

# Redux Thunk

Thunk **middleware** for Redux.

build passing npm v2.3.0 downloads 12M/month

```
npm install --save redux-thunk
```

## Note on 2.x Update

Most tutorials today assume Redux Thunk 1.x so you might run into an issue when running their code with 2.x.

If you use Redux Thunk 2.x in CommonJS environment, **don't forget to add `.default` to your import:**

```
- var ReduxThunk = require('redux-thunk')
+ var ReduxThunk = require('redux-thunk').default
```

리액트에서 디스패치에 FSA를 스토어로 전송하면, 리듀서가 생성되어  
페이로드 값으로 전역 상태가 변경된다.

서버와 통신해야 함.

리액트  $\xrightarrow{\text{thunk}}$  리덕스      (app) 리액트  $\xrightarrow{\text{동기}}$  복트(서버)

미들웨어 : 액션과 리듀서 사이에 중간자

미들웨어는 리액트와 리덕스 사이에 존재하는 통신과 데이터 관리하는  
소프트웨어이다.

썩크는 비동기 처리를 위한 미들웨어 함수다.

미들웨어는 리듀서에서 액션을 처리하기 전에 지정된 작업을 설정한다.

리액트에서 디스패치에 FSA를 스토어로 전송하면, 리듀서가 생성되어  
페이로드 값으로 [...] 전역상태가 변경된다.

foo is a thunk!

Redux Thunk **middleware** allows you to write action creators that return a function instead of an action. The thunk can be used to delay the dispatch of an action, or to dispatch only if a certain condition is met. The inner function receives the store methods dispatch and getState as parameters.

An action creator that returns a function to perform asynchronous dispatch:

A **thunk** is a function that wraps an expression to delay its evaluation.

```
// calculation of 1 + 2 is immediate
```

```
// x === 3
```

```
let x = 1 + 2; → 즉시 3 eager
```

```
// calculation of 1 + 2 is delayed
```

```
// foo can be called later to perform the calculation
```

```
// foo is a thunk!
```

```
let foo = () => 1 + 2;
```

↓ 호출시 3 lazy

Thunk → 지연된 실행

⇒ Thunk가 있을때 까지 기다려라 → "delay"시키는 함수: "thunk"

The term **originated** as a humorous past-tense version of "think".

npm install --save redux-thunk

Then, to enable Redux Thunk, use applyMiddleware(): → Function 호출

toolkit, thunk 설정하기.

<https://velog.io/@kyungjune/reduxtoolkit과-thunk-기본개념-연습>

\* **무엇을** : 패키지 (파일들의 집합) 을 압축해서 파일로 만드느것

**무엇을** : 기능들을 쪼개는것

**모듈성** ↑ : 파일들을 여러개로 나눌수있는것.

CreateSlice

## extraReducers#

One of the key concepts of Redux is that each slice reducer "owns" its slice of state, and that many slice reducers can independently respond to the same action type.

extraReducers allows createSlice to respond to other action types besides the types it has generated.

As case reducers specified with extraReducers are meant to reference "external" actions, they will not have actions generated in slice.actions.

As with reducers, these case reducers will also be passed to createReducer and may "mutate" their state safely.

If two fields from reducers and extraReducers happen to end up with the same action type string, the function from reducers will be used to handle that action type.

The extraReducers "builder callback" notation#

The recommended way of using extraReducers is to use a callback that receives a ActionReducerMapBuilder instance.

This builder notation is also the only way to add matcher reducers and default case reducers to your slice.

같은 액션 타입이지만 Response는 다른것

인용구 사용

↑  
overriding

```
const userSlice = createSlice({
  name: "users",
  initialState: [],
  reducers: {},
  extraReducers: (builder) => {
    builder.addCase(getUsers, (state, action) => {})
    .addMatcher(isRejectedAction, (state, action) => {})
    .addDefaultCase((state, action) => {})
  },
})
```

→ 타입스크립트-완벽하게 지원

We recommend using this API as it has better TypeScript support (and thus, IDE autocomplete even for JavaScript users), as it will correctly infer the action type in the reducer based on the provided action creator. It's particularly useful for working with actions produced by `createAction` and `createAsyncThunk`.

See the ["Builder Callback Notation" section of the `createReducer` reference](#) for details on how to use `builder.addCase`, `builder.addMatcher`, and `builder.addDefault`

## The extraReducers "map object" notation#

Like reducers, extraReducers can be an object containing Redux case reducer functions. However, the keys should be other Redux string action type constants, and `createSlice` will not auto-generate action types or action creators for reducers included in this parameter. Action creators that were generated using `createAction` may be used directly as the keys here, using computed property syntax.

### Return Value

Each function defined in the reducers argument will have a corresponding action creator generated using `createAction` and included in the result's actions field using the same function name. The generated reducer function is suitable for passing to the Redux `combineReducers` function as a "slice reducer".

You may want to consider destructuring the action creators and exporting them individually, for ease of searching for references in a larger codebase.

## createAsyncThunk#

=> 3개의 파라미터를 가지고 있으며 1개씩 리턴한다.

### Overview#

A function that accepts a Redux action type string and a callback function that should return a promise. It generates promise lifecycle action types based on the action type prefix that you pass in, and returns a thunk action creator that will run the promise callback and dispatch the lifecycle actions based on the returned promise. ↳ then() / catch() / reject()

This abstracts the standard recommended approach for handling async request lifecycles.

It does not generate any reducer functions, since it does not know what data you're fetching, how you want to track loading state, or how the data you return needs to be processed. You should write your own reducer logic that handles these actions, with whatever loading state and processing logic is appropriate for your own app.

### Parameters#

createAsyncThunk accepts three parameters: a string action type value, a payloadCreator callback, and an options object.

### type#

A string that will be used to generate additional Redux action type constants, representing the lifecycle of an async request: For example, a type argument of 'users/requestStatus' will generate these action types:

- pending: 'users/requestStatus/pending'
- fulfilled: 'users/requestStatus/fulfilled'
- rejected: 'users/requestStatus/rejected'

## payloadCreator#

A callback function that should return a promise containing the result of some asynchronous logic. It may also return a value synchronously. → Response Entity (자바)  
에러를 넣으면 2

If there is an error, it should either return a rejected promise containing an Error instance or a plain value such as a descriptive error message or otherwise a resolved promise with a `RejectWithValue` argument as returned by the `thunkAPI.rejectWithValue` function.

The `payloadCreator` function can contain whatever logic you need to calculate an appropriate result. This could include a standard AJAX data fetch request, multiple AJAX calls with the results combined into a final value, interactions with React Native `AsyncStorage`, and so on.

The `payloadCreator` function will be called with two arguments:

- arg: a single value, containing the first parameter that was passed to the `thunk` action creator when it was dispatched. This is useful for passing in values like item IDs that may be needed as part of the request. If you need to pass in multiple values, pass them together in an object when you dispatch the `thunk`, like `dispatch(fetchUsers({status: 'active', sortBy: 'name'}))`. 값 두 개 → JSON은 묶어서  
Single Value로 만들  
어서 넘겨야.
- thunkAPI: an object containing all of the parameters that are normally passed to a Redux `thunk` function, as well as additional options: ↓  
Callback은  
JSON은 그냥  
한단다.
  - `dispatch`: the Redux store `dispatch` method
  - `getState`: the Redux store `getState` method
  - `extra`: the "extra argument" given to the `thunk` middleware on setup, if available
  - `requestId`: a unique string ID value that was automatically generated to identify this request sequence
  - `signal`: an `AbortController.signal` object that may be used to see if another part of the app logic has marked this request as needing cancellation.
  - `rejectWithValue`: `rejectWithValue` is a utility function that you can return in your action creator to return a rejected response with a defined payload. It will pass whatever value you give it and return it in the payload of the rejected action.

The logic in the `payloadCreator` function may use any of these values as needed to calculate the result.

## Options#

An object with the following optional fields:

- condition: a callback that can be used to skip execution of the payload creator and all action dispatches, if desired. See [Canceling Before Execution](#) for a complete description.
- dispatchConditionRejection: if `condition()` returns false, the default behavior is that no actions will be dispatched at all. If you still want a "rejected" action to be dispatched when the thunk was canceled, set this flag to true.
- idGenerator: a function to use when generating the requestId for the request sequence. Defaults to use `nanoid`.

## Return Value#

→ FSA

`createAsyncThunk` returns a standard Redux thunk action creator. The thunk action creator function will have plain action creators for the pending, fulfilled, and rejected cases attached as nested fields. Using the `fetchUserById` example above, `createAsyncThunk` will generate four functions:

- `fetchUserById`, the thunk action creator that kicks off the async payload callback you wrote
  - `fetchUserById.pending`, an action creator that dispatches an `'users/fetchByIdStatus/pending'` action
  - `fetchUserById.fulfilled`, an action creator that dispatches an `'users/fetchByIdStatus/fulfilled'` action → Success
  - `fetchUserById.rejected`, an action creator that dispatches an `'users/fetchByIdStatus/rejected'` action → error/fail

wait

신청할 필요 X 사용가능.

Success

error/fail

→ 클라이언트

When dispatched, the thunk will:

- dispatch the pending action
- call the payloadCreator callback and wait for the returned promise to settle
- when the promise settles:
  - if the promise resolved successfully, dispatch the fulfilled action <sup>→ 클라이언트 reducer의 thunk</sup> with the promise value as `action.payload`
  - if the promise resolved with a `rejectWithValue(value)` return value, dispatch the rejected action with the value passed into `action.payload` and 'Rejected' as `action.error.message`
  - if the promise failed and was not handled with `rejectWithValue`, dispatch the rejected action with a serialized version of the error value as `action.error`

Return a fulfilled promise containing the final dispatched action (either the fulfilled or rejected action object)



아마존 RDS 인-엔진  
CORS

8080 3000

교차 출처 리소스 공유(Cross-origin resource sharing, CORS), 교차 출처 자원 공유는 웹 페이지 상의 제한된 리소스를 최초 자원이 서비스된 도메인 밖의 다른 도메인으로부터 요청할 수 있게 허용하는 구조이다.[1] 웹페이지는 교차 출처 이미지, 스타일시트, 스크립트, iframe, 동영상을 자유로이 임베드할 수 있다. [2] 특정한 도메인 간(cross-domain) 요청, 특히 Ajax 요청은 동일-출처 보안 정책에 의해 기본적으로 금지된다.

CORS는 교차 출처 요청을 허용하는 것이 안전한지 아닌지를 판별하기 위해 브라우저와 서버가 상호 통신하는 하나의 방법을 정의한다.[3] 순수하게 동일한 출처 요청보다 더 많은 자유와 기능을 허용하지만 단순히 모든 교차 출처 요청을 허용하는 것보다 더 안전하다. CORS의 사양은 원래 W3C 권고안으로 출판되었으나[4] 해당 문서는 구식(obsolete)인 상태이다.[5] 현재 CORS를 정의하면서 활발히 유지보수된 사양은 WHATWG의 Fetch Living Standard이다.[6]