

18. 리덕스 미들웨어를 통한 비동기 작업 관리

- 리액트 프로젝트에서 리덕스를 사용하고 있으며 비동기작업을 관리해야 한다면 `middleware`를 사용하면 효율적이고 편하게 상태관리를 할 수 있다.

18.1 작업환경 설정

- 설치 : `yarn add redux react-redux redux-actions`

modules/counter.js

```
import { createAction, handleActions } from 'redux-actions';

const INCREASE = 'counter/INCREASE';
const DECREASE = 'counter/DECREASE';

export const increase = createAction(INCREASE);
export const decrease = createAction(DECREASE);

const initialState = 0;

const counter = handleActions({
  [INCREASE]: state => state + 1,
  [DECREASE]: state => state - 1,
}, initialState)

export default counter;
```

modules/index.js

```
import counter from './counter';
import { combineReducers } from 'redux';

const rootReducer = combineReducers({
  counter
});

export default rootReducer
```

index.js

```
import React from 'react';
import ReactDOM from 'react-dom/client';
import './index.css';
import App from './App';
import rootReducer from './modules';
import { createStore } from 'redux';
import { Provider } from 'react-redux';

const store = createStore(rootReducer);

const root = ReactDOM.createRoot(document.getElementById('root'));
```

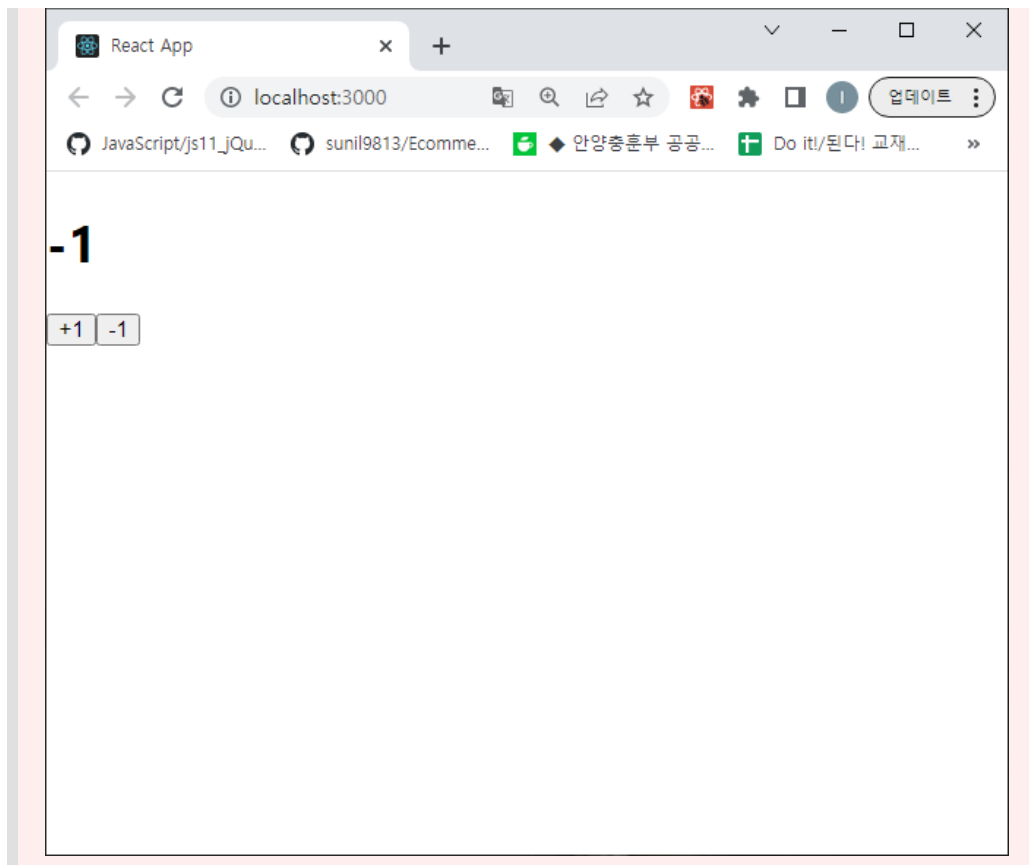
```
root.render(  
  <Provider store={store}>  
    <App />  
  </Provider>  
);
```

containers/CounterContainer.js

```
import { connect } from 'react-redux';  
import { increase, decrease } from '../modules/counter';  
import Counter from '../components/Counter';  
  
const CounterContainer = ({ number, increase, decrease }) => {  
  return (  
    <Counter number={number} onIncrease={increase} onDecrease={decrease} />  
  )  
}  
  
export default connect(  
  state => ({  
    number: state.counter  
  }),  
  {  
    increase,  
    decrease  
  }  
)(CounterContainer);
```

App.js

```
import CounterContainer from './containers/CounterContainer';  
  
function App() {  
  return (  
    <div>  
      <CounterContainer />  
    </div>  
  );  
}  
  
export default App;
```



18.2 미들웨어란?

- 리덕스 미들웨어는 액션을 디스패치 했을 때 리듀서에서 액션을 처리하기 전에 이전에 지정된 작업을 실행한다.
- 미들웨어는 액션과 리듀서사이의 중간자 라 할 수 있다.

18.2.1 미들웨어 만들기

- 액션이 디스패치될 때마다 액션정보와 액션이 디스패치되지 전후의 상태를 콘솔에 출력하는 미들웨어 작성

src/lib/loggerMiddleware.js

```
const loggerMiddleware = store => next => action => {
  console.group(action && action.type); // 액션타입으로 log를 그룹화
  console.log('이전 상태', store.getState());
  console.log('액션', action);
  next(action); // 다음 미들웨어 or 리듀서에게 전달
  console.log('다음 상태', store.getState()); // 업데이트된 상태
  console.groupEnd(); // 그룹종료
}

export default loggerMiddleware;
```

- 미들웨어는 결국 함수를 반환하는 함수를 반환하는 함수
- next(next의 파라미터는 함수형태)는 store.dispatch와 비슷한 역할을 한다.
- 큰 차이점은 next(action)을 호출하면 그다음 처리해야 할 미들웨어에게 action을 넘겨주고
- 만약에 미들웨어가 없다면 리듀서에게 액션을 전달한다.

index.js

```

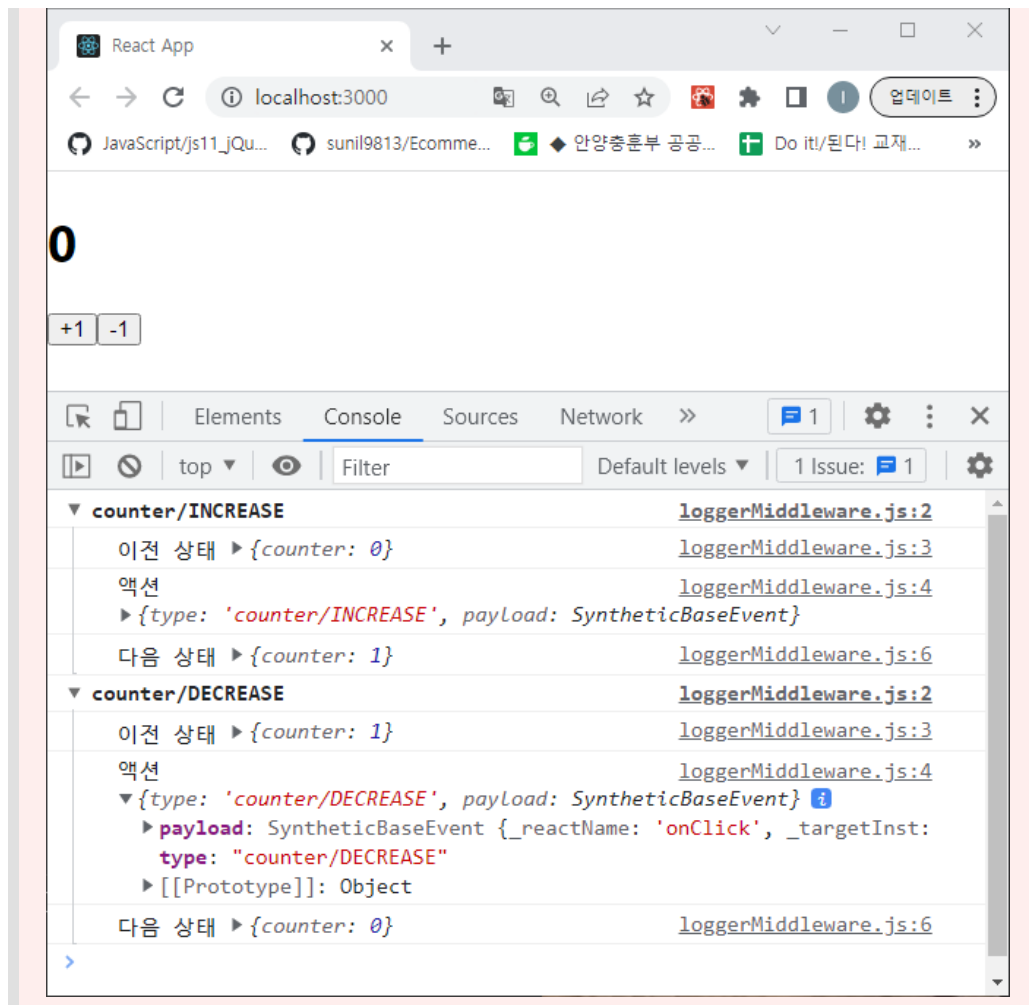
import React from 'react';
import ReactDOM from 'react-dom/client';
import './index.css';
import App from './App';
import rootReducer from './modules';
import { createStore, applyMiddleware } from 'redux';
import { Provider } from 'react-redux';
import loggerMiddleware from './lib/loggerMiddleware';

// const store = createStore(rootReducer);

// 미들웨어적용
const store = createStore(rootReducer, applyMiddleware(loggerMiddleware));

const root = ReactDOM.createRoot(document.getElementById('root'));
root.render(
  <Provider store={store}>
    <App />
  </Provider>
);

```



- 액션정보와 업데이트되기 전후 상태를 확인
- 미들웨어는 여러 종류의 작업을 할 수가 있다. 조건에 따라 액션무시, 액션변경, 리듀서에 전달해 줄 수 있다.
- 이런 미들웨어 속성을 이용해서 비동기작업을 관리하면 매우 유용 하다.

18.2.2 redux-logger사용하기

- 설치 : `yarn add redux-logger`

index.js

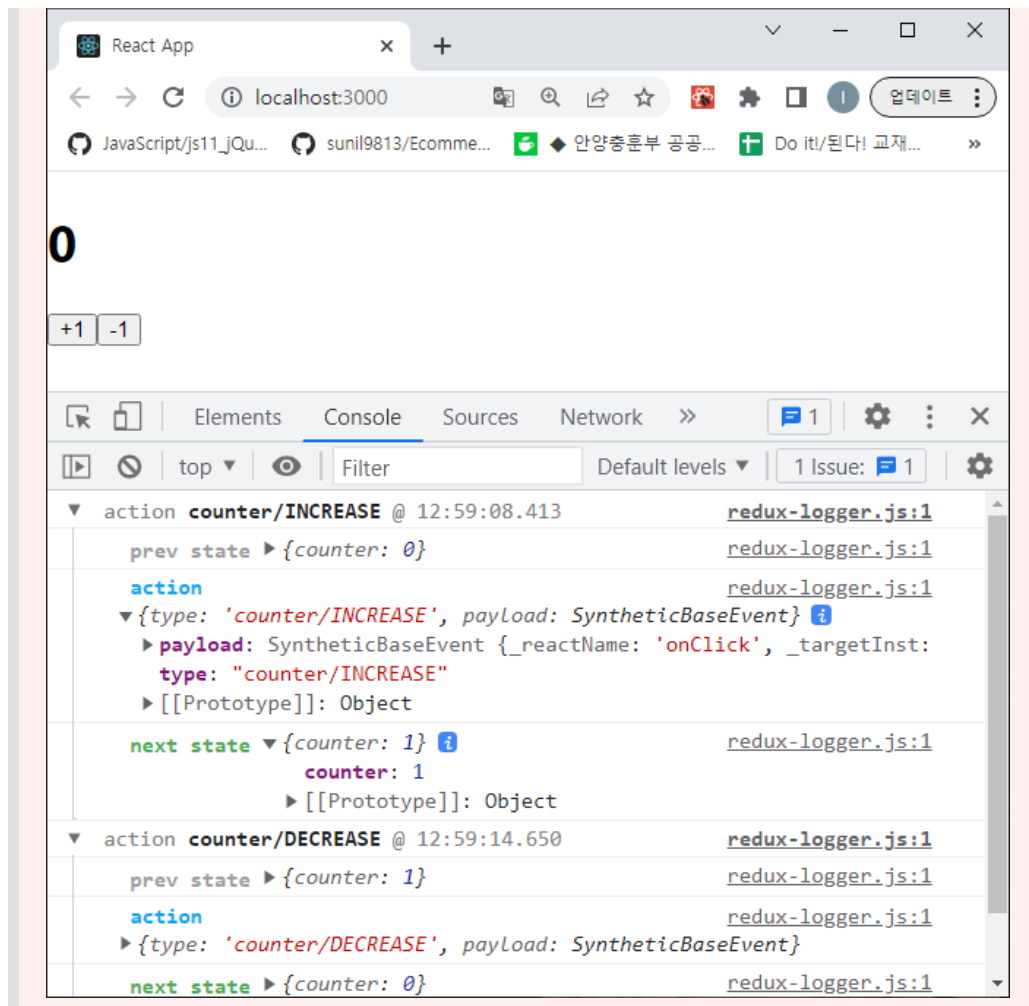
```
import React from 'react';
import ReactDOM from 'react-dom/client';
import './index.css';
import App from './App';
import rootReducer from './modules';
import { createStore, applyMiddleware } from 'redux';
import { Provider } from 'react-redux';
import loggerMiddleware from './lib/loggerMiddleware';
import { createLogger } from 'redux-logger';

// const store = createStore(rootReducer);

// 18.2.1 미들웨어적용
// const store = createStore(rootReducer, applyMiddleware(loggerMiddleware));

// 18.2.2 redux-logger적용
const logger = createLogger();
const store = createStore(rootReducer, applyMiddleware(logger));

const root = ReactDOM.createRoot(document.getElementById('root'));
root.render(
  <Provider store={store}>
    <App />
  </Provider>
);
```



18.3 비동기작업 처리 미들웨어 사용하기

- 비동기 미들웨어
 1. `redux-thunk` : 가장 많이 사용하는 비동기 미들웨어, 객체가 아닌 함수형태의 액션을 디스패치할 수 있게 한다.
 2. `redux-sega` : 다음으로 많이 사용, 특정액션이 디스패치되었을 때 정해진 로직에 따라 다른 액션을 디스패치시키는 규칙작성후 처리

18.3.1 Thunk란?

- Thunk는 특정 작업을 나중에 할 수 있도록 미루기 위해 함수 형태로 감싼 것을 의미한다.
- 설치 : `yarn add redux-thunk`

index.js

```
import React from 'react';
import ReactDOM from 'react-dom/client';
import './index.css';
import App from './App';
import rootReducer from './modules';
import { createStore, applyMiddleware } from 'redux';
import { Provider } from 'react-redux';
import loggerMiddleware from './lib/loggerMiddleware';
import { createLogger } from 'redux-logger';
import thunk from 'redux-thunk';
```

```
// const store = createStore(rootReducer);

// 18.2.1 미들웨어적용
// const store = createStore(rootReducer, applyMiddleware(loggerMiddleware));

// 18.2.2 redux-logger적용
// const logger = createLogger();
// const store = createStore(rootReducer, applyMiddleware(logger));

// 18.3.1 redux-thunk적용
const logger = createLogger();
const store = createStore(rootReducer, applyMiddleware(logger, thunk));

const root = ReactDOM.createRoot(document.getElementById('root'));
root.render(
  <Provider store={store}>
    <App />
  </Provider>
);
```

modules/counter.js

- redux-thunk는 액션생성함수에서 일반 액션객체를 반환하는 대신에 함수를 반환한다.

```
import { createAction, handleActions } from 'redux-actions';

const INCREASE = 'counter/INCREASE';
const DECREASE = 'counter/DECREASE';

export const increase = createAction(INCREASE);
export const decrease = createAction(DECREASE);

// 18.3.1 Thunk 생성함수 만들기
// 1초 뒤에 increase 혹은 decrease함수를 디스패치함
export const increaseAsync = () => dispatch => {
  setTimeout(() => {
    dispatch(increase());
  }, 1000);
}

export const decreaseAsync = () => dispatch => {
  setTimeout(() => {
    dispatch(decrease());
  }, 1000);
}

const initialState = 0;

const counter = handleActions({
  [INCREASE]: state => state + 1,
  [DECREASE]: state => state - 1,
}, initialState)

export default counter;
```

containers/CounterContainer.js

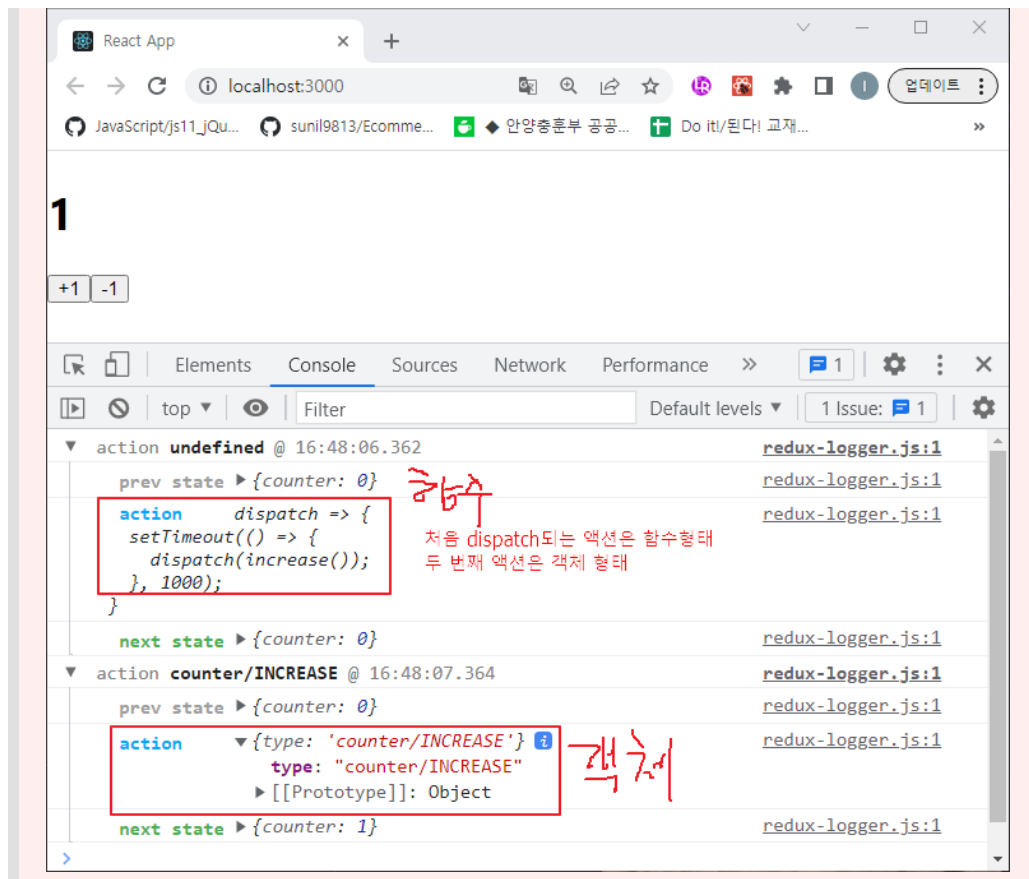
```
import { connect } from 'react-redux';
// import { increase, decrease } from '../modules/counter';
import Counter from '../components/Counter';
import { increaseAsync, decreaseAsync } from '../modules/counter';

// const CounterContainer = ({ number, increase, decrease }) => {
//   return (
//     <Counter number={number} onIncrease={increase} onDecrease={decrease} />
//   )
// }

// 18.3.1 Thunk적용
const CounterContainer = ({ number, increaseAsync, decreaseAsync }) => {
  return (
    <Counter
      number={number}
      onIncrease={increaseAsync}
      onDecrease={decreaseAsync}
    />
  )
}

export default connect(
  state => ({
    number: state.counter
  }),
  {
    // increase,
    // decrease

    // 18.3.1 Thunk적용
    increaseAsync,
    decreaseAsync
  }
)(CounterContainer);
```

18.3.2 axios

- API를 호출할 때는 주로 Promise기반 웹 클라이언트인 axios를 사용
- 설치 : yarn add axios

lib/api.js

```
import axios from 'axios'

export const getPost = id => {
  axios.get(`https://jsonplaceholder.typicode.com/posts/${id}`);
}
export const getUsers = id => {
  axios.get(`https://jsonplaceholder.typicode.com/users`);
}
```

modules/sample.js

```
import { handleActions } from "redux-actions";
import * as api from '../lib/api';

// 액션타입을 선언
// 한 요청당 세개를 만들어야 한다.

const GET_POST = 'sample/GET_POST';
const GET_POST_SUCCESS = 'sample/GET_POST_SUCCESS';
const GET_POST_FAILURE = 'sample/GET_POST_FAILURE';

const GET_USERS = 'sample/GET_USERS';
const GET_USERS_SUCCESS = 'sample/GET_USERS_SUCCESS';
const GET_USERS_FAILURE = 'sample/GET_USERS_FAILURE';
```

// thunk 함수를 생성
 // thunk 함수 내부에서는 시작, 성공, 실패했을 때 다른 액션을 디스패치한다.

```
export const getPost = id => async dispatch => {
  dispatch({ type: GET_POST }); // 요청시작
  try {
    const response = await api.getPost(id);
    dispatch({
      type: GET_POST_SUCCESS,
      payload: response.data
    }); // 요청성공
  } catch (error) {
    dispatch({
      type: GET_POST_FAILURE,
      payload: error,
      error: true
    }); // 에러발생
    throw error; // 나중에 컴퍼넌트단에서 에러를 조회할 수 있게 한다.
  }
};
```

```
export const getUsers = () => async dispatch => {
  dispatch({ type: GET_USERS }); // 요청시작
  try {
    const response = await api.getUsers();
    dispatch({
      type: GET_USERS_SUCCESS,
      payload: response.data
    }); // 요청성공
  } catch (error) {
    dispatch({
      type: GET_USERS_FAILURE,
      payload: error,
      error: true
    }); // 에러발생
    throw error; // 나중에 컴퍼넌트단에서 에러를 조회할 수 있게 한다.
  }
};
```

// 초기상태 선언
 // 요청의 로딩중에 loading이라는 객체에서 관리

```
const initialState = {
  loading: {
    GET_POST: false,
    GET_USERS: false
  },
  post: null,
  users: null
};

const sample = handleActions(
  {
    [GET_POST]: state => ({
      ...state,
      loading: {
        ...state.loading,
```

```

    GET_POST: true // 요청시작
  }
}),
[GET_POST_SUCCESS]: (state, action) => ({
  ...state,
  loading: {
    ...state.loading,
    GET_POST: false // 요청완료
  },
  post: action.payload
}),
[GET_POST_FAILURE]: (state, action) => ({
  ...state,
  loading: {
    ...state.loading,
    GET_POST: false // 요청완료
  }
}),
[GET_USERS]: state => ({
  ...state,
  loading: {
    ...state.loading,
    GET_USERS: true // 요청시작
  }
}),
[GET_USERS_SUCCESS]: (state, action) => ({
  ...state,
  loading: {
    ...state.loading,
    GET_USERS_SUCCESS: false // 요청완료
  },
  users: action.payload
}),
[GET_USERS_FAILURE]: (state, action) => ({
  ...state,
  loading: {
    ...state.loading,
    GET_USERS: false // 요청완료
  }
})
}, initialState
);

```

```
export default sample;
```

modules/index.js

```

import counter from './counter';
import { combineReducers } from 'redux';
import sample from './sample';

const rootReducer = combineReducers({
  counter,
  sample
})

export default rootReducer

```

containers/SampleContainer.js

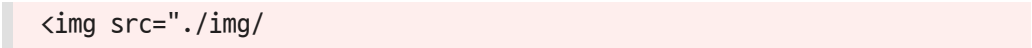
- 데이터를 불러와서 렌더링해 줄 때는 유효성검사를 해 주는 것이 중요하다.
- 데이터가 없다면 javascript는 오류를 발생하기 때문에 반드시 유효성을 검사해 주어야 한다.

```
import React from 'react';

const Sample = ({ loadingPost, loadingUsers, post, users }) => {
  return (
    <div>
      <section>
        <h1>포스트</h1>
        {loadingPost && '로딩중...'}
        {!loadingPost && post && (
          <div>
            <h3>{post.title}</h3>
            <h3>{post.body}</h3>
          </div>
        )}
      </section>
      <hr />
      <section>
        <h1>사용자 목록</h1>
        {loadingUsers && '로딩중...'}
        {!loadingUsers && users && (
          <ul>
            {users.map(user => (
              <li key={user.id}>
                {user.username} ({user.email})
              </li>
            ))}
          </ul>
        )}
      </section>
    </div>
  );
};

export default Sample;
```

src/pages/.js


 <img src='./img/'