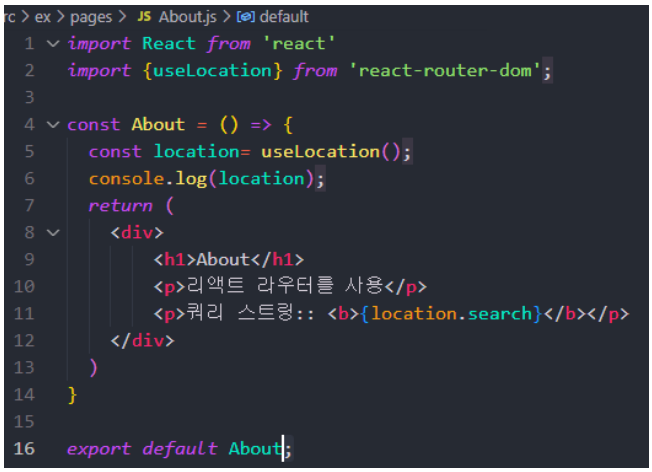
존재하지 않는 프로필 클릭시

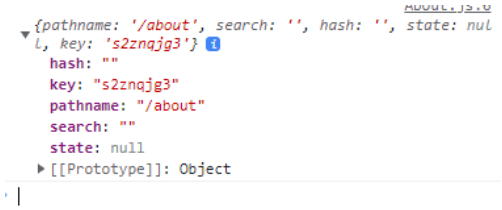


쿼리 스트링

- 주소의 뒷부분에 ? 문자열 이후에 key=value로 값을 정의하며 &로 구분하는 형태
- 키워드 검색,페이지네이션,정령 방식 등 데이터 조회에 필요한 옵션을 전달할때 사용
- URL 파라미터와 달리 Route 컴포넌트를 사용할때 별도로 설정해야하는것이 없음
- useLocation()
  - location객체를 반환
  - 현재 사용자가 보고있는 페이지의 정보를 지니고 있음
  - pathname: 현재 주소의 경로(쿼리스트링 제외)
  - search: 맨 앞의 ? 문자를 포함한 쿼리스트링 값
  - hash: 주소의 # 문자열 뒤의 값
    (주로 History API가 지원되지 않는 구형 브라우저에서 클라이언트 라우팅을 사용할때 쓰는 해시 라우터에서 사용됨)
  - state: 페이지로 이동할때 임의로 넣을수있는 상태값

    - key: location객체의 고유값, 초기에는 default이며 페이지가 변경될 때마다 고유의 값이 생성됨
- location.search값을 통해 조회 가능

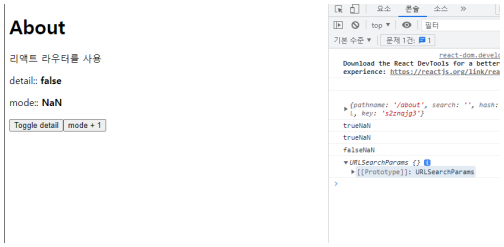




책의 예제대로 했지만 search값은 빈 문자열로 값이 화면에 나타나지 않음

- 쿼리스트링값이 제대로 나오면 문자열 앞에 있는 ?를 지우고,&문자열로 분리한 뒤 key와 value를 파싱하는 작업을 해야했지만 **useSearchParams**라는 Hook으로 처리 가능
- useSearchParams
    - 배열 타입의 값을 반환
    - 첫번째 원소: 쿼리파라미터를 조회하거나 수정하는 메서드들이 담긴 객체를 반환
        - get메서드를 통해 특정 쿼리 파라미터를 조회할수 있음
        - set메서드를 통해 특정 쿼리 파라미터를 업데이트 할수있음
        - 만약 조회시 쿼리 파라미터가 존재하지 않는다면 null로 조회됨
    - 두번째 원소: 쿼리 파라미터를 객체 형태로 업데이트할수 있는 함수를 반환
    - 쿼리 파라미터를 사용할때 주의할점
        - 조회할때 값은 문자열 타입
        - true 또는 false값을 넣는다면 값을 비교할때 꼭 ‘’따옴표로 감싸서 비교해야하며 숫자를 다룬다면 paseInt를 사용해 숫자 타입으로 변환 해야함

```
rc > ex > pages > JS About.js > [@] About
1  import React from 'react'
2  import {useSearchParams} from 'react-router-dom';
3
4  const About = () => {
5    const [searchParams, setSearchParams] = useSearchParams();
6    const detail= searchParams.get('detail');
7    const mode= searchParams.get('mode');
8    console.log(searchParams);
9    const onToggleDetail= ()=> {
10     setSearchParams({mode, detail: detail=== 'true' ? false : true});
11   }
12   const onOncereaseMode= ()=> {
13     const nextMode= mode=== null ? 1 : parseInt(mode)+ 1;
14     setSearchParams({mode: nextMode, detail})
15   }
16   return (
17     <div>
18         <h1>About</h1>
19         <p>리액트 라우터를 사용</p>
20         <p>detail:: <b>{detail}</b></p>
21         <p>mode:: <b>{mode}</b></p>
22         <button onClick={onToggleDetail}> Toggle detail</button>
23         <button onClick={onOncereaseMode}> mode + 1</button>
24     </div>
25   )
26 }
27
28 export default About;
```



중첩 라우트

- 서브 라우트
  - 라우트 내부에 또 라우트를 정의
  - 라우트로 사용되고있는 컴포넌트의 내부에 라우트를 또 사용한것

```
<Route path='/profiles' exact render={()=><div>Hi</div>}
```

- props를 설정할때 값 생략: 자동으로 true로 설정됨
  - exact⇒ exact={true}
- render()사용: path의 url의태그를 노출 시킴
  - 컴포넌트 자체를 전달하는것이 아닌 보여주고 싶은 JSX를 넣음

```jsx
import React from "react";
import { Route, Routes } from "react-router-dom";
import Home from "./ex/pages/Home";
import About from "./ex/pages/About";
import URLparameter from "./ex/pages/URLparameter";
import Articles from "./ex/pages/Articles"
import InArticles from "./ex/pages/InArticles"

function App() {
  return (
    <div>
      <Routes>
        <Route path='/' exact={true} element={<Home/>}/>
        <Route path='/about' element={<About/>}/>
        <Route path='/profiles/:username' element={<URLparameter/>}/>
        <Route path='/articles' element={<Articles/>}/>
        <Route path='/articles/:id' element={<InArticles/>}/>

      </Routes>
    </div>
  );
}

export default App;
```

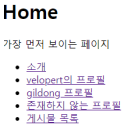
```jsx
import React from "react";
import { Link } from "react-router-dom";

const Home = () => {
  return (
    <div>
      <h1>Home</h1>
      <p>가장 먼저 보이는 페이지</p>
      <ul>
        <li>
          <Link to="/about">소개</Link>
        </li>
        <li>
          <Link to="/profiles/velopert">velopert의 프로필</Link>
        </li>
        <li>
          <Link to="/profiles/gildong">gildong 프로필</Link>
        </li>
        <li>
          <Link to="/profiles/void">존재하지 않는 프로필</Link>
        </li>
        <li>
          <Link to="/articles/">게시물 목록</Link>
        </li>
      </ul>
    </div>
  );
};

export default Home;
```

```jsx
import React from 'react'
import {Link} from 'react-router-dom';

const Articles = () => {
  return (
    <div>
        <ul>
            <li>
                <Link to='/articles/1'> 게시글 1</Link>
            </li>
            <li>
                <Link to='/articles/2'> 게시글 2</Link>
            </li>
            <li>
                <Link to='/articles/3'> 게시글 3</Link>
            </li>
        </ul>
    </div>
  )
}

export default Articles
```

```jsx
import React from 'react'
import {useParams} from 'react-router-dom';

const InArticles = () => {
    const {id}= useParams();
  return (
    <div><h2>게시글 <b>{id}</b></h2></div>
  )
}

export default InArticles
```

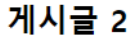
출력 결과

초기 화면

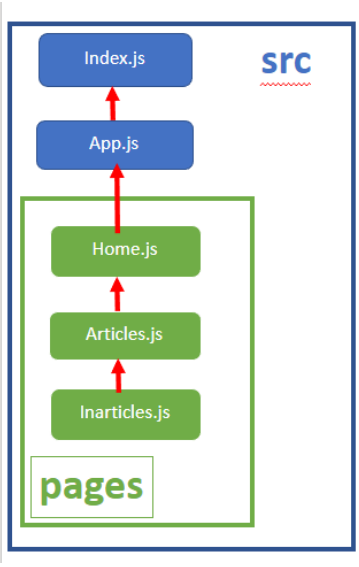


게시물 목록 클릭시

게시글 클릭시



DOM구조

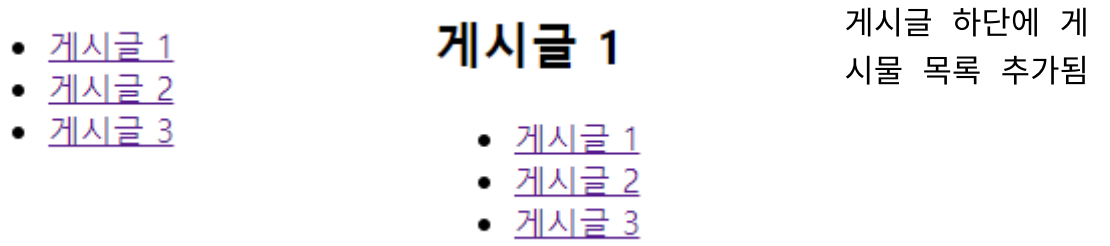


- Outlet 컴포넌트 사용
  - 라우트의 children으로 들어가는 JSX element를 보여주는 역할

```
<Route path="/articles" element={<Articles />}>
  <Route path=":id" element={<InArticles />} />
</Route>
```

```
import React from 'react'
import {Link, Outlet} from 'react-router-dom';

const Articles = () => {
  return (
    <div>
        <Outlet/>
        <ul>
            <li>
                <Link to='/articles/1'> 게시글 1</Link>
            </li>
            <li>
                <Link to='/articles/2'> 게시글 2</Link>
            </li>
            <li>
                <Link to='/articles/3'> 게시글 3</Link>
            </li>
        </ul>
    </div>
  )
}

export default Articles
```

게시물 목록 클릭→



공통된 레이아웃 컴포넌트

- 중첩된 라우트와 Outlet은 페이지끼리 공통적으로 보여줘야하는 레이아웃이 있을때도 유용하게 사용
- 중첩된 라우트를 사용하는 방식을 사용하면 컴포넌트를 한번만 사용해도 된다는 장점이 있음

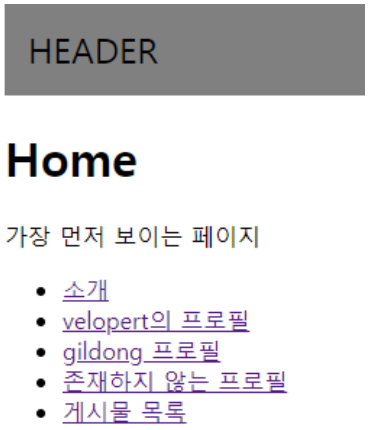
App.js > App

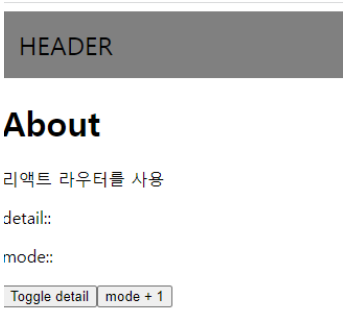
```jsx
import React from "react";
import { Route, Routes } from "react-router-dom";
import Home from "./ex/pages/Home";
import About from "./ex/pages/About";
import URLparameter from "./ex/pages/URLparameter";
import Articles from "./ex/pages/Articles";
import InArticles from "./ex/pages/InArticles";
import Layout from "./styled/Layout";

function App() {
  return (
    <div>
      <Routes>
        <Route element={<Layout />}>
          <Route path="/" exact={true} element={<Home />} />
          <Route path="/about" element={<About />} />
          <Route path="/profiles/:username" element={<URLparame
        </Route>      You, 1분 전 • Uncommitted changes
        <Route path="/articles" element={<Articles />}>
          <Route path=":id" element={<InArticles />} />
        </Route>
      </Routes>
    </div>
  );
}

export default App;
```

styled > Layout.js > default

```jsx
import React from 'react';
import {Outlet} from 'react-router-dom';

const Layout = () => {
  return (
    <div>
        <header style={{
            background:'Gray',padding: '16px', fontSize:'24px'
        }}>HEADER</header>
        <main>
            <Outlet/>
        </main>
    </div>
  )
}

export default Layout
```

출력 결과



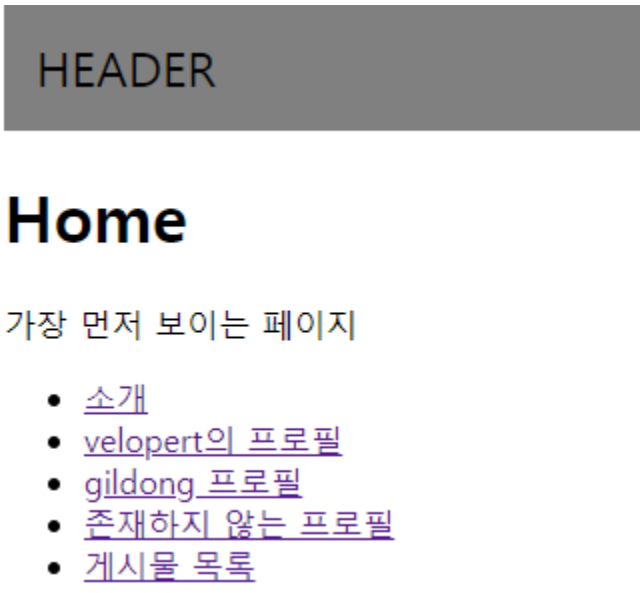


index props

- Router 컴포넌트에는 index라는 porps가 존재
- path=’/’와 동일한 역할,이를 좀더 명시적으로 표현하는 방법
- 사용하면 상위 라우트의경로와 일치하지만, 그 이후에 경로가 주어지지 않았을때 보여지는 라우트를 설정할수 있음

```
<Route index exact={true} element={<Home />} />
```

path=’/’ ⇒ index로 변경했을시 출력 화면



리액트 라우터 부가 기능

useNavigate

- Link컴포넌트를 사용하지 않고 다른 페이지로 이동하는 상황에 사용하는 hook

```
styled > JS Layout.js > [e] default
1  import React from 'react';
2  import {Outlet, useNavigate} from 'react-router-dom';
3
4  const Layout = () => {
5      const navigate= useNavigate();
6      const goBack= () =>{
7          //이전 페이지로 이동
8          navigate(-1);
9      }
10     const goArticles= () =>{
11         //articles 경로로 이동
12         navigate('/articles');
13     }
14   return (
15     <div>
16         <header style={{
17             background:'Gray',padding: '16px', fontSize:'24px'
18         }}>HEADER
19         <button onClick={goBack}>뒤로가기</button>
20         <button onClick={goArticles}>게시글 목록</button>
21         </header>
22         <main>
23             <Outlet/>
24         </main>
25     </div>
26   )
27 }
28
29 export default Layout
```

출력 결과

Home

가장 먼저 보이는 페이지

- 소개
- velopert의 프로필
- gildong 프로필
- 존재하지 않는 프로필
- 게시물 목록

- navigate()함수를 사용할때 파라미터가 숫자 타입일 경우
  - navigate(-1): 뒤로 한번
  - navigate(-2): 뒤로 두번
  - navigate(1): 앞으로 한번- 뒤로가기를 한번 한 상태에서
- replace
  - 다른 페이지로 이동할때 옵션
  - 페이지를 이동할때 현재 페이지를 페이지 기록에 남기지 않음

```
}
const goArticles= () =>{
    //articles 경로로 이동
    navigate('/articles',{replace: true});
}       You, 51분 전 • '.' …
```

- Home→ 소개 링크 클릭→ 상단의 게시글 목록 버튼 클릭→뒤로가기 버튼 클릭
  ⇒Home이 나타남, {replace:true}설정이 없다면 About이 나타남

NavLink

- 링크에서 사용하는 경로가 현재 라우트의 경로와 일치하는 경우 특정 스타일 또는 CSS클래스를 적용하는 컴포넌트
- 링크가 활성화되어있을 때의 스타일 적용시: activeStyle값을..?
- CSS클래스 적용할시: activeClassName값을 porps로 넣어줌
  - 더 쉬운 방법: 활성화 되었을때 active클래스가 자동 적용되므로 active 클래스에 대한 Global CSS만 미리 정의해두면됨

- 현재 NavLink를 감싼 또 다른 컴포넌트를 만들어 리렉터링하여 사용하는것을 권장

NotFind 페이지

- 사전에 저의되지 않는 경로에 사용자가 진입했을때 보여주는 페이지- 페이지를 찾을수 없을때 나타나는 페이지
- '*'
  - wildcard문자
  - 아무 텍스트나 매칭한다는 뜻
- 이 라우트의 상단에 위피하는 라우트들의 규칙을 모두 확인후 일치하는 라우트가 없다면 이 라우트가 화면에 나타남

```jsx
function App() {
  return (
    <div>
      <Routes>
        <Route element={<Layout />}>
          <Route index exact={true} element={<Home />} />
          <Route path="/about" element={<About />} />
          <Route path="/profiles/:username" element={<URLparameter />} />
        </Route>
        <Route path="/articles" element={<Articles />}>
          <Route path=":id" element={<InArticles />} />
        </Route>
        <Route path='*' element={<NotFind/>}/>
      </Routes>
    </div>
  );
}
```

```jsx
import React from 'react';

const NotFind = () => {
  return (
    <div style={{
        display: 'flex',
        alignItems:'center',
        justifyContent: 'center',
        fontSize: '64px',
        position: 'absolute',
        width: '100%',
        height: '100%'
    }}>
        404 Not Found
    </div>
  )
}

export default NotFind
```

Navigate 컴포넌트

- 컴포넌트를 화면에 보여주는 순간 다른 페이지로 이동하고 싶을때 사용하는 컴포넌트- 페이지를 리다이렉트 하고싶을때 사용
  - ex)사용자의 로그인이 필요한 페이지인데 로그인을 하지 않았을경우 로그인 페이지를 보여줄때

```jsx
import React from 'react'

const Login = () => {
  return (
    <div>Login</div>
  )
}

export default Login;
```

```jsx
import React from 'react';
import { Navigate } from 'react-router-dom';

const Mypages = () => {
    const isLogin= false;
    if(!isLogin) {
        return <Navigate to='/login' replace={true}/>
    }
  return (
    <div>Mypages</div>
  )
}

export default Mypages
```

- 로그인 상태에 따라 isLogin값이 true or false를 가리킨다고 가정
- false라면 Navigate컴포넌트를 통해 /login경로로 이동

- replace props는 페이지를 이동할때 현제 페이지를 기록에 남기지 않기때문에 이동후 뒤로가기를 눌렀을때 두 페이지 전의 페이지로 이동함
- localhost:3000/login 을 주소창에 입력시 login페이지로 이동함

SPA의 단점

- 큰 규모의 프로젝트 진행시 자바스크립트 파일의 크기가 매우 커짐- 최초 접속시 모든 화면을 구성하는 스크립트를 한꺼번에 모두 로딩해야하기 때문
- 라우트에 따라 필요한 컴포넌트만 불러오고, 다른 컴포넌트는 다른 페이지로 이동하는 등 필요한 시점에 불러오는것이 효율적
  - 코드 스플리팅이라는 기술로 해결가능(소스 파일 분할)
- 리액트 라우터처럼 브라우저에서 JS를 사용해 라우팅을 관리하는것은 JS를 실행하지 않는 일반 크롤러에서는 페이지의 정보를 제대로 수집하지 못한다는 잠재적인 단점이 따름- 구글만 가능
- 검색엔진의 검색 결과에 페이지가 잘 나타나지 않을수도 있음
  - 검색 엔진에 쥐약이다
  - →서버사이드 렌더링을 통해 해결 가능- Next.js를 통해