4장 API 연동하기

1. API 연동의 기본

* API를 호출하기 위해서는 axios라는 라이브러리를 설치해야합니다.
* REST API를 사용할때에는 하고 싶은 작업에 따라 다른 메서드로 요청할 수있다.
  + GET : 데이터 조회
  + POST : 데이터 등록
  + PUT : 데이터 수정
  + DELETE : 데이터 제거
* Axios의 사용법
  + Ex) import axios from ‘axios’;
  + Axios.get(‘/users/1’);
  + Axios.post() 로 데이터를 등록 할 때에는 두번째 파라미터에 등록하고자 하는 정보를 넣을 수있습니다.
  + Ex) axios.post(‘/users/’,{ username: ‘blabla’, name:’blabla’ });
* useState와 useEffect로 데이터 로딩하기
  + useState를 사용하여 요청 상태 관리하고, useEffect를 사용하여 컴포넌트가 렌더링되는 시점에 요청을 시작하는 작업을 해보겠습니다.
  + 상태 관리해야하는 3가지
  + 1. 요청의 결과 2. 로딩상태 3. 에러
  + useEffect에 첫번째로 파라미터로 등록하는 함수에는 async를 사용할 수없기 때문에 함수 내부에서 async를 사용하는 새로운 함수를 선언해주어야 합니다.

1. useReducer로 요청 상태 관리하기

* User 컴포넌트에서 useState 대신에 useReducer 를 사용해서 구현을 해보도록하겠습니다.
* useReducer를 사용하여 LOADING, SUCCESS, ERROR 액션에 따라 다르게 처리를 해봅시다.
* Fuction reducer(state, action){
  + Switch(action.type){
    - Case ‘LOADING’:
      * Return{
        + Loading : true,
        + Data : null,
        + Error : null
    - };
    - Case ‘SUCCESS’:
      * Return {
        + Loading : false,
        + Data : action.data,
        + Error : null
      * };
    - Case ‘ERROR’:
      * Return{
        + Loading : false,
        + Data : null,
        + Error : action.error
      * };
      * Default:
        + Throw new Error(`Unhandled action type: ${action.type}`);

}

}

Function Users(){

Const [state, dispatch] = useReducer(reducer, {

Loading:false,

Data : null,

Error : null

});

Const fetchUsers = async () => {

Dispatch({type: ‘LOADING’});

Try{

Const response = await axios.get(

‘https://jsonplaceholder.typicode.com/users’

);

Dispatch({type:’SUCCESS’, data: response.data});

} catch(e){

Dispatch({type:’ERROR’, error:e});

}

};

Const {loading, data:users, error} = state; // state.data 를 users키워드로 조회

...

1. useAsync 커스텀 Hook 만들어서 사용하기

* 매번 반복되는 코드를 작성하는 대신에, 커스텀 Hook을 만들어서 요청 상태 관리 로직을 쉽게 재사용하는 방법
* Ex) function useAsync(callback, deps=[]){
  + Const[state, dispatch] = useReducer(reducer, {
    - Loading: false,
    - Data: null,
    - Error ; false
  + });
* useEffect
* useAsync 함수는 두가지 파라미터를 받아옵니다. 첫번째 파라미터 API 요청을 시작하는 함수이고, 두 번째 파라미터는 deps 인데 이 deps 값은 해당 함수 안에서 사용하는 useEffect 의 deps로 설정됩니다.
* 이 값은 나중에 우리가 사용 할 비동기함수에서 파라미터가 필요하고, 그 파라미터가 바뀔 때 새로운 데이터를 불러오고 싶은 경우에 활용할 수있습니다.
* 이 값의 기본값은 [] 입니다. 즉, 컴포넌트가 가장 처음 렌더링 할 때만 API를 호출하고 싶다는 의미.
* 데이터 나중에 불러오기
  + POST, DELETE, PUT, PATCH 등의 HTTP메서드를 사용하게 된다면 필요한 시점에만 API를 호출해야 하기 때문에, 필요할 때에만 요청할 수있는 기능이 필요합니다.
  + 만약 특정 버튼을 눌렀을 때만 API를 요청하고 싶다면, 어떻게 해야할까?
  + useAsync.js
* Async function getUser({id}){}
* 프로미스를 반환하는 함수의 파라미터를 객체 형태로 해주어야합니다.
* useAsync 를 사용할 때, watch 값에 특정 값을 넣어주면 이 값이 바뀔 때마다 promiseFn에 넣은 함수를 다시 호출해줍니다.
* Context와 함께 사용하기
* 리액트의 Context와 API 연동을 함계하는 방법
* 특정 데이터들은 다양한 컴포넌트에서 필요하게 될 때도 있는데,(예: 현재 로그인된 사용자의 정보, 설정 등) 그럴때 context를 사용하면 됩니다.
* Context 준비하기

5장 리액트 라우터

* + React-router를 통한 리액트 싱글페이지 애플리케이션 만들기
  + SPA란?
    - Single page application 의 약자입니다. 말 그대로, 페이지 가 1개인 어플리케이션이란 뜻.
    - 다른 주소에 따라 다른 뷰를 보여주는 것을 라우팅이라고 합니다.
    - 리액트 자체에는 이 기능이 내장되어있지 않습니다.
  + 1. 프로젝트 준비 및 기본적인 사용법
    - Home을 위한 라우트에 exact 라는 props를 true로 설정하면됩니다.
    - Ex) <Route path=”/about” component={About}/>
    - Link : 누르면 다른 주소로 이동시키기
      * Link 컴포넌트는 클릭하면 다른 주소로 이동시키는 컴포넌트 입니다.
      * 리액트 라우터를 사용할 땐 일반 <a href=”...”>...</a> 태그를 사용하면 안됩니다.
      * Ex) <Link to=”/”>홈</link> <Link to = “/about”>소개</link>
      * Ex) <Route path=”/” exact={true} component={Home}/>
      * Ex) <Route path=”/about” component={About}/>
      * </div>
  + 2. 파라미터와 쿼리
    - 페이지 주소를 정의할 때, 유동적인 값을 전달해야 할 때도 있는데,]
    - 파라미터 : /profiles/velopert
    - 쿼리 : /about?details=true
    - -> 일반적으로 파라미터는 특정 id나 이름을 가지고 조회를 할 때 사용하고, 쿼리의 경우엔 어떤 키워드를 검색하거나, 요청을 할 때 필요한 옵션을 전달할 때 사용됩니다.
    - URL Params
      * /profile/velopert 이런식으로 뒷부분에 username을 넣어줄 떄 해당 값을 파라미터로 받아와보겠습니다.
      * Match 객체 안에는 현재의 주소 route 컴포넌트에서 정한 규칙과 어떻게 일치하는지에 대한 정보가 들어있습니다.
    - Query
      * 쿼리는 라우트 컴포넌트에게 props 전달되는 location 객체에 있는 search 값에서 읽어올 수있습니다. location 객체는 현재 앱이 갖고 있는 주소에 대한 정보를 지니고 있습니다.
      * 여기서 Search 값을 확인해야 하는데, 이 값을 객체 형태로 변환하는 건 우리가 직접 해주어야 합니다.
      * 이 작업은 qs라는 라이브러리를 사용하여 쉽게 할 수있습니다.
      * Yarn add qs;
      * Ex)

const About = ({ location }) => {

  const query = qs.parse(location.search, {

    ignoreQueryPrefix: true,

  });

  const detail = query.detail === "true"; //쿼리의 파싱결과 값은 문자열입니다.

  return (

    <div>

      <h1>소개</h1>

      {detail && <p>추가적인 정보가 어쩌고 저쩌고..</p>}

    </div>

  );

};

1. 서브라우트

* 서브라우트는, 라우트 내부의 라우트를 만드는 것을 의미합니다.
* 컴포넌트를 만들어서 그 안에 또 Route 컴포넌트를 렌더링 하면 됩니다.
* 서브라우트 만들어보기