React Part2

form 요소 사용자 정보 입력받기
useRef hook 사용하기
페이지 라우팅

```jsx
function App() {
  let [data, setData] = useState('');

  return (
    <>
      <div>
        <input
          type="text"
          value={data}
          onChange={(e) => {
            setData(e.target.value);
          }}
        />
      </div>
      <p>입력받은 값 : {data}</p>
    </>
  )
}
```

- input 태그에 입력한 값을 저장하기 위한 state 변수를 생성한다.
- input태그의 값이 변경될때마다 실행되는 이벤트를 처리하는 onChange 이벤트 핸들러 속성을 추가한다.
- 이벤트 발생마다(input 태그의 값 변경이 일어날때마다) state변경함수를 호출하여 state변수의 값을 업데이트 한다.
- input태그에 입력된 값은 모든 이벤트정보가 담겨있는 이벤트 객체로부터 받을 수 있다.
- e.target.value는 이벤트가 발생한 태그의 value속성값을 의미한다.
- value 속성값은 input 태그에 입력된 데이터다.

```jsx
function App() {
  let [info, setInfo] = useState({
    name : '',
    age : '',
    addr : ''
  });

  function changeInfo(e){
    setInfo({
      ...info,
      [e.target.name] : e.target.value
    });
  }

  return (
    <>
      <div>
        <input type="text" name='name' value={info.name}
          onChange={changeInfo} />
      </div>
      <p>입력받은 값 : {info.name} {info.age} {info.addr}</p>
    </>
  )
}
```

- 객체를 사용하여 입력받은 여러 정보를 한 번에 관리할 수 있다.
- 객체의 key값과 input 태그의 name 속성은 반드시 일치시킨다.
- state 변경함수의 인자에 spread 연산자를 사용하면 특정 데이터만 업데이트 할 수 있다.
- e.target.name은 이벤트가 발생한 태그의 name 속성값이다.
- e.target.value는 이벤트가 발생한 태그의 value 속성값이다.
- 객체의 key값을 변수로 사용 할 때는 []에 감싼다.

ex) info.name 과 info['name']은 같다.
const text = 'name' 변수가 생성할 때
info[text];문법은 key가 변수일 때 사용 가능하지만,
info.text은 불가능하다.

```jsx
function App() {
  let [fruit, setFruit] = useState('orange');

  function changeFruit(e){
    setFruit(e.target.value);
  }

  return (
    <>
      <div>
        <select value={fruit} onChange={changeFruit}>
          <option value="apple">사과</option>
          <option value="banana">바나나</option>
          <option value="orange">오렌지</option>
        </select>
      </div>
    </>
  )
}
```

- select 태그에 onChange 이벤트 핸들러 속성을 추가한다.
- option 태그에는 반드시 value 속성을 추가한다.
- option을 선택하면 화면에 보이는 글자가 아닌 value 속성의 값이 데이터로 저장된다.
- value 속성이 없다면 option태그 사이에 작성한 글자가 데이터로 저장된다.
- select의 value 속성으로 지정한 값이 최초 화면 렌더링 시 선택되는 값이다.

```
function App() {
  let [feel, setFeel] = useState('soso');

  function changeFeel(e){
    setFeel(e.target.value);
  }

  return (
    <>
      <div>
        <input type="radio" value={'sad'}
        onChange={changeFeel} checked={feel === 'sad'}/>슬픔
        <input type="radio" value={'soso'}
        onChange={changeFeel} checked={feel === 'soso'}/>그럭저럭
        <input type="radio" value={'happy'}
        onChange={changeFeel} checked={feel === 'happy'}/>기쁨
      </div>
    </>
  )
}
```

- state변수의 초기값은 radio 태그 중 화면에 먼저 보여질 radio 태그의 value 속성값으로 초기화한다.
- radio를 체크하면 눈에 보이는 글자가 아닌 value 속성의 값이 데이터다.
- checked 속성의 연산결과 true로 판명되는 radio를 체크한다.

자바스크립트 부분

```
//체크박스를 그릴 데이터
const fruitList = ['사과', '바나나', '오렌지'];
//체크하여 선택된 데이터가 담길 state 변수
const [checkedItems, setCheckedItems] = useState([]);

//전체 체크박스 클릭 시 실행 함수
function checkAll(e){
  setCheckedItems(e.target.checked ? fruitList : []);
}

//각각의 체크박스 클릭 시 실행 함수
function checkItem(e){
  if(e.target.checked){
    setCheckedItems([...checkedItems, e.target.value]);
  }
  else{
    //배열의 filter 함수는 return에 작성한 조건에 맞는 데이터만 걸러낸다
    const copyItems = checkedItems.filter((item) => {return item !== e.target.value});
    setCheckedItems(copyItems);
  }
}
```

전체 체크박스

```
<input type="checkbox" onChange={(e) => {
  checkAll(e);
}}/>
```

각 행의 체크박스

```
<tbody>
  {
    fruitList.map((item, index) => {
      return (
        <tr key={index}>
          <td>
            <input type="checkbox" value={item}
              checked={checkedItems.includes(item)}
              onChange={(e) => {checkItem(e)}}/>
          </td>
          <td>{item}</td>
        </tr>
      )
    })
  }
</tbody>
```

```
const [data1, setData1] = useState(1);
const [data2, setData2] = useState([1,2,3]);
const [data3, setData3] = useState({
  name : 'kim',
  age : 20
});

const data4 = useRef(1);
const data5 = useRef([1,2,3]);
const data6 = useRef({
  name : 'kim',
  age : 20
});

console.log(data4);
console.log(data5);
console.log(data6);
```

```
▶ {current: 1}
▶ {current: Array(3)}
▶ {current: {…}}
```

- useRef hook을 사용하면 useState와 마찬가지로 데이터를 저장할 수 있는 변수를 만들 수 있다.
- useState로 만들어진 변수와의 차이점은 값이 변경되도 component가 리렌더링되지 않는다는 점이다.
- useRef로 만들어진 변수는 객체 형태로 저장되며, key 값으로 current를 갖는다.
- 기본자료형의 변수와 useRef로 선언한 변수의 차이점의 리렌더링 시 값 초기화 여부이다. useRef로 선언한 변수는 리렌더링되어도 값이 초기화되지 않는다.
- 그렇기 때문에 component가 리렌더링되더라도 값을 지속적으로 관리하고 싶은 데이터는 useRef를 사용하여 선언한다.

```jsx
function App() {
  const input1_tag = useRef();
  const [data, setData] = useState({
    tag_type : '',
    tag_id :'',
    tag_className : '',
    tag_value : ''
  });

  return (
    <>
      <input type="text" ref={input1_tag} id='aaa' className='bbb' />

      <button type='button' onClick={(e) => {
        setData({
          tag_type : input1_tag.current.type,
          tag_id :input1_tag.current.id,
          tag_className : input1_tag.current.className,
          tag_value : input1_tag.current.value
        });
      }}>input 태그 값 읽기!</button>

      <div>
        input 태그의 type 속성값 : {data.tag_type} <br />
        input 태그의 id 속성값 : {data.tag_id} <br />
        input 태그의 className 속성값 : {data.tag_className} <br />
        input 태그의 value 속성값 : {data.tag_value}
      </div>
    </>
  )
}
```

- useRef를 사용하면 다른 태그의 값을 참고할 수 있다.
- useRef로 변수 선언 시 초기값을 주지 않고, 값을 참조할 태그의 ref 속성으로 useRef로 선언한 변수를 넣어준다.
- 이렇게 하면 useRef로 선언한 변수는 태그 자체를 값으로 갖는다.
- 참조하는 태그의 모든 속성값은 객체의 데이터를 읽는 문법과 동일하다.
- input 태그에 입력한 내용은 value 속성값으로 접근할 수 있다.

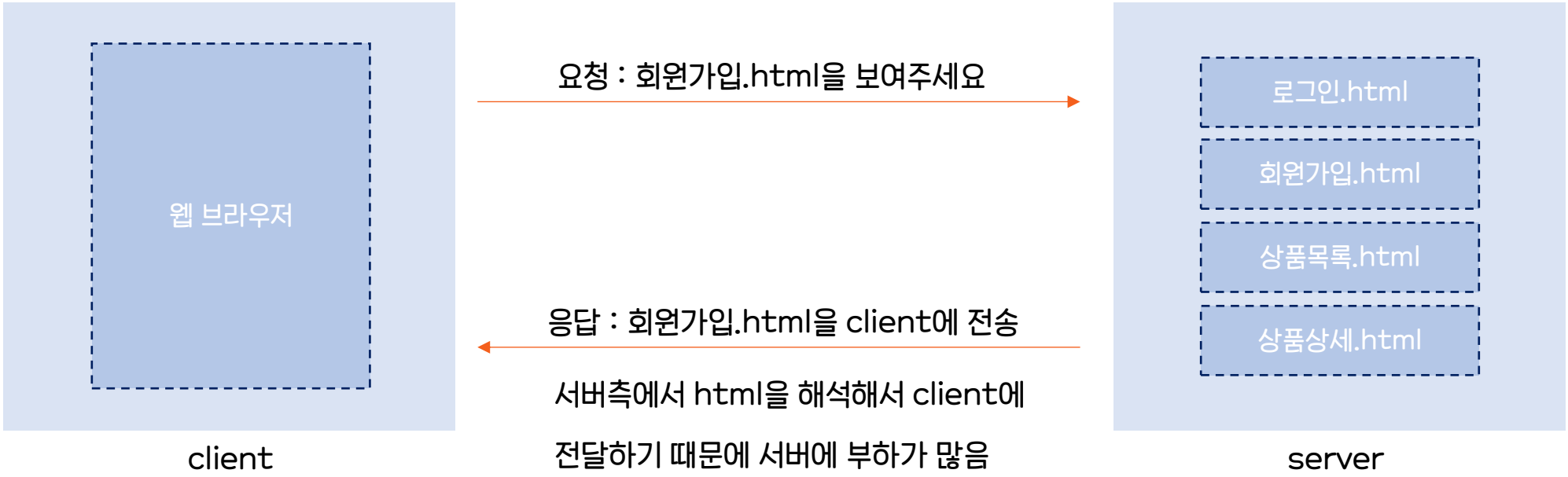
페이지 라우팅이란 요청 경로(url)에 따라 알맞은 페이지를 렌더링하는 과정을 말한다.

리액트를 사용하기 이전의 웹 싸이트는 각 화면에 대응하면 html 파일을 여러 개 만들었다.

이러한 방식을 MPA(Multi Page Application)이라 한다.

이 방식은 클라이언트가 page를 요청하면 기존의 html을 요청한 html파일로 완전 교체하기 때문에 화면에 깜빡임이 발생한다.

또한 MPA 방식은 전통적이며 가장 직관적인 웹 싸이트 제작 방식이지만 서버 사이드 렌더링 방식을 사용하기 때문에 서버에 부하가 많이 생긴다.



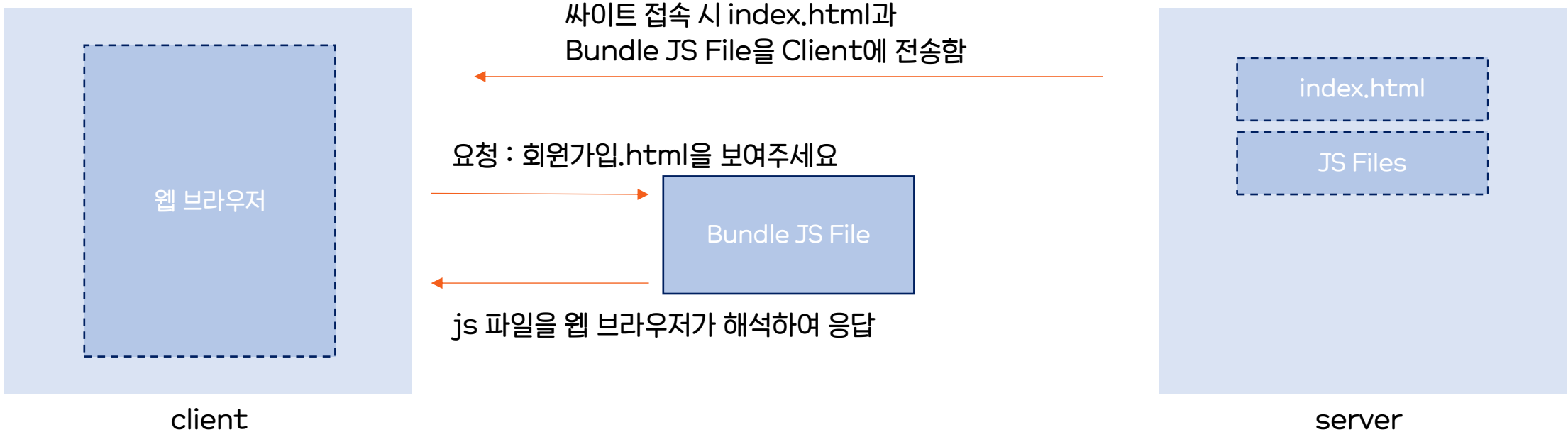
리액트는 SPA(Single Page Application)으로 html 파일을 하나만 갖는다.

리액트로 구현한 웹 싸이트는 client가 싸이트에 접속하면 빈 화면인 index.html과 모든 js File들을 client에 전달한다.

모든 JS파일은 하나의 JS파일로 묶어서 클라이언트로 보내지는데 이러한 동작을 Bundling이라 하고, 하나도 통합된 파일을 Bundle JS File 이라 부른다.

이후 클라이언트가 특정 페이지를 요청하면 서버가 처리하는 것이 아니라, 클라이언트 측의 웹 브라우저가 이 요청을 처리한다.

이를 클라이언트 사이드 렌더링이라 하며, 매 요청을 서버가 처리하지 않기 때문에 서버에 과부하가 적게 발생한다.



리액트의 페이지 라우팅은 React Router 라이브러리를 사용한다.

터미널에 'npm install react-router-dom'을 입력하면 npmjs 싸이트에 접속하지 않아도 쉽게 설치할 수 있다.(프로젝트마다 설치)

```
createRoot(document.getElementById('root')).render(
  <BrowserRouter>
    <App />
  </BrowserRouter>
)
```

설치 후 라우팅을 사용하기 위해 우선 main.jsx의 <App /> 컴포넌트를 <BrowerRouter> 컴포넌트로 감싼다.

```
function App() {
  return (
    <>
      <h2>Hello React</h2>
      <Routes>
        <Route path='' element={<Home />}/>
        <Route path='/login' element={<Login />}/>
        <Route path='/join' element={<Join />}/>
      </Routes>
    </>
  )
}
```

다음으로 <Routes>와 <Route>를 컴포넌트 사용하여 라우팅을 처리할 수 있다.

<Route>컴포넌트 하나가 하나의 페이지를 표현하는 것이라 생각하면 된다.

왼쪽과 같은 경우 3개의 페이지를 표현한 것이다.

각각의 <route> 컴포넌트에는 path와 element 정보를 작성할 수 있다.

path는 라우팅 요청경로이며, element는 렌더링될 컴포넌트이다.

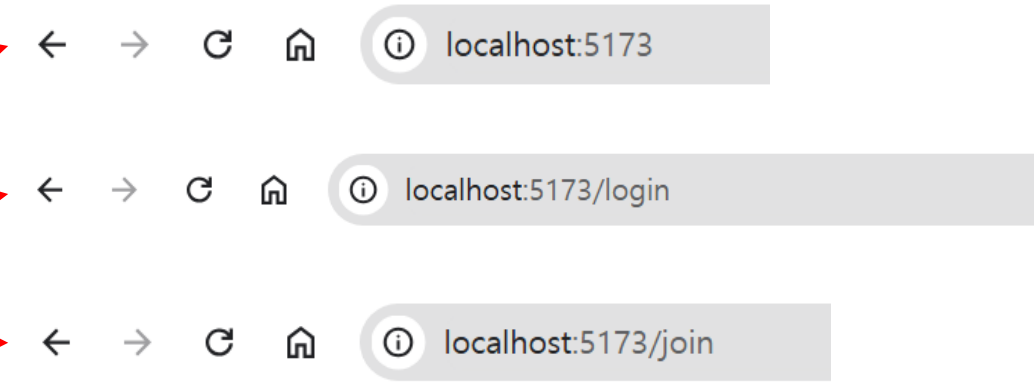
path='/login'이고 element={<Login/>}을 해석하면 '/login' 요청이 들어오면 Login컴포넌트를 렌더링하라는 의미이다.

다음장에 계속…

요청 경로란 웹 페이지에 접속하는 url(인터텟 접속 주소)을 말한다.

Route 컴포넌트에 작성한 path를 웹 브라우저의 url에 작성하면, 해당하는 Route 컴포넌트의 element 속성의 컴포넌트가 화면에 표현된다.

```jsx
function App() {
  return (
    <>
      <h2>Hello React</h2>
      <Routes>
        <Route path='' element={<Home />}/>
        <Route path='/login' element={<Login />}/>
        <Route path='/join' element={<Join />}/>
      </Routes>
    </>
  )
}
```




```jsx
<Routes>
  <Route path='' element={<Home />}/>
  <Route path='/login' element={<Login />}/>
  <Route path='/join' element={<Join />}/>
  <Route path='*' element={<NotFound />}/>
</Routes>
```

Route 컴포넌트의 path에 작성한 요청 경로와 일치하지 않는 url을 주소창에 입력하면 화면이 나오지 않는다. 이런 경우를 대비하여 path가 '*'인 Route를 작성한다.

path가 '*'인 Route 컴포넌트는 반드시 모든 Route 컴포넌트 중 마지막에 작성하여야 하며, 작성된 모든 Route의 path 경로를 제외한 나머지 요청경로를 전부 처리한다.

아래와 같이 코드를 작성하면 모든 요청 경로마다 공통된 부분을 계속해서 보여줄 수 있다.

```
function App() {
  return (
    <>
      <h2>Hello React</h2>
      <Routes>
        <Route path='' element={<Home />}/>
        <Route path='/login' element={<Login />}/>
        <Route path='/join' element={<Join />}/>
        <Route path='*' element={<NotFound />}/>
      </Routes>
      <Footer />
    </>
  )
}
```

이 두 부분은 Routes 컴포넌트 밖에 있기 때문에 모든 화면에서 표현 됨

Routes와 Route 컴포넌트를 사용하면 요청경로를 웹 브라우저의 주소창에 입력하여 원하는 컴포넌트만 화면에 보여줄 수 있었다.

하지만 우리가 원하는 컴포넌트만 보여주기 위해 매번 주소창에 요청 url을 입력하는 것은 바람직하지 못하다.

예를 들면, html의 <a> 태그를 사용하면 글자나 버튼 클릭 만으로 원하는 페이지를 화면에 띄울 수 있었다.

html의 이러한 기능을 react에서 사용하기 위해 Link 컴포넌트와 useNavigate hook을 사용한다.

이 두 기능은 모두 react-router-dom 라이브러리가 설치되어 있어야 사용가능하다.

```jsx
function App() {

  return (
    <>
      <div>
        <Link to={'/'}>HOME</Link>
        <Link to={'/login'}>LOGIN</Link>
        <Link to={'/join'}>JOIN</Link>
      </div>

      <h2>Hello React</h2>
      <Routes>
        <Route path='/' element={<Home />}/>
        <Route path='/login' element={<Login />}/>
        <Route path='/join' element={<Join />}/>
        <Route path='*' element={<NotFound />}/>
      </Routes>
      <Footer />
    </>
  )
}
```

- react에서는 html의 <a>태그를 사용하지 않는다.
- a태그 대신 react에서는 Link 컴포넌트를 사용한다.
  (Link 컴포넌트 안의 내용을 클릭하면 페이지 이동)
- Link의 to 속성에 요청경로를 작성 후, Link 컴포넌트 사이의 글자 클릭 시 페이지 라우팅이 처리된다.(페이지 이동과 같은 효과, 페이지 이동은 아님)
- Link 컴포넌트에 CSS를 적용하려면 a 태그를 선택자로 사용하여 작성하면 된다.

버튼 클릭 시 또는 함수 호출 시 페이지 라우팅은 Link 컴포넌트를 사용하면 될까?

예를 들어 아래 코드처럼 작성하면 버튼 클릭 시 페이지 이동이 될지 생각해보자.

```
function App() {

  return (
    <>
      <button type='button' onClick={(e) => {
        <Link to={'/login'}>Login</Link>
      }}>로그인 페이지로 이동</button>

      <h2>Hello React</h2>
      <Routes>
        <Route path='/' element={<Home />}/>
        <Route path='/login' element={<Login />}/>
        <Route path='/join' element={<Join />}/>
        <Route path='*' element={<NotFound />}/>
      </Routes>
      <Footer />
    </>
  )
}
```

버튼 클릭 혹은 함수 호출을 통해 라우팅을 처리하기 위해서 useNavigate hook을 사용한다.

```
function App() {
  const nav = useNavigate();

  return (
    <>
      <button type='button' onClick={(e) => {
        nav('/login');
      }}>login페이지로 이동</button>

      <h2>Hello React</h2>
      <Routes>
        <Route path='/' element={<Home />}/>
        <Route path='/login' element={<Login />}/>
        <Route path='/join' element={<Join />}/>
        <Route path='*' element={<NotFound />}/>
      </Routes>
      <Footer />
    </>
  )
}
```

- useNavigate()를 실행하면 함수를 리턴한다.
- 그렇기 때문에 useNavigate()함수의 결과를 받는 변수는 함수호출과 같은 문법으로 사용한다.
- 매개변수의 인자로 요청경로를 입력하면 페이지 라우팅을 구현할 수 있다.
- 또는 인자로 숫자를 전달하면 이전 페이지, 다음페이지로의 이동 기능을 사용할 수 있다.

ex>

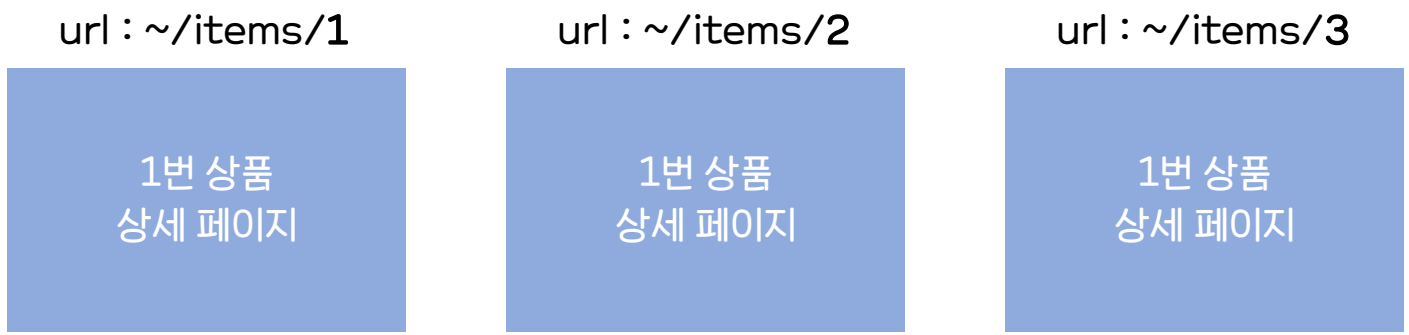
nav(-1) : 이전 페이지로 이동(뒤로 가기)

nav(-2) : 이전 이전 페이지로 이동

nav(+1) : 다음 페이지로 이동(앞으로 가기)

동적 경로(url)란 url이 고정되어 있지 않고, 필요에 따라 변화되는 것을 의미한다.

이러한 동적 경로는 주로 페이지 이동(라우팅 처리) 시 데이터를 전달하기 위해 사용한다.



- 위 이미지는 각 상품의 상세 정보를 보여주는 페이지 url을 동적으로 작성한 예시이다.
- REST API에서 학습한 것과 마찬가지로 위 url에서 1,2,3은 상세페이지에서 보여줄 각 상품의 상품번호이다.
- 1,2,3 이라는 상품 번호 데이터를 url로 전달하는 것이며, 이렇게 데이터를 전달할 때 url이 변하게 된다. 이렇게 작성한 url을 동적 경로(url)라 한다.

이렇게 url를 통해 데이터를 전달하는 동적 경로는 데이터를 전달하는 방식에 따라 아래처럼 2가지로 구분할 수 있다.

두 방식 다 실무에서 많이씀

URL Parameter 방식

- REST API 방식처럼, '/'뒤에 전달할 데이터를 작성하는 방식이다.
- 주로 전달할 데이터가 1개일 때 사용한다.
- ex> localhost:5173/items/1, localhost:5173/items/2

Query String 방식

- url 뒤에 전달할 데이터를 'key=value' 형태로 작성하며, url과 전달 데이터를 '?'로 구분하여 url을 작성하는 방식이다.
- react를 사용하지 않을 때 많이 사용하는 방식이며, 전달할 데이터가 2개 이상일 때 URL Parameter 방식 대신 사용한다.

ex>

- localhost:5173/items?itemCode=1

  -> '/items' url로 이동하면서 itemCode라는 이름으로 1을 전달하겠다는 의미이다.
- localhost:5173/items?itemCode=1&itemName=상의

  -> '/items' url로 이동하면서 itemCode라는 이름으로 1을, itemName이라는 이름으로 '상의 ' 라는 2개의 데이터를 가져가겠다는 의미이다.

  전달한 데이터가 여러개일 경우, 위의 예시처럼 '&'로 연결한다.

URL Parameter를 이용하여 동적 라우팅을 처리하기 위해, 우선 Route 컴포넌트에 다음과 같이 코드를 작성해야 한다.

```
<Routes>
  <Route path='' element={<Home />}/>
  <Route path='/login/:num' element={<Login />}/>
  <Route path='/join' element={<Join />}/>
  <Route path='*' element={<NotFound />}/>
</Routes>
```

- path='/login/:num' 에서 ':num' 부분이 데이터를 받기 위해 작성한 코드이다.
- ':num'에서 'num'은 변수명이라 생각하면 된다.
- Spring에서 @GetMapping("/items/{num}")과 같은 기능이다.
- 만약 웹 브라우저에서 'localhost:5173/items/3 ' 이라 입력하면 num이라는 변수에 3이 전달되면서 Login 컴포넌트가 실행되는 것이다.

URL Parameter로 전달되는 데이터는 컴포넌트에서 받아 사용할 수 있다. 위 예시에서는 Login 컴포넌트를 보여주는 Route의 path에 URL Parameter를 작성하였기 때문에, num이라는 이름으로 전달되는 데이터는 Login 컴포넌트에서 받을 수 있다.

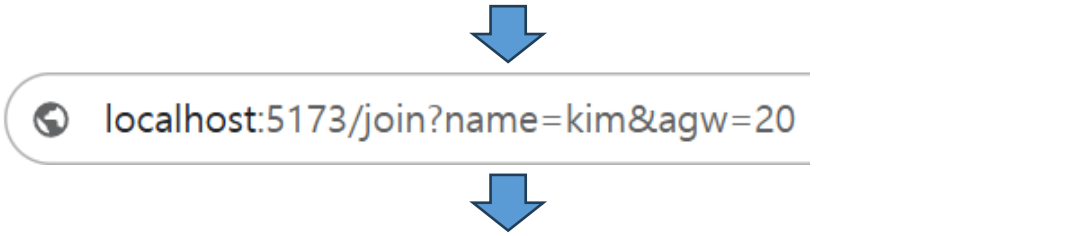
```
const Login = () => {
  const param = useParams();
  console.log(param);

  return (
    <>
      <div>Login 페이지</div>
      <div>
        URL Parameter로 전달받은 값 : {param.num}
      </div>
    </>
  )
}
```

- useParams() 훅을 사용하면 URL Parameter로 전달되는 데이터를 받을 수 있다.
- useParams()는 URL로 전달되는 데이터를 받아, 객체타입으로 리턴한다. (리턴값을 출력해보면 객체 형식임을 알 수 있을 것이다)
- 그렇기 때문에 useParams()의 리턴값은 일반적으로 구조분해할당 방식으로 작성한다.

Query String을 사용하여 동적 url을 사용할 때에는 URL Parameter처럼 Route에 무언가를 작성할 필요가 없다.

동적 url을 통해 데이터를 전달받는 컴포넌트에서 useSearchParams() 훅을 사용하여 Query String으로 전달된 데이터만 받아주면 된다.

```
<Route path='/join' element={<Join />}/>
```




```
const Join = () => {
  const [params, setParams] = useSearchParams();
  const name = params.get('name');
  const age = params.get('age');

  return (
    <>
      <div>Join 페이지</div>
      <div>Query String으로 전달받은 이름 : {name}</div>
      <div>Query String으로 전달받은 나이 : {age}</div>
    </>
  )
}
```

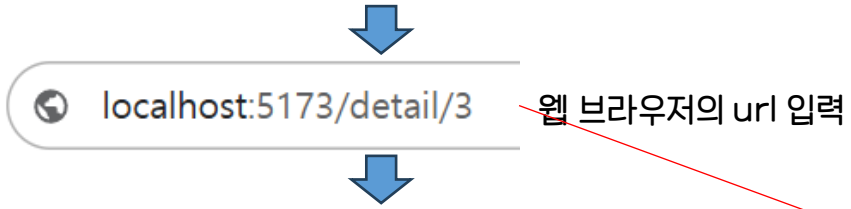
- Route 컴포넌트의 path 속성에 작성한 url을 웹 브라우저에 입력하면 Join 컴포넌트를 화면에 보여준다.
- 왼쪽의 예시는 Join컴포넌트로 라우팅(페이지 이동) 시 Query String 방식으로 name 키와 age 키에 각각 'kim', 20 데이터를 받아 Join컴포넌트로 전달한다.
- Query String으로 전달된 데이터는 데이터를 받는 컴포넌트에서 useSearchParams() 훅을 사용하면 받을 수 있다.
- useSearchParams()는 useState() 훅과 사용하면 문법이 동일하다.
  useSearchParams()는 배열 데이터를 리턴하며,
  배열의 첫번째 요소에는 Query String으로 전달된 데이터를 갖고 있으며,
  두번째 요소에는 전달된 데이터를 변경할 수 있는 함수가 있다.(잘 사용 안 함)
- params.get('name')은 name 키 값을 받은 코드이며,
  params.get('age')는 age 키 값을 받는 코드이다.

이번에는 URL Parameter 방식의 동적 경로 작성과 Query String 방식의 동적 경로 작성을 Spring과 연결지어 생각해보겠다.

URL Parameter 방식을 이용한 동적 경로를 Spring과 함께 사용하기

```
<Routes>
  <Route path='/detail/:itemCode' element={<ItemDetail />}/>
</Routes>
```

App 컴포넌트의 Route 설정 코드



웹 브라우저의 url 입력

```
const ItemDetail = () => {
  const {itemCode} = useParams();
  const [itemInfo, setItemInfo] = useState({});

  useEffect(() => {
    axios.get(`/api/items/${itemCode}`)
    .then((res) => {setItemInfo(res.data)})
    .catch();
  }, []);

  return (
    <><div>상품 상세 페이지</div></>
  )
}
```

```
@GetMapping("/items/{itemCode}")
public void getItemDetail(@PathVariable("itemCode") int itemCode){
    System.out.println("itemCode = " + itemCode); //3 출력
    //itemCode가 3번인 상품의 정보를 조회하여 리액트로 리턴하는 코드 작성
}
```

Spring Controller의 코드

- App 컴포넌트에 작성한 Route 코드에 의해 웹 브라우저에 '~/detail/3'을 입력하면 ItemDetail 컴포넌트가 실행된다.
- URL Parameter로 전달된 3을 useParams()로 받는다.
- axios를 사용하여 Spring과 통신하며, 이때 전달받은 3을 스프링에도 전달한다. get() 안에는 변수 사용을 위해 백틱으로 url을 작성하였다.
- Spring에서는 전달받은 3을 받아, 3번 상품의 정보를 디비에서 조회 후 조회한 데이터를 리액트로 리턴시켜주는 코드를 작성한다.
- 리액트에서는 Spring으로 받은 3번 상품의 정보를 화면에 뿌려준다.

Query String 방식을 이용한 동적 경로를 Spring과 함께 사용하기

```
<Routes>
  <Route path='/detail' element={<ItemDetail />}/>
</Routes>
```

App 컴포넌트의 Route 설정 코드

localhost:5173/detail?itemCode=3 웹 브라우저의 url 입력

```
const ItemDetail = () => {
  const [params, setParams] = useSearchParams();
  const [itemInfo, setItemInfo] = useState({});

  const itemCode = params.get('itemCode');

  useEffect(() => {
    axios.get(`/api/items?itemCode=${itemCode}`)
    .then((res) => {setItemInfo(res.data)})
    .catch();
  }, []);

  return (
    <><div>상품 상세 페이지</div></>
  )
}
```

```
@GetMapping("/items")
public void getItemDetail(@RequestParam("itemCode") int itemCode){
    System.out.println("itemCode = " + itemCode); //3 출력
    //itemCode가 3번인 상품의 정보를 조회하여 리액트로 리턴하는 코드 작성
}
```

Spring Controller의 코드 첫번째 방식 – @RequestParam 어노테이션 활용

```
@GetMapping("/items")
public void getItemDetail(ItemDTO itemDTO){
    System.out.println(itemDTO); //3 출력
    //itemCode가 3번인 상품의 정보를 조회하여 리액트로 리턴하는 코드 작성
}
```

Spring Controller의 코드 두번째 방식 – DTO 클래스 활용

- Query String방식은 Route에는 특별히 작성할 코드가 없으며, 웹 브라우저에서 url만 Query String 문법에 맞게 작성하면 된다.
- 전달되는 데이터는 useSearchParams()를 통해 받으며, get()함수 안에 전달되는 데이터의 키값을 작성되면 된다.
- Query String을 통해 Spring서버로 전달되는 데이터는 @RequestParam 어노테이션을 사용하여 받거나, DTO 클래스로 받을 수 있다. DTO 클래스로 데이터를 받을 때는 @RequestParam 어노테이션을 사용하지 않는다.