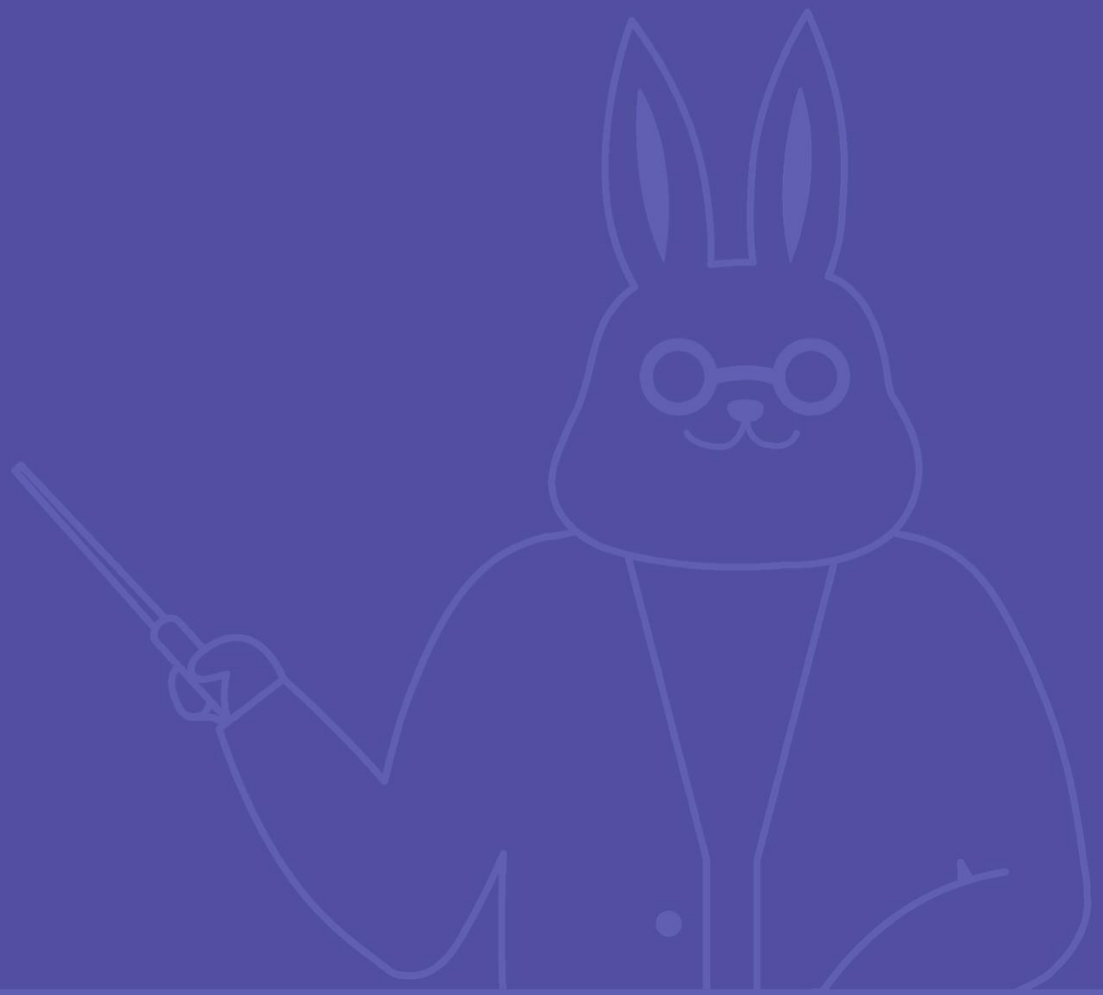
React 심화 II

00 수업 소개



/* elice */

커리큘럼

Redux를 활용한 상태 관리

redux의 배경이 되는 flux pattern을 학습하고, React 앱과 Redux를 연동합니다.

React 테스팅

React 컴포넌트를 테스트하는 방법에 대해 알아봅니다.

커리큘럼

Server Side Rendering

서버에서 페이지를 렌더링하는 SSR에 대해 학습합니다.

React 앱 빌드와 배포

Nginx 서버에서 React 앱을 빌드하고 배포합니다.

💡 추천대상

1. HTML, CSS, JS의 기본 문법과 내용을 이해하고 있는 분

HTML/CSS를 이용해 정적 페이지를 구성하고 JS를 이용해 이벤트를 달거나 DOM element를 검색해 동적 처리를 해보신 분

2. React를 이용해 간단한 UI를 구성할 수 있는 분

Virtual DOM, JSX, React Component, React hooks의 개념을 알고 간단한 UI를 구성해본 경험을 활용하고 싶은 분

3. React 관련 라이브러리를 자세히 배우고 싶은 분

axios, jest, react-router, redux, styled-components 등의 라이브러리를 들어보기는 했지만, 사용법을 자세히 익혀보고 싶은 분

🎯 수강목표

1. React 관련 도구로 원하는 기능을 구현할 수 있다.

React 관련 라이브러리를 알고, 그것들을 이용해 원하는 기능을 구현할 수 있다.

2. React 관련 기술 중 목적에 맞는 기술을 선택할 수 있다.

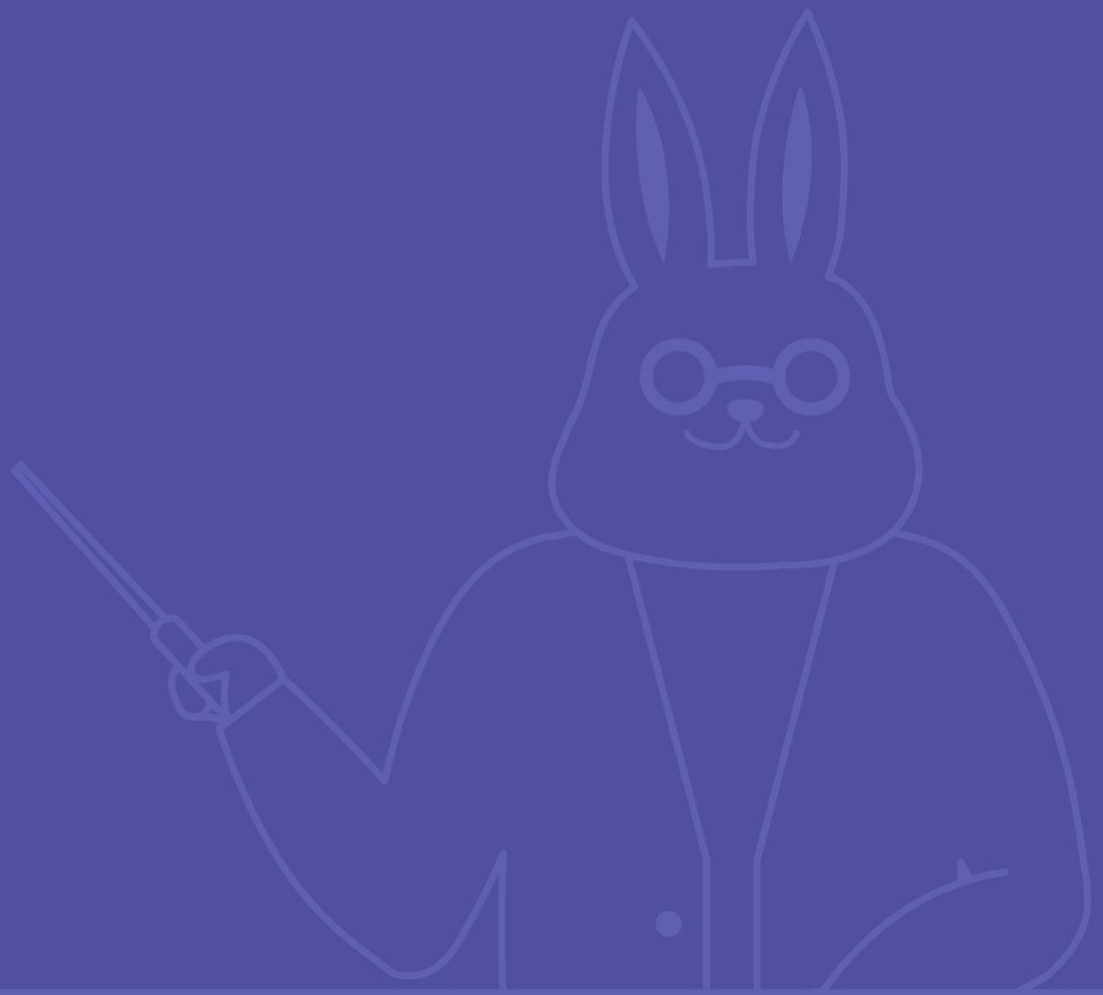
React 앱을 구성하는 여러 기술을 이해하고, 구현하고자 하는 목적에 맞는 기술을 선택할 수 있다.

3. 각 React 관련 기술이 왜 필요한지 이해한다.

React 앱을 구성하는 여러 도구가 어떤 문제를 해결하기 위한 것인지 파악할 수 있다.

React 심화 II

01 Redux를 활용한 상태 관리

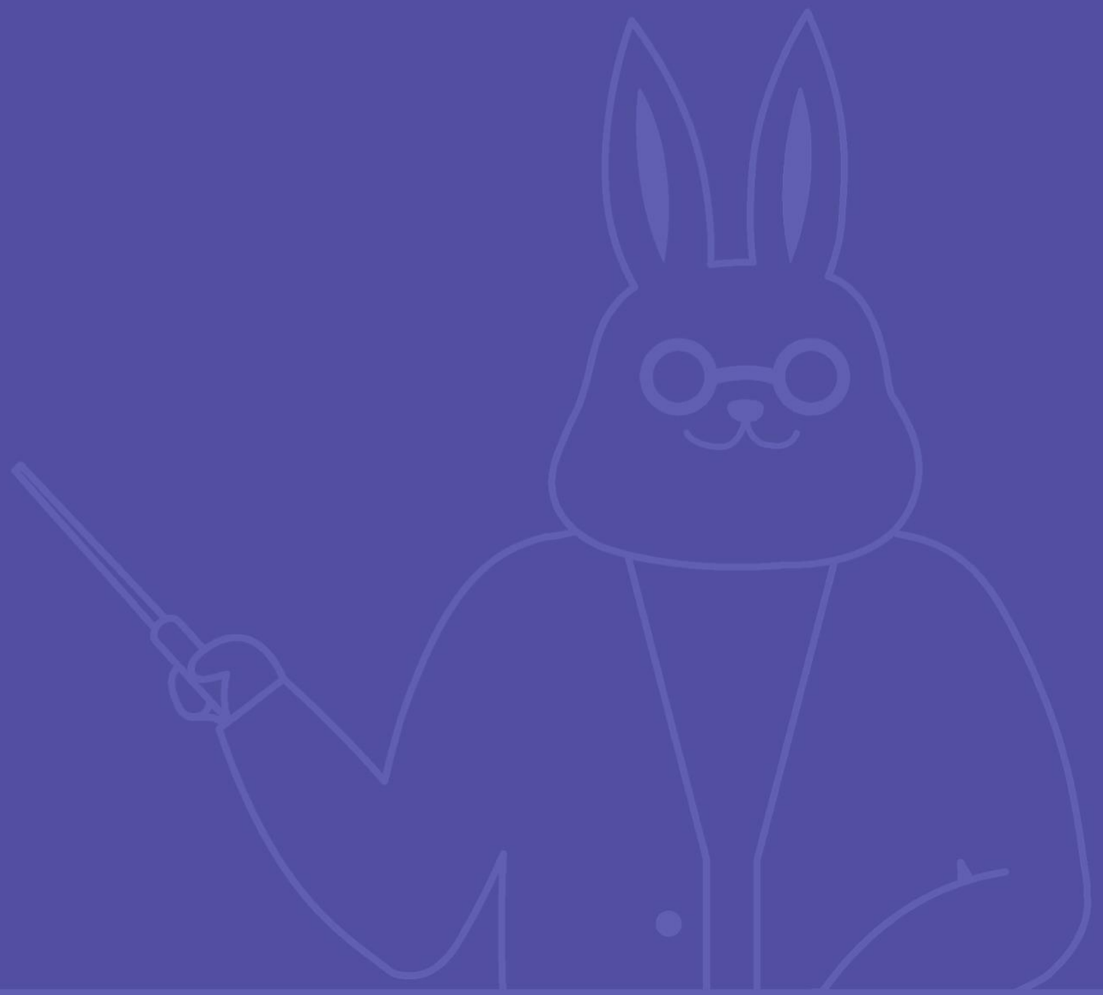


/* elice */

목차

01

Redux 소개



Redux 소개

- 앱 전체 상태를 쉽게 관리하기 위한 라이브러리.
- Redux의 많은 개념들이 Flux pattern에서 차용됨.
- 주로 React 앱과 같이 사용.
- redux.js.org에서 수많은 문서를 참고할 수 있고, 웹상에 Redux를 활용한 앱 구축 사례가 많음.

언제 Redux를 써야 하는가

- 앱 전체의 상태 관리가 필요할 때.
- 복잡한 비동기 처리가 있는 상태 관리가 필요할 때.
- 앱의 상태가 복잡하고, 이를 체계적으로 관리하고 싶을 때.
- 상태 관리 패턴을 도입하여, 여러 개발자와 협업하고 싶을 때.
- logger, devtool 등을 활용하여 상태를 관리할 필요가 있을 때.

핵심 원칙

- Single source of truth - Store는 단 하나이며, 모든 앱의 상태는 이곳에 보관됨.
- Immutability – 상태는 오로지 읽을 수만 있다. 변경하려면 모든 상태가 변경되어야 함.
- Pure function – 상태의 변경은 어떠한 사이드 이펙트도 만들지 않아야 함.

Action

Code

```
const action1 = {
  type: 'namespace/getMyData',
  payload : {
    id: 123
  }
}
```

- Action은 상태의 변경을 나타내는 개념.
- 어떤 형태든지 상관없으나, 주로 type, payload를 포함하는 JavaScript 객체.

Action Creator

Code

```
const addObj = (id) => ({
  type: 'namespace/getMyData',
  payload : {
    id: String(id).slice(1)
  }
})
```

- Action을 생성하는 함수.
- 직접 Action을 생성하는 것보다 Action Creator를 활용하면 재사용성이 좋고, 하나의 레이어를 추가할 수 있음.

Store

Code

```
const store =
createStore(reducer,
initialState)
```

- 앱 전체의 상태를 보관하는 곳.
- Action에 따라 reducer에서는 새로운 상태를 만들어내며, Store는 그 상태를 저장.
- Store의 상태는 불변하며, 매 액션이 발생할 때마다 새로운 객체가 만들어짐.

Reducer

Code

```
const reducer = (state, action) => {
  switch (action.type) {
    case 'namespace/getMyData':
      const obj = { id: action.payload.id }
      return { ...state, obj }
    default:
      return state
  }
}

const store = createStore(reducer,
initialState)
```

- Action을 받아 새로운 State를 만듦.
- (state, action) => state의 인터페이스를 따름.
- 상태 변경 시 사이드 이펙트가 없어야 함.

Dispatch

Code

```
function MyApp() {
  const dispatch = useDispatch()
  return (
    <button
      onClick={
      () => dispatch(
        addObj(1234)
      )}
    >Submit</button>
  )
}
```

- Action을 redux로 보내는 함수.
- dispatch 후에 action은 middleware를 거쳐 reducer에 도달.

Selector

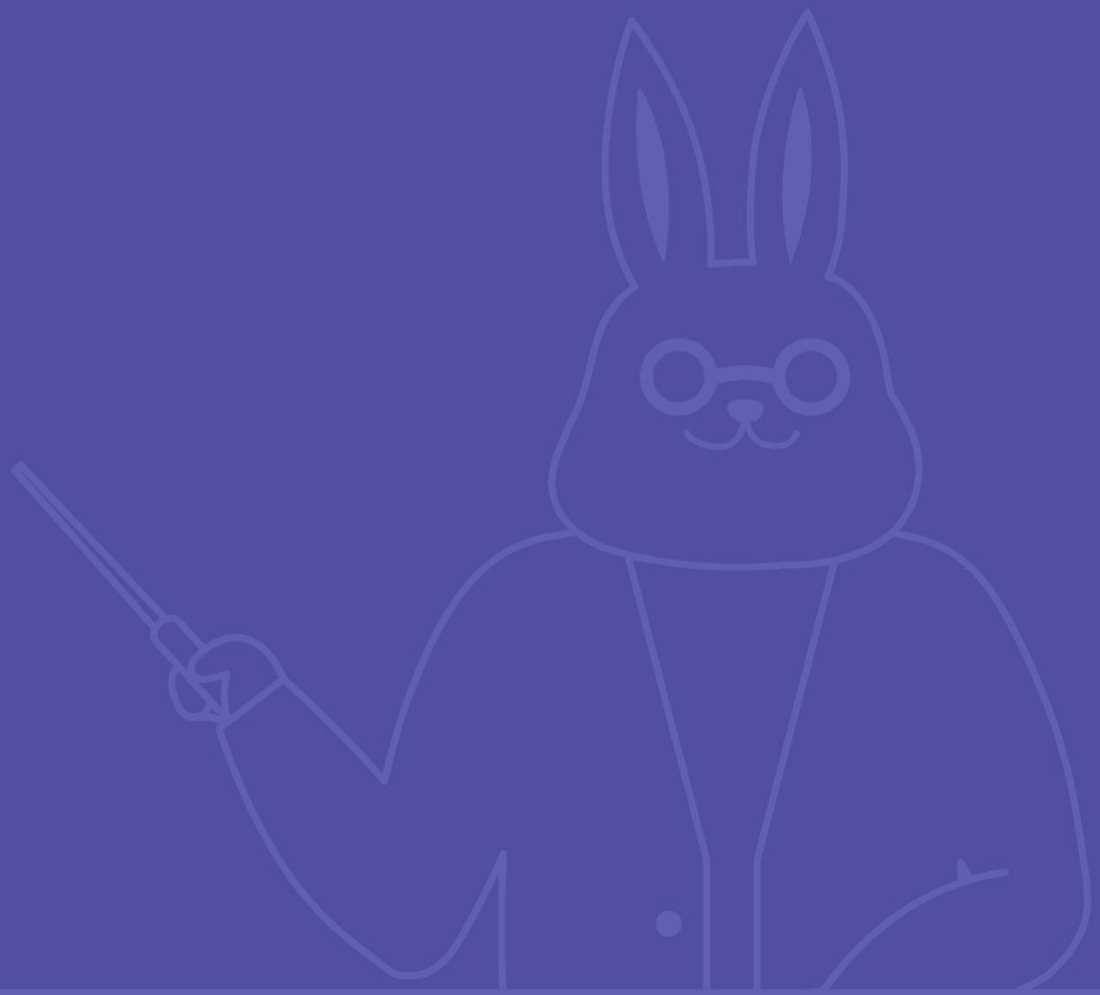
Code

```
function MyApp() {
  const obj = useSelector(state
=> state.obj)
  return (
    <div>
      {JSON.stringify(obj)}
    </div>
  )
}
```

- 특정 state 조각을 store로부터 가져오는 함수.
- store의 state는 raw data를 저장하고, 계산된 값 등을 selector로 가져오는 등의 패턴을 구사할 때 유용.

02

Redux의 구조



Redux의 구조

- redux는 자유롭게 확장하여 사용할 수 있음.
- 내부적으로 action과 데이터가 어떻게 흐르는지 이해하고,
  middleware, enhancer 등을 이용하여 redux를 확장함.

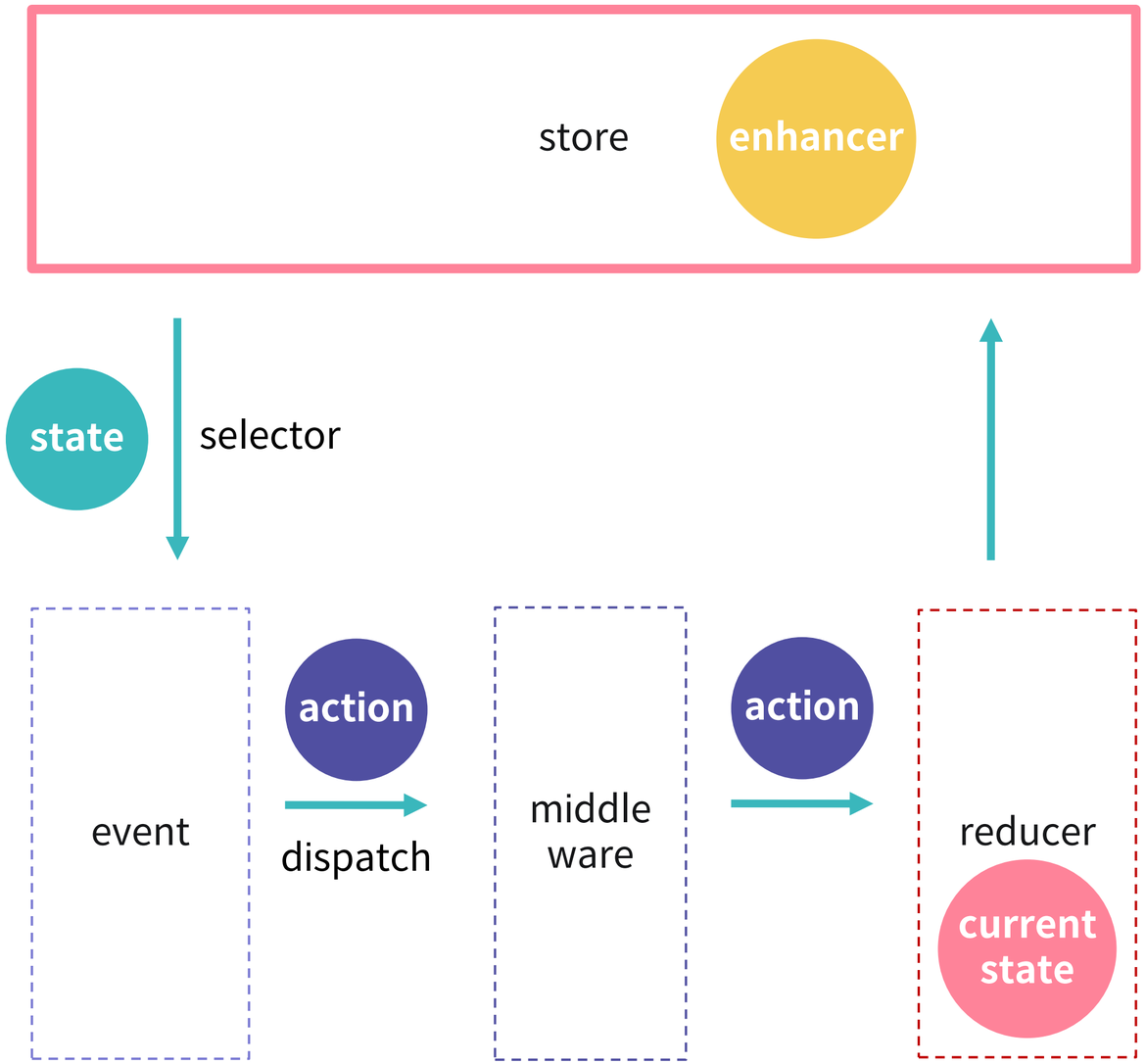
middleware

- action은 dispatch 이후 모든 middleware를 먼저 통과한 후에 reducer에 도달.
- redux-thunk, redux-logger 등의 라이브러리를 적용.

enhancer

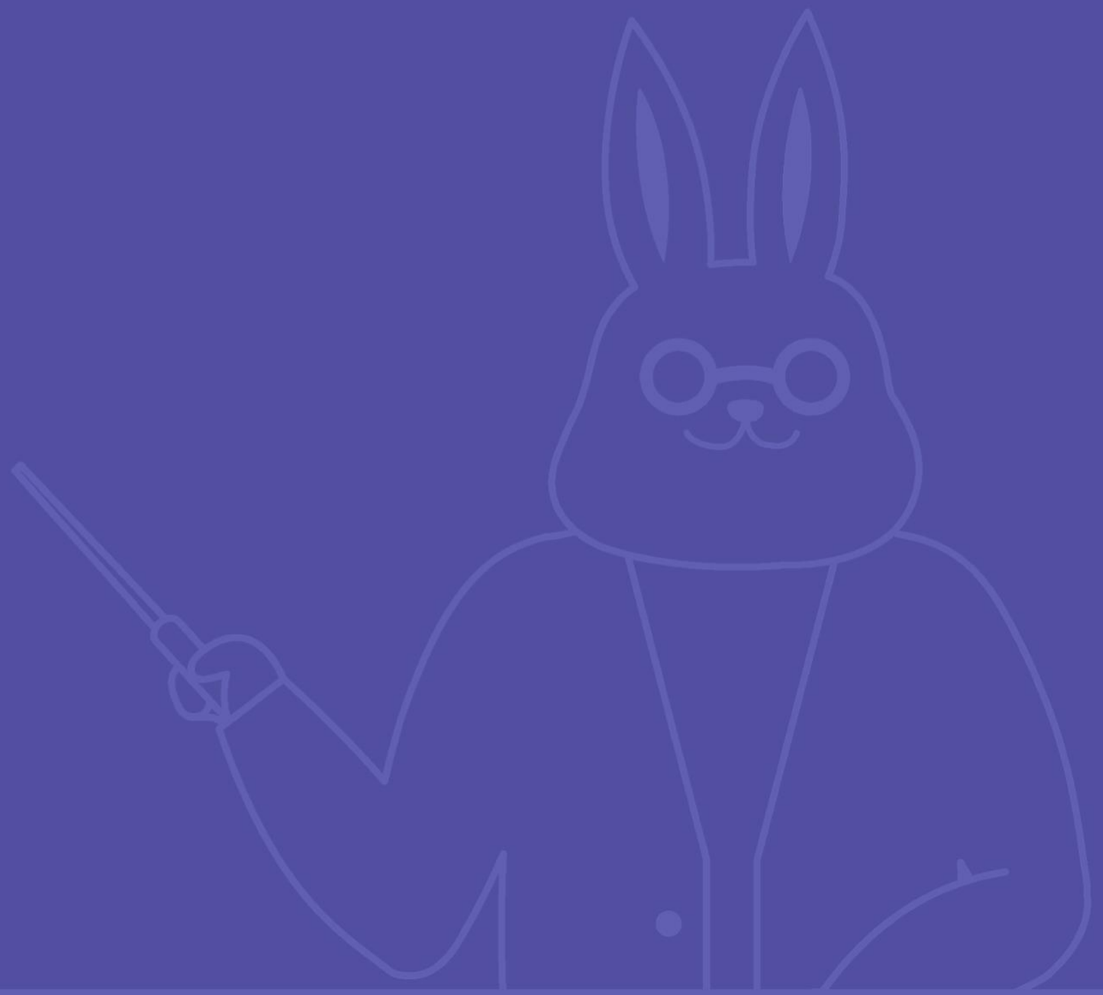
- action은 dispatch 이후 모든 middleware를 먼저 통과한 후에 reducer에 도달.
- redux devtools 등의 라이브러리를 적용.

Redux의 구조


store
enhancer
state
selector
event
action
dispatch
middle
ware
action
reducer
current
state

03

redux-toolkit 활용



redux-toolkit

- redux에서 공식적으로 추천하는, helper 라이브러리.
- 기존에 만들어야 하는 수많은 보일러 플레이트를 제거하고,
  유용한 라이브러리를 포함하여 redux 코드를 쉽게 작성하게 함.
- redux-devtools, immerjs, redux-thunk, reselect 등의 라이브러리가 미리 포함됨.

redux-toolkit API - configureStore

Code

```
const store = configureStore({
  reducer: {
    posts: postsReducer,
    users: usersReducer
  }
})
```

- redux의 createStore 함수를 래핑.
- named parameter로 쉽게 store를 생성.
- reducer - 객체를 받아, combineReducers를 적용함.

redux-toolkit API - createAction

Code

```
const addPost =
createAction('post/addPost')

addPost({ title: 'post 1' })
/*
{
  type: 'post/addPost',
  payload : { title : 'post 1' }
}
*/
```

- Action creator를 만드는 함수.
- 만들어진 action creator에 데이터를 넘기면, payload 필드로 들어감.
- 생성된 action creator는 toString() 메서드를 오버라이드해, 자신이 생성하는 액션의 타입 String을 리턴.

redux-toolkit API - createReducer

Code

```
const postsReducer =
createReducer(initState,
  builder => {
    builder.addCase(addPost,
      (state, action) => {
        state.posts
          .push(action.payload)
    })
})
```

- reducer를 만듦.
- builder의 addCase 메서드를 이용하여, action마다 state의 변경을 정의.
- immerjs를 내부적으로 사용하므로, mutable code를 이용해 간편하게 변경 코드를 작성.

redux-toolkit API - createSlice

Code

```
const postsSlice = createSlice({
  name : 'posts',
  initialState,
  reducers: {
    addPost(state, action) {
      state.posts
        .push(action.payload)
    }
  }
})

const { addPost } = postsSlice.actions
const reducer = postsSlice.reducer
```

- Slice는 Action creator, reducer 등 별도로 만들어야 하는 여러 Redux 구현체를 하나의 객체로 모은 것.
- createSlice 함수를 이용하여, 많은 보일러 플레이트를 없애고 쉽게 action creator, reducer를 만듦.

redux-toolkit API - createSelector

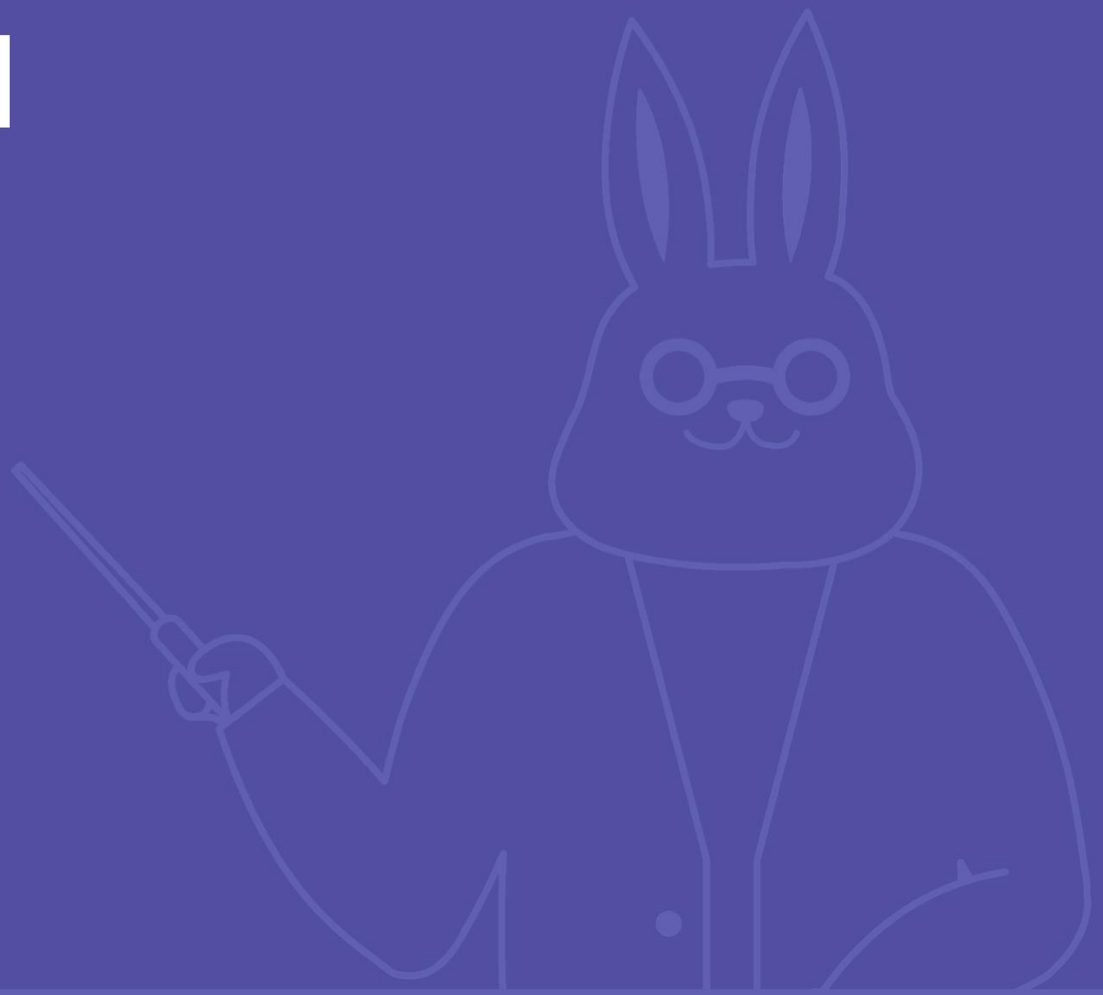
Code

```
const postsSelector = state => state.posts

const userSelector = state => state.user

const postsByUserIdSelector = createSelector(
  postsSelector,
  userSelector,
  (posts, user) =>
    posts.filter(post =>
      post.username === user.username
  )
)
```

- createSelector 함수를 이용해, state를 이용한 특정 데이터를 리턴하도록 함.
- 내부적으로 데이터를 캐시하며, 데이터가 변동이 없다면 캐시된 데이터를 리턴함.

04

Redux를 React에 연결하기



react-redux

- redux를 react 앱에 연결하게 하는 라이브러리.
- redux에서 관리하는 상태, dispatch 함수 등을 가져올 수 있음.
- 클래스 컴포넌트, 함수형 컴포넌트에 모두 연결할 수 있음.

react-redux API - Provider

Code

```
const store = configureStore({
  reducer: rootReducer
})

function App() {
  return (
    <Provider store={store}>
      <MyPage />
    </Provider>
  )
}
```

- Redux store를 React와 연결하기 위해서는 반드시 Provider로 컴포넌트를 감싸야만 함.
- Provider 안에서 렌더링된 컴포넌트들은 state에 접근할 수 있음.

react-redux API - useDispatch

Code

```
const addPost = createAction('addPost')

function MyPage() {
  const dispatch = useDispatch()

  const handleClick = () =>
    dispatch(addPost())

  return (
    <button
      onClick={handleClick}
    >Submit</button>
  )
}
```

- redux의 dispatch 함수를 가져오기 위한 API.
- dispatch로 action creator가 생성한 action을 보내면 redux 내부로 보내지게 됨.

react-redux API - useSelector

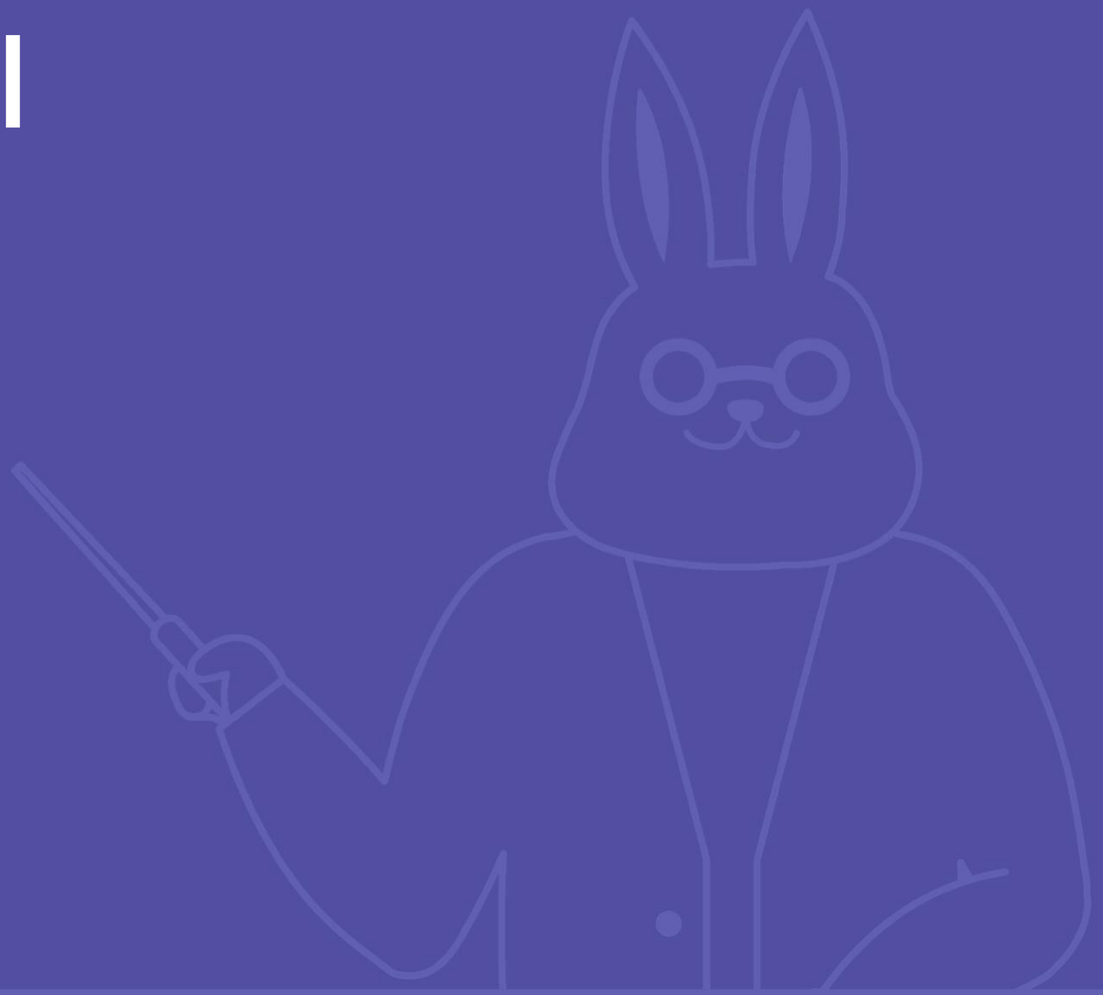
Code

```
function MyPage() {
  const posts = useSelector(state
=> state.posts)

  return posts.map(
    post => <Post {...post} />)
}
```

- Redux store로부터 데이터를 얻기 위한 API.
- selector function을 인자로 넘김.
- selector functio은 데이터에 어떤 변경을 가하면 안 됨.
- 데이터를 특정 형태로 계산하여 읽을 수 있음.

05

Redux를 이용한 비동기 처리



Redux를 이용한 비동기 처리

- redux 비동기 처리를 위해서는 비동기를 위한 middleware를 추가하여야 함.
- redux-thunk는 Promise를 이용한 비동기 Action을 쉽게 처리하도록 하는 middleware.

createAsyncThunk

- redux-toolkit에서는 thunk middleware를 디폴트로 추가.
- redux-toolkit은 createAsyncThunk API를 제공함.
  fulfilled, rejected, pending 3가지 상태에 대해 각각 reducer를 작성.
- TypeScript 환경에서 reducer 작성 시, builder callback을 사용하여 작성해야 정확한 타이핑이 가능.

createAsyncThunk

Code

```
const addPost =
createAsyncThunk('posts/addPost',
  async (title) => {
    const result = await PostAPI.addPost({
title })
    return result.data
  }
)
// Component
useEffect(() => {
  dispatch(addPost("post 1"))
}, [])
```

- createAsyncThunk는 두 인자 action type, async callback(payload creator)를 받음.
- action type을 주어지면, pending, fulfilled, rejected가 각각 postfix로 붙어 reducer로 들어옴.
  ex) posts/addPost/pending

createAsyncThunk

Code

```
const addPost =
createAsyncThunk('posts/addPost',
  async (title) => {
    const result = await
PostAPI.addPost({ title })
    return result.data
  }
)
```

- createAsyncThunk로 만들어진 action creator는 4가지 함수로 구성.
- addPost - async 함수를 dispatch하는 함수.
- addPost.pending - promise를 생성했을 때 발생하는 액션.
- addPost.fulfilled - promise가 fulfilled 되었을 때 발생하는 액션.
- addPost.rejected - promise가 rejected 되었을 때 발생하는 액션.

createAsyncThunk

Code

```
const postsSlice = createSlice({
  // ...
  extraReducers: builder => {
    builder
      .addCase(addPost.pending, state =>
...)
      .addCase(addPost.fulfilled, state =>
...)
      .addCase(addPost.rejected, state =>
...)
  }
})
```

- createSlice의 extraReducers 함수를 이용해, builder에 각 상황에 대한 리듀서를 추가.
- 공식적으로 builder pattern을 추천하는데, 타입스크립트에서 타이핑이 용이하기 때문임.

createAsyncThunk

Code

```
const postsSlice = createSlice({
  // ...
  extraReducers: builder => {
    builder
      .addCase(addPost.pending, state =>
...)
      .addCase(addPost.fulfilled, state =>
...)
      .addCase(addPost.rejected, state =>
...)
  }
})
```

- fulfilled시 데이터는 payload로 들어옴.
  ex) action.payload.todos
- rejected시 에러는 action.error로 들어오며, payload는 undefined가 됨.

연속적인 비동기 처리

Code

```
dispatch(addPost("post1"))
  .then(() =>

dispatch(updatePost("post2")))
```

- thunk 함수를 dispatch하면 promise가 리턴.
- 따라서, .then() 메서드로 연속적인 비동기 처리를 이어 실행.

동시 비동기 처리

Code

```
Promise.all([
  dispatch(addPost("post1")),
  dispatch(updatePost("post2"))
])
  .then(() =>
console.log("DONE"))
```

- Promise.all 을 이용해, 여러 비동기 처리를 동시에 실행한다.
- 주의할 점은, thunk의 promise가 rejected 되어도 .then() 으로 들어온다는 것.

크레딧

/* elice */

코스 매니저
이재성

콘텐츠 제작자
김일식

강사
김일식

감수자
-

디자이너
강혜정

연락처

TEL
070-4633-2015

WEB
https://elice.io

E-MAIL
contact@elice.io