Redux Toolkit

Version - **1.9.5**

https://redux-toolkit.js.org

Introduction

Install Redux Toolkit and React-Redux

Add the Redux Toolkit and React-Redux packages to your project:

```
npm install @reduxjs/toolkit react-redux
```

Create a Redux Store

Create a file named `src/app/store.js`. Import the `configureStore` API from Redux Toolkit. We'll start by creating an empty Redux store, and exporting it:

```
import { configureStore } from '@reduxjs/toolkit'

export const store = configureStore({
  reducer: {},
})

// Infer the `RootState` and `AppDispatch` types from the store itself
export type RootState = ReturnType<typeof store.getState>
// Inferred type: {posts: PostsState, comments: CommentsState, users: UsersState}
export type AppDispatch = typeof store.dispatch
```

This creates a Redux store, and also automatically configure the Redux DevTools extension so that you can inspect the store while developing.

Provide the Redux Store to React

Once the store is created, we can make it available to our React components by putting a React-Redux `<Provider>` around our application in `src/index.js`. Import the Redux store we just created, put a `<Provider>` around your `<App>`, and pass the store as a prop:

```
import React from 'react'
import ReactDOM from 'react-dom'
import './index.css'
import App from './App'
import { store } from './app/store'
import { Provider } from 'react-redux'



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

Create a Redux State Slice

Add a new file named `src/features/counter/counterSlice.js`. In that file, import the `createSlice` API from Redux Toolkit.

Creating a slice requires a string name to identify the slice, an initial state value, and one or more reducer functions to define how the state can be updated. Once a slice is created, we can export the generated Redux action creators and the reducer function for the whole slice.

Redux requires that we write all state updates immutably, by making copies of data and updating the copies. However, Redux Toolkit's `createSlice` and `createReducer` APIs use Immer inside to allow us to write "mutating" update logic that becomes correct immutable updates.

```
import { createSlice } from '@reduxjs/toolkit'
import type { PayloadAction } from '@reduxjs/toolkit'

export interface CounterState {
  value: number
}

const initialState: CounterState = {
  value: 0,
}

export const counterSlice = createSlice({
  name: 'counter',
  initialState,
  reducers: {
    increment: (state) => {
      // Redux Toolkit allows us to write "mutating" logic in reducers. It
      // doesn't actually mutate the state because it uses the Immer library,
      // which detects changes to a "draft state" and produces a brand new
      // immutable state based off those changes
      state.value += 1
    },
    decrement: (state) => {
      state.value -= 1
    },
    incrementByAmount: (state, action: PayloadAction<number>) => {
      state.value += action.payload
    },
  },
})

// Action creators are generated for each case reducer function
export const { increment, decrement, incrementByAmount } = counterSlice.actions

export default counterSlice.reducer
```

Add Slice Reducers to the Store

Next, we need to import the reducer function from the counter slice and add it to our store. By defining a field inside the `reducer` parameter, we tell the store to use this slice reducer function to handle all updates to that state.

```
import { configureStore } from '@reduxjs/toolkit'
import counterReducer from '../features/counter/counterSlice'

export const store = configureStore({
  reducer: {
    counter: counterReducer,
  },
})

// Infer the `RootState` and `AppDispatch` types from the store itself
export type RootState = ReturnType<typeof store.getState>
// Inferred type: {posts: PostsState, comments: CommentsState, users: UsersState}
export type AppDispatch = typeof store.dispatch
```

Use Redux State and Actions in React Components

Now we can use the React-Redux hooks to let React components interact with the Redux store. We can read data from the store with `useSelector`, and dispatch actions using `useDispatch`. Create a `src/features/counter/Counter.js` file with a `<Counter>` component inside, then import that component into `App.js` and render it inside of `<App>`.

```
import React from 'react'
import type { RootState } from '../../app/store'
import { useSelector, useDispatch } from 'react-redux'
import { decrement, increment } from './counterSlice'

export function Counter() {
  const count = useSelector((state: RootState) => state.counter.value)
  const dispatch = useDispatch()

  return (
    <div>
      <div>
        <button
          aria-label="Increment value"
          onClick={() => dispatch(increment())}
        >
          Increment
        </button>
        <span>{count}</span>
        <button
          aria-label="Decrement value"
          onClick={() => dispatch(decrement())}
        >
          Decrement
        </button>
      </div>
    </div>  )}
```

Now, any time you click the "Increment" and "Decrement" buttons:

- The corresponding Redux action will be dispatched to the store
- The counter slice reducer will see the actions and update its state
- The `<Counter>` component will see the new state value from the store and re-render itself with the new data

RTK Query Quick Start

Introduction

RTK Query is an advanced data fetching and caching tool, designed to simplify common cases for loading data in a web application. RTK Query itself is built on top of the Redux Toolkit core, and leverages RTK's APIs like `createSlice` and `createAsyncThunk` to implement its capabilities.

RTK Query is included in the `@reduxjs/toolkit` package as an additional addon. You are not required to use the RTK Query APIs when you use Redux Toolkit, but we think many users will benefit from RTK Query's data fetching and caching in their apps.

How to Read This Tutorial

For this tutorial, we assume that you're using Redux Toolkit with React, but you can also use it with other UI layers as well. The examples are based on [a typical Create-React-App folder structure]() where all the application code is in a `src`, but the patterns can be adapted to whatever project or folder setup you're using.

Setting up your store and API service

To see how RTK Query works, let's walk through a basic usage example. For this example, we'll assume you're using React and want to make use of RTK Query's auto-generated React hooks.

Create an API service

First, we'll create a service definition that queries the publicly available [PokeAPI]().

```
// Need to use the React-specific entry point to import createApi
import { createApi, fetchBaseQuery } from '@reduxjs/toolkit/query/react'
import type { Pokemon } from './types'

// Define a service using a base URL and expected endpoints
export const pokemonApi = createApi({
  reducerPath: 'pokemonApi',
  baseQuery: fetchBaseQuery({ baseUrl: 'https://pokeapi.co/api/v2/' }),
  endpoints: (builder) => ({
    getPokemonByName: builder.query<Pokemon, string>({
      query: (name) => `pokemon/${name}`,
    }),
  }),
})

// Export hooks for usage in functional components, which are
// auto-generated based on the defined endpoints
export const { useGetPokemonByNameQuery } = pokemonApi
```

With RTK Query, you usually define your entire API definition in one place. This is most likely different from what you see with other libraries such as `swr` or `react-query`, and there are several reasons for that. Our perspective is that it's *much* easier to keep track of how requests, cache invalidation, and general app configuration behave when they're all in one central location in comparison to having X number of custom hooks in different files throughout your application.

Add the service to your store

An RTKQ service generates a "slice reducer" that should be included in the Redux root reducer, and a custom middleware that handles the data fetching. Both need to be added to the Redux store.

```
import { configureStore } from '@reduxjs/toolkit'
// Or from '@reduxjs/toolkit/query/react'
import { setupListeners } from '@reduxjs/toolkit/query'
import { pokemonApi } from './services/pokemon'

export const store = configureStore({
  reducer: {
    // Add the generated reducer as a specific top-level slice
    [pokemonApi.reducerPath]: pokemonApi.reducer,
  },
  // Adding the api middleware enables caching, invalidation, polling,
  // and other useful features of `rtk-query`.
  middleware: (getDefaultMiddleware) =>
    getDefaultMiddleware().concat(pokemonApi.middleware),
})

// optional, but required for refetchOnFocus/refetchOnReconnect behaviors
// see `setupListeners` docs - takes an optional callback as the 2nd arg for customization
setupListeners(store.dispatch)
```

Wrap your application with the `Provider`

If you haven't already done so, follow the standard pattern for providing the Redux store to the rest of your React application component tree:

```
import * as React from 'react'
import { render } from 'react-dom'
import { Provider } from 'react-redux'

import App from './App'
import { store } from './app/store'

const rootElement = document.getElementById('root')
render(
  <Provider store={store}>
    <App />
  </Provider>,
  rootElement
)
```

Use the query in a component

Once a service has been defined, you can import the hooks to make a request.

```
import * as React from 'react'
import { useGetPokemonByNameQuery } from './services/pokemon'

export default function App() {
  // Using a query hook automatically fetches data and returns query values
  const { data, error, isLoading } = useGetPokemonByNameQuery('bulbasaur')
  // Individual hooks are also accessible under the generated endpoints:
  // const { data, error, isLoading } = pokemonApi.endpoints.getPokemonByName.useQuery('bulbasaur')

  return (
    <div className="App">
      {error ? (
        <>Oh no, there was an error</>
      ) : isLoading ? (
        <>Loading...</>
      ) : data ? (
        <>
          <h3>{data.species.name}</h3>
          <img src={data.sprites.front_shiny} alt={data.species.name} />
        </>
      ) : null}
    </div>
  )
}
```

When making a request, you're able to track the state in several ways. You can always check `data`, `status`, and `error` to determine the right UI to render. In addition, `useQuery` also provides utility booleans like `isLoading`, `isFetching`, `isSuccess`, and `isError` for the latest request.

Redux Toolkit TypeScript Quick Start

Introduction

This tutorial will briefly show how to use TypeScript with Redux Toolkit.

Project Setup

Define Root State and Dispatch Types

Using configureStore should not need any additional typings. You will, however, want to extract the `RootState` type and the `Dispatch` type so that they can be referenced as needed. Inferring these types from the store itself means that they correctly update as you add more state slices or modify middleware settings.

Since those are types, it's safe to export them directly from your store setup file such as `app/store.ts` and import them directly into other files.

```
import { configureStore } from '@reduxjs/toolkit'
// ...

export const store = configureStore({
  reducer: {
    posts: postsReducer,
    comments: commentsReducer,
    users: usersReducer,
  },
})

// Infer the `RootState` and `AppDispatch` types from the store itself
export type RootState = ReturnType<typeof store.getState>
// Inferred type: {posts: PostsState, comments: CommentsState, users: UsersState}
export type AppDispatch = typeof store.dispatch
```

Define Typed Hooks

While it's possible to import the `RootState` and `AppDispatch` types into each component, it's **better to create typed versions of the `useDispatch` and `useSelector` hooks for usage in your application**. This is important for a couple reasons:

- For `useSelector`, it saves you the need to type `(state: RootState)` every time
- For `useDispatch`, the default `Dispatch` type does not know about thunks. In order to correctly dispatch thunks, you need to use the specific customized `AppDispatch` type from the store that includes the thunk middleware types, and use that with `useDispatch`. Adding a pre-typed `useDispatch` hook keeps you from forgetting to import `AppDispatch` where it's needed.

Since these are actual variables, not types, it's important to define them in a separate file such as `app/hooks.ts`, not the store setup file. This allows you to import them into any component file that needs to use the hooks, and avoids potential circular import dependency issues.

```
import { useDispatch, useSelector } from 'react-redux'
import type { TypedUseSelectorHook } from 'react-redux'
import type { RootState, AppDispatch } from './store'

// Use throughout your app instead of plain `useDispatch` and `useSelector`
export const useAppDispatch: () => AppDispatch = useDispatch
export const useAppSelector: TypedUseSelectorHook<RootState> = useSelector
```

Application Usage

Define Slice State and Action Types

Each slice file should define a type for its initial state value, so that `createSlice` can correctly infer the type of `state` in each case reducer.

All generated actions should be defined using the `PayloadAction<T>` type from Redux Toolkit, which takes the type of the `action.payload` field as its generic argument.

You can safely import the `RootState` type from the store file here. It's a circular import, but the TypeScript compiler can correctly handle that for types. This may be needed for use cases like writing selector functions.

```
import { createSlice } from '@reduxjs/toolkit'
import type { PayloadAction } from '@reduxjs/toolkit'
import type { RootState } from '../../app/store'




// Define a type for the slice state
interface CounterState {
  value: number
}

// Define the initial state using that type
const initialState: CounterState = {
  value: 0,
}

export const counterSlice = createSlice({
  name: 'counter',
  // `createSlice` will infer the state type from the `initialState` argument
  initialState,
  reducers: {
    increment: (state) => {
      state.value += 1
    },
    decrement: (state) => {
      state.value -= 1
    },
    // Use the PayloadAction type to declare the contents of `action.payload`
    incrementByAmount: (state, action: PayloadAction<number>) => {
      state.value += action.payload
    },
  },
})

export const { increment, decrement, incrementByAmount } = counterSlice.actions

// Other code such as selectors can use the imported `RootState` type
export const selectCount = (state: RootState) => state.counter.value

export default counterSlice.reducer
```

The generated action creators will be correctly typed to accept a `payload` argument based on the `PayloadAction<T>` type you provided for the reducer. For example, `incrementByAmount` requires a `number` as its argument.

In some cases, TypeScript may unnecessarily tighten the type of the initial state. If that happens, you can work around it by casting the initial state using `as`, instead of declaring the type of the variable:

```
// Workaround: cast state instead of declaring variable type
const initialState = {
  value: 0,
} as CounterState
```

Use Typed Hooks in Components

In component files, import the pre-typed hooks instead of the standard hooks from React-Redux.

```
import React, { useState } from 'react'

import { useAppSelector, useAppDispatch } from 'app/hooks'

import { decrement, increment } from './counterSlice'

export function Counter() {
  // The `state` arg is correctly typed as `RootState` already
  const count = useAppSelector((state) => state.counter.value)
  const dispatch = useAppDispatch()

  // omit rendering logic
}
```

Usage Guide

The Redux core library is deliberately unopinionated. It lets you decide how you want to handle everything, like store setup, what your state contains, and how you want to build your reducers.

This is good in some cases, because it gives you flexibility, but that flexibility isn't always needed. Sometimes we just want the simplest possible way to get started, with some good default behavior out of the box. Or, maybe you're writing a larger application and finding yourself writing some similar code, and you'd like to cut down on how much of that code you have to write by hand.

As described in the Quick Start page, the goal of Redux Toolkit is to help simplify common Redux use cases. It is not intended to be a complete solution for everything you might want to do with Redux, but it should make a lot of the

Redux-related code you need to write a lot simpler (or in some cases, eliminate some of the hand-written code entirely).

Redux Toolkit exports several individual functions that you can use in your application, and adds dependencies on some other packages that are commonly used with Redux (like Reselect and Redux-Thunk). This lets you decide how to use these in your own application, whether it be a brand new project or updating a large existing app.

Let's look at some of the ways that Redux Toolkit can help make your Redux-related code better.

Store Setup

Every Redux app needs to configure and create a Redux store. This usually involves several steps:

- Importing or creating the root reducer function
- Setting up middleware, likely including at least one middleware to handle asynchronous logic
- Configuring the Redux DevTools Extension
- Possibly altering some of the logic based on whether the application is being built for development or production

Manual Store Setup

The following example from the Configuring Your Store page in the Redux docs shows a typical store setup process:

```
import { applyMiddleware, createStore } from 'redux'
import { composeWithDevTools } from 'redux-devtools-extension'
import thunkMiddleware from 'redux-thunk'

import monitorReducersEnhancer from './enhancers/monitorReducers'
import loggerMiddleware from './middleware/logger'
import rootReducer from './reducers'

export default function configureStore(preloadedState) {
  const middlewares = [loggerMiddleware, thunkMiddleware]
  const middlewareEnhancer = applyMiddleware(...middlewares)
  const enhancers = [middlewareEnhancer, monitorReducersEnhancer]
  const composedEnhancers = composeWithDevTools(...enhancers)

  const store = createStore(rootReducer, preloadedState, composedEnhancers)

  if (process.env.NODE_ENV !== 'production' && module.hot) {
    module.hot.accept('./reducers', () => store.replaceReducer(rootReducer))
  }

  return store
}
```

This example is readable, but the process isn't always straightforward:

- The basic Redux `createStore` function takes positional arguments: `(rootReducer, preloadedState, enhancer)`. Sometimes it's easy to forget which parameter is which.
- The process of setting up middleware and enhancers can be confusing, especially if you're trying to add several pieces of configuration.
- The Redux DevTools Extension docs initially suggest using some hand-written code that checks the global namespace to see if the extension is available. Many users copy and paste those snippets, which make the setup code harder to read.

Simplifying Store Setup with `configureStore`

`configureStore` helps with those issues by:

- Having an options object with "named" parameters, which can be easier to read
- Letting you provide arrays of middleware and enhancers you want to add to the store, and calling `applyMiddleware` and `compose` for you automatically
- Enabling the Redux DevTools Extension automatically

In addition, `configureStore` adds some middleware by default, each with a specific goal:

- `redux-thunk` is the most commonly used middleware for working with both synchronous and async logic outside of components
- In development, middleware that check for common mistakes like mutating the state or using non-serializable values.

This means the store setup code itself is a bit shorter and easier to read, and also that you get good default behavior out of the box.

The simplest way to use it is to just pass the root reducer function as a parameter named `reducer`

```
import { configureStore } from '@reduxjs/toolkit'
import rootReducer from './reducers'

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

export default store
```

You can also pass an object full of "slice reducers", and `configureStore` will call `combineReducers` for you:

```
import { configureStore } from '@reduxjs/toolkit'
import usersReducer from './usersReducer'
import postsReducer from './postsReducer'

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

export default store
```

Note that this only works for one level of reducers. If you want to nest reducers, you'll need to call `combineReducers` yourself to handle the nesting.

If you need to customize the store setup, you can pass additional options. Here's what the hot reloading example might look like using Redux Toolkit:

```
import { configureStore } from '@reduxjs/toolkit'

import monitorReducersEnhancer from './enhancers/monitorReducers'
import loggerMiddleware from './middleware/logger'
import rootReducer from './reducers'

export default function configureAppStore(preloadedState) {
  const store = configureStore({
    reducer: rootReducer,
    middleware: (getDefaultMiddleware) =>
      getDefaultMiddleware().concat(loggerMiddleware),
    preloadedState,
    enhancers: [monitorReducersEnhancer],
  })
  if (process.env.NODE_ENV !== 'production' && module.hot) {
    module.hot.accept('./reducers', () => store.replaceReducer(rootReducer))
  }

  return store
}
```

If you provide the `middleware` argument, `configureStore` will only use whatever middleware you've listed. If you want to have some custom middleware *and* the defaults all together, you can use the callback notation, call `getDefaultMiddleware` and include the results in the `middleware` array you return.

Writing Reducers

Reducers are the most important Redux concept. A typical reducer function needs to:

- Look at the `type` field of the action object to see how it should respond
- Update its state immutably, by making copies of the parts of the state that need to change and only modifying those copies

While you can use any conditional logic you want in a reducer, the most common approach is a `switch` statement, because it's a straightforward way to handle multiple possible values for a single field. However, many people don't like switch statements. The Redux docs show an example of writing a function that acts as a lookup table based on action types, but leave it up to users to customize that function themselves.

The other common pain points around writing reducers have to do with updating state immutably. JavaScript is a mutable language, updating nested immutable data by hand is hard, and it's easy to make mistakes.

Simplifying Reducers with `createReducer`

Since the "lookup table" approach is popular, Redux Toolkit includes a `createReducer` function similar to the one shown in the Redux docs. However, our `createReducer` utility has some special "magic" that makes it even better. It uses the Immer library internally, which lets you write code that "mutates" some data, but actually applies the updates immutably. This makes it effectively impossible to accidentally mutate state in a reducer.

In general, any Redux reducer that uses a `switch` statement can be converted to use `createReducer` directly. Each `case` in the switch becomes a key in the object passed to `createReducer`. Immutable update logic, like spreading objects or copying arrays, can probably be converted to direct "mutation". It's also fine to keep the immutable updates as-is and return the updated copies, too.

Here's some examples of how you can use `createReducer`. We'll start with a typical "todo list" reducer that uses switch statements and immutable updates:

```
function todosReducer(state = [], action) {
  switch (action.type) {
    case 'ADD_TODO': {
      return state.concat(action.payload)
    }
    case 'TOGGLE_TODO': {
      const { index } = action.payload
      return state.map((todo, i) => {
        if (i !== index) return todo

        return {
          ...todo,
          completed: !todo.completed,
        }
      })
    }
    case 'REMOVE_TODO': {
      return state.filter((todo, i) => i !== action.payload.index)
    }
    default:
      return state
  }
}
```

Notice that we specifically call `state.concat()` to return a copied array with the new todo entry, `state.map()` to return a copied array for the toggle case, and use the object spread operator to make a copy of the todo that needs to be updated.

With `createReducer`, we can shorten that example considerably:

```
const todosReducer = createReducer([], (builder) => {
  builder
    .addCase('ADD_TODO', (state, action) => {
      // "mutate" the array by calling push()
      state.push(action.payload)
    })
    .addCase('TOGGLE_TODO', (state, action) => {
      const todo = state[action.payload.index]
      // "mutate" the object by overwriting a field
      todo.completed = !todo.completed
    })
    .addCase('REMOVE_TODO', (state, action) => {
      // Can still return an immutably-updated value if we want to
      return state.filter((todo, i) => i !== action.payload.index)
    })
})
```

The ability to "mutate" the state is especially helpful when trying to update deeply nested state. This complex and painful code:

```
case "UPDATE_VALUE":
  return {
    ...state,
    first: {
      ...state.first,
      second: {
        ...state.first.second,
        [action.someId]: {
          ...state.first.second[action.someId],
          fourth: action.someValue
        }
       } }}
```

Can be simplified down to just:

```
updateValue(state, action) {
   const {someId, someValue} = action.payload;
   state.first.second[someId].fourth = someValue;
}
```

Considerations for Using `createReducer`

While the Redux Toolkit `createReducer` function can be really helpful, keep in mind that:

- The "mutative" code only works correctly inside of our `createReducer` function
- Immer won't let you mix "mutating" the draft state and also returning a new state value

See the `createReducer` API reference for more details.

Writing Action Creators

Redux encourages you to write "action creator" functions that encapsulate the process of creating an action object. While this is not strictly required, it's a standard part of Redux usage.

Most action creators are very simple. They take some parameters, and return an action object with a specific `type` field and the parameters inside the action. These parameters are typically put in a field called `payload`, which is part of the Flux Standard Action convention for organizing the contents of action objects. A typical action creator might look like:

```
function addTodo(text) {
  return {
   type: 'ADD_TODO',
   payload: { text },
  }
}
```

Defining Action Creators with `createAction`

Writing action creators by hand can get tedious. Redux Toolkit provides a function called `createAction`, which simply generates an action creator that uses the given action type, and turns its argument into the `payload` field:

```
const addTodo = createAction('ADD_TODO')

addTodo({ text: 'Buy milk' })

// {type : "ADD_TODO", payload : {text : "Buy milk"}})
```

`createAction` also accepts a "prepare callback" argument, which allows you to customize the resulting `payload` field and optionally add a `meta` field. See the `createAction` API reference for details on defining action creators with a prepare callback.

Using Action Creators as Action Types

Redux reducers need to look for specific action types to determine how they should update their state. Normally, this is done by defining action type strings and action creator functions separately. Redux Toolkit `createAction` function uses a couple tricks to make this easier.

First, `createAction` overrides the `toString()` method on the action creators it generates. **This means that the action creator itself can be used as the "action type" reference in some places**, such as the keys provided to `builder.addCase` or the `createReducer` object notation.

Second, the action type is also defined as a `type` field on the action creator.

```
const actionCreator = createAction('SOME_ACTION_TYPE')

console.log(actionCreator.toString())

// "SOME_ACTION_TYPE"

console.log(actionCreator.type)

// "SOME_ACTION_TYPE"

const reducer = createReducer({}, (builder) => {

 // actionCreator.toString() will automatically be called here

 // also, if you use TypeScript, the action type will be correctly inferred

 builder.addCase(actionCreator, (state, action) => {})

 // Or, you can reference the .type field:

 // if using TypeScript, the action type cannot be inferred that way

builder.addCase(actionCreator.type, (state, action) => {})})
```

This means you don't have to write or use a separate action type variable, or repeat the name and value of an action type like `const SOME_ACTION_TYPE = "SOME_ACTION_TYPE"`.

Unfortunately, the implicit conversion to a string doesn't happen for switch statements. If you want to use one of these action creators in a switch statement, you need to call `actionCreator.toString()` yourself:

```
const actionCreator = createAction('SOME_ACTION_TYPE')

const reducer = (state = {}, action) => {
  switch (action.type) {
    // ERROR: this won't work correctly!
    case actionCreator: {
      break
    }
    // CORRECT: this will work as expected
    case actionCreator.toString(): {
      break
    }
    // CORRECT: this will also work right
    case actionCreator.type: {
      break
    }
  }
}
```

If you are using Redux Toolkit with TypeScript, note that the TypeScript compiler may not accept the implicit `toString()` conversion when the action creator is used as an object key. In that case, you may need to either manually cast it to a string (`actionCreator as string`), or use the `.type` field as the key.

Creating Slices of State

Redux state is typically organized into "slices", defined by the reducers that are passed to `combineReducers`:

```
import { combineReducers } from 'redux'
import usersReducer from './usersReducer'
import postsReducer from './postsReducer'

const rootReducer = combineReducers({
  users: usersReducer,
  posts: postsReducer,
})
```

In this example, both `users` and `posts` would be considered "slices". Both of the reducers:

- "Own" a piece of state, including what the initial value is
- Define how that state is updated
- Define which specific actions result in state updates

The common approach is to define a slice's reducer function in its own file, and the action creators in a second file. Because both functions need to refer to the same action types, those are usually defined in a third file and imported in both places:

```
// postsConstants.js
const CREATE_POST = 'CREATE_POST'
const UPDATE_POST = 'UPDATE_POST'
const DELETE_POST = 'DELETE_POST'

// postsActions.js
import { CREATE_POST, UPDATE_POST, DELETE_POST } from './postConstants'

export function addPost(id, title) {
  return {
    type: CREATE_POST,
    payload: { id, title },
  }
}

// postsReducer.js
import { CREATE_POST, UPDATE_POST, DELETE_POST } from './postConstants'

const initialState = []

export default function postsReducer(state = initialState, action) {
  switch (action.type) {
    case CREATE_POST: {
      // omit implementation
    }
    default:
      return state
  }
}
```

The only truly necessary part here is the reducer itself. Consider the other parts:

- We could have written the action types as inline strings in both places
- The action creators are good, but they're not *required* to use Redux - a component could skip supplying a `mapDispatch` argument to `connect`, and

just call `this.props.dispatch({type : "CREATE_POST", payload : {id : 123, title : "Hello World"}})` itself

- The only reason we're even writing multiple files is because it's common to separate code by what it does

The "ducks" file structure proposes putting all of your Redux-related logic for a given slice into a single file, like this:

```
// postsDuck.js
const CREATE_POST = 'CREATE_POST'
const UPDATE_POST = 'UPDATE_POST'
const DELETE_POST = 'DELETE_POST'

export function addPost(id, title) {
  return {
    type: CREATE_POST,
    payload: { id, title },
  }
}

const initialState = []

export default function postsReducer(state = initialState, action) {
  switch (action.type) {
    case CREATE_POST: {
      // Omit actual code
      break
    }
    default:
      return state
  }
}
```

That simplifies things because we don't need to have multiple files, and we can remove the redundant imports of the action type constants. But, we still have to write the action types and the action creators by hand.

Defining Functions in Objects

In modern JavaScript, there are several legal ways to define both keys and functions in an object (and this isn't specific to Redux), and you can mix and match different key definitions and function definitions. For example, these are all legal ways to define a function inside an object:

```
const keyName = "ADD_TODO4";

const reducerObject = {
  // Explicit quotes for the key name, arrow function for the reducer
  "ADD_TODO1" : (state, action) => { }

  // Bare key with no quotes, function keyword
  ADD_TODO2 : function(state, action){  }

  // Object literal function shorthand
  ADD_TODO3(state, action) { }

  // Computed property
  [keyName] : (state, action) => { }
}
```

Using the "object literal function shorthand" is probably the shortest code, but feel free to use whichever of those approaches you want.

Simplifying Slices with `createSlice`

To simplify this process, Redux Toolkit includes a `createSlice` function that will auto-generate the action types and action creators for you, based on the names of the reducer functions you provide.

Here's how that posts example would look with `createSlice`:

```
const postsSlice = createSlice({
  name: 'posts',
  initialState: [],
  reducers: {
    createPost(state, action) {},
    updatePost(state, action) {},
    deletePost(state, action) {},
  },
})

console.log(postsSlice)
/*
{
  name: 'posts',
  actions : {
    createPost,
    updatePost,
    deletePost,
  },
  reducer
}
*/
```

```
const { createPost } = postsSlice.actions

console.log(createPost({ id: 123, title: 'Hello World' }))
// {type : "posts/createPost", payload : {id : 123, title : "Hello World"}}
```

`createSlice` looked at all of the functions that were defined in the `reducers` field, and for every "case reducer" function provided, generates an action creator that uses the name of the reducer as the action type itself. So, the `createPost` reducer became an action type of `"posts/createPost"`, and the `createPost()` action creator will return an action with that type.

Exporting and Using Slices

Most of the time, you'll want to define a slice, and export its action creators and reducers. The recommended way to do this is using ES6 destructuring and export syntax:

```
const postsSlice = createSlice({
  name: 'posts',
  initialState: [],
  reducers: {
    createPost(state, action) {},
    updatePost(state, action) {},
    deletePost(state, action) {},
  },
})

// Extract the action creators object and the reducer
const { actions, reducer } = postsSlice
// Extract and export each action creator by name
export const { createPost, updatePost, deletePost } = actions
// Export the reducer, either as a default or named export
export default reducer
```

You could also just export the slice object itself directly if you prefer.

Slices defined this way are very similar in concept to the "Redux Ducks" pattern for defining and exporting action creators and reducers. However, there are a couple potential downsides to be aware of when importing and exporting slices.

First, **Redux action types are not meant to be exclusive to a single slice**. Conceptually, each slice reducer "owns" its own piece of the Redux state, but it should be able to listen to any action type and update its state appropriately. For example, many different slices might want to respond to a "user logged out" action by clearing data or resetting back to initial state values. Keep that in mind as you design your state shape and create your slices.

Second, **JS modules can have "circular reference" problems if two modules try to import each other**. This can result in imports being undefined, which will likely break the code that needs that import. Specifically in the case of "ducks" or slices, this can occur if slices defined in two different files both want to respond to actions defined in the other file.

Asynchronous Logic and Data Fetching

Using Middleware to Enable Async Logic

By itself, a Redux store doesn't know anything about async logic. It only knows how to synchronously dispatch actions, update the state by calling the root reducer function, and notify the UI that something has changed. Any asynchronicity has to happen outside the store.

But, what if you want to have async logic interact with the store by dispatching or checking the current store state? That's where Redux middleware come in. They extend the store, and allow you to:

- Execute extra logic when any action is dispatched (such as logging the action and state)
- Pause, modify, delay, replace, or halt dispatched actions
- Write extra code that has access to `dispatch` and `getState`
- Teach `dispatch` how to accept other values besides plain action objects, such as functions and promises, by intercepting them and dispatching real action objects instead

The most common reason to use middleware is to allow different kinds of async logic to interact with the store. This allows you to write code that can dispatch actions and check the store state, while keeping that logic separate from your UI.

There are many kinds of async middleware for Redux, and each lets you write your logic using different syntax. The most common async middleware are:

- `redux-thunk`, which lets you write plain functions that may contain async logic directly
- `redux-saga`, which uses generator functions that return descriptions of behavior so they can be executed by the middleware
- `redux-observable`, which uses the RxJS observable library to create chains of functions that process actions

> - TIP
> - Redux Toolkit's **RTK Query data fetching API** is a purpose built data fetching and caching solution for Redux apps, and can **eliminate the need to write *any* thunks or reducers to manage data fetching**. We encourage you to try it out and see if it can help simplify the data fetching code in your own apps!

If you do need to write data fetching logic yourself, we recommend using the Redux Thunk middleware as the standard approach, as it is sufficient for most typical use cases (such as basic AJAX data fetching). In addition, use of the `async/await` syntax in thunks makes them easier to read.

The Redux Toolkit `configureStore` function automatically sets up the thunk middleware by default, so you can immediately start writing thunks as part of your application code.

Defining Async Logic in Slices

Redux Toolkit does not currently provide any special APIs or syntax for writing thunk functions. In particular, **they cannot be defined as part of a `createSlice()` call**. You have to write them separate from the reducer logic, exactly the same as with plain Redux code.

Thunks typically dispatch plain actions, such as `dispatch(dataLoaded(response.data))`.

Many Redux apps have structured their code using a "folder-by-type" approach. In that structure, thunk action creators are usually defined in an "actions" file, alongside the plain action creators.

Because we don't have separate "actions" files, **it makes sense to write these thunks directly in our "slice" files**. That way, they have access to the plain action creators from the slice, and it's easy to find where the thunk function lives.

A typical slice file that includes thunks would look like this:

```
// First, define the reducer and action creators via `createSlice`
const usersSlice = createSlice({
  name: 'users',
  initialState: {
    loading: 'idle',
    users: [],
  },
  reducers: {
    usersLoading(state, action) {
      // Use a "state machine" approach for loading state instead of booleans
      if (state.loading === 'idle') {
        state.loading = 'pending'
      }
    },
    usersReceived(state, action) {
      if (state.loading === 'pending') {
        state.loading = 'idle'
        state.users = action.payload
      }
    },
  },
})

// Destructure and export the plain action creators
export const { usersLoading, usersReceived } = usersSlice.actions

// Define a thunk that dispatches those action creators
const fetchUsers = () => async (dispatch) => {
  dispatch(usersLoading())
  const response = await usersAPI.fetchAll()
  dispatch(usersReceived(response.data))
}
```

Redux Data Fetching Patterns

Data fetching logic for Redux typically follows a predictable pattern:

- A "start" action is dispatched before the request to indicate that the request is in progress. This may be used to track loading state, to allow skipping duplicate requests, or show loading indicators in the UI.
- The async request is made
- Depending on the request result, the async logic dispatches either a "success" action containing the result data, or a "failure" action containing error details. The reducer logic clears the loading state in both cases, and either processes the result data from the success case, or stores the error value for potential display.

These steps are not required, but are recommended in the Redux tutorials as a suggested pattern.

A typical implementation might look like:

```
const getRepoDetailsStarted = () => ({
  type: "repoDetails/fetchStarted"
})
const getRepoDetailsSuccess = (repoDetails) => ({
  type: "repoDetails/fetchSucceeded",
  payload: repoDetails
})
const getRepoDetailsFailed = (error) => ({
  type: "repoDetails/fetchFailed",
  error
})
const fetchIssuesCount = (org, repo) => async dispatch => {
  dispatch(getRepoDetailsStarted())
  try {
    const repoDetails = await getRepoDetails(org, repo)
    dispatch(getRepoDetailsSuccess(repoDetails))
  } catch (err) {
    dispatch(getRepoDetailsFailed(err.toString()))
  }
}
```

However, writing code using this approach is tedious. Each separate type of request needs repeated similar implementation:

- Unique action types need to be defined for the three different cases
- Each of those action types usually has a corresponding action creator function
- A thunk has to be written that dispatches the correct actions in the right sequence

`createAsyncThunk` abstracts this pattern by generating the action types and action creators and generating a thunk that dispatches those actions.

Async Requests with `createAsyncThunk`

As a developer, you are probably most concerned with the actual logic needed to make an API request, what action type names show up in the Redux action history log, and how your reducers should process the fetched data. The repetitive details of defining the multiple action types and dispatching the actions in the right sequence aren't what matters.

`createAsyncThunk` simplifies this process - you only need to provide a string for the action type prefix and a payload creator callback that does the actual async logic and returns a promise with the result. In return, `createAsyncThunk` will give you a thunk that will take care of dispatching the right actions based on the promise you return, and action types that you can handle in your reducers:

```
import { createAsyncThunk, createSlice } from '@reduxjs/toolkit'
import { userAPI } from './userAPI'

// First, create the thunk
const fetchUserById = createAsyncThunk(
  'users/fetchByIdStatus',
  async (userId, thunkAPI) => {
    const response = await userAPI.fetchById(userId)
    return response.data
  }
)

// Then, handle actions in your reducers:
const usersSlice = createSlice({
  name: 'users',
  initialState: { entities: [], loading: 'idle' },
  reducers: {
    // standard reducer logic, with auto-generated action types per reducer
  },
  extraReducers: (builder) => {
    // Add reducers for additional action types here, and handle loading state as needed
    builder.addCase(fetchUserById.fulfilled, (state, action) => {
      // Add user to the state array
      state.entities.push(action.payload)
    })
  },
})

// Later, dispatch the thunk as needed in the app
dispatch(fetchUserById(123))
```

The thunk action creator accepts a single argument, which will be passed as the first argument to your payload creator callback.

The payload creator will also receive a `thunkAPI` object containing the parameters that are normally passed to a standard Redux thunk function, as well as an auto-generated unique random request ID string and an `AbortController.signal` object:

```
interface ThunkAPI {
  dispatch: Function
  getState: Function
  extra?: any
  requestId: string
  signal: AbortSignal
}
```

You can use any of these as needed inside the payload callback to determine what the final result should be.

Managing Normalized Data

Most applications typically deal with data that is deeply nested or relational. The goal of normalizing data is to efficiently organize the data in your state. This is typically done by storing collections as objects with the key of an `id`, while storing a sorted array of those `ids`. For a more in-depth explanation and further examples, there is a great reference in the Redux docs page on "Normalizing State Shape".

Normalizing by hand

Normalizing data doesn't require any special libraries. Here's a basic example of how you might normalize the response from a `fetchAll` API request that returns data in the shape of `{ users: [{id: 1, first_name: 'normalized', last_name: 'person'}] }`, using some hand-written logic:

```
import { createSlice, createAsyncThunk } from '@reduxjs/toolkit'
import userAPI from './userAPI'

export const fetchUsers = createAsyncThunk('users/fetchAll', async () => {
  const response = await userAPI.fetchAll()
  return response.data
})

export const slice = createSlice({
  name: 'users',
  initialState: {
    ids: [],
    entities: {},
  },
  reducers: {},
  extraReducers: (builder) => {
    builder.addCase(fetchUsers.fulfilled, (state, action) => {
      // reduce the collection by the id property into a shape of { 1: { ...user }}
      const byId = action.payload.users.reduce((byId, user) => {
        byId[user.id] = user
        return byId
      }, {})
      state.entities = byId
      state.ids = Object.keys(byId)
    })
  },
})
```

Although we're capable of writing this code, it does become repetitive, especially if you're handling multiple types of data. In addition, this example only handles loading entries into the state, not updating them.

Normalizing with `normalizr`

`normalizr` is a popular existing library for normalizing data. You can use it on its own without Redux, but it is very commonly used with Redux. The typical usage is to format collections from an API response and then process them in your reducers.

```
import { createSlice, createAsyncThunk } from '@reduxjs/toolkit'
import { normalize, schema } from 'normalizr'

import userAPI from './userAPI'

const userEntity = new schema.Entity('users')

export const fetchUsers = createAsyncThunk('users/fetchAll', async () => {
  const response = await userAPI.fetchAll()
  // Normalize the data before passing it to our reducer
  const normalized = normalize(response.data, [userEntity])
  return normalized.entities
})

export const slice = createSlice({
  name: 'users',
  initialState: {
    ids: [],
    entities: {},
  },
  reducers: {},
  extraReducers: (builder) => {
    builder.addCase(fetchUsers.fulfilled, (state, action) => {
      state.entities = action.payload.users
      state.ids = Object.keys(action.payload.users)
    })
  },
})
```

As with the hand-written version, this doesn't handle adding additional entries into the state, or updating them later - it's just loading in everything that was received.

Normalizing with `createEntityAdapter`

Redux Toolkit's `createEntityAdapter` API provides a standardized way to store your data in a slice by taking a collection and putting it into the shape of `{ ids: [], entities: {} }`. Along with this predefined state shape, it generates a set of reducer functions and selectors that know how to work with the data.

```
import {
  createSlice,
  createAsyncThunk,
  createEntityAdapter,
} from '@reduxjs/toolkit'
import userAPI from './userAPI'

export const fetchUsers = createAsyncThunk('users/fetchAll', async () => {
  const response = await userAPI.fetchAll()
  // In this case, `response.data` would be:
  // [{id: 1, first_name: 'Example', last_name: 'User'}]
  return response.data
})

export const updateUser = createAsyncThunk('users/updateOne', async (arg) => {
  const response = await userAPI.updateUser(arg)
  // In this case, `response.data` would be:
  // { id: 1, first_name: 'Example', last_name: 'UpdatedLastName'}
  return response.data
})

export const usersAdapter = createEntityAdapter()

// By default, `createEntityAdapter` gives you `{ ids: [], entities: {} }`.
// If you want to track 'loading' or other keys, you would initialize them here:
// `getInitialState({ loading: false, activeRequestId: null })`
const initialState = usersAdapter.getInitialState()

export const slice = createSlice({
  name: 'users',
  initialState,
  reducers: {
    removeUser: usersAdapter.removeOne,
  },
  extraReducers: (builder) => {
    builder.addCase(fetchUsers.fulfilled, usersAdapter.upsertMany)
    builder.addCase(updateUser.fulfilled, (state, { payload }) => {
      const { id, ...changes } = payload
      usersAdapter.updateOne(state, { id, changes })
    })
  },
})

const reducer = slice.reducer
export default reducer

export const { removeUser } = slice.actions
```

Using `createEntityAdapter` with Normalization Libraries

If you're already using `normalizr` or another normalization library, you could consider using it along with `createEntityAdapter`. To expand on the examples above, here is a demonstration of how we could use `normalizr` to format a payload, then leverage the utilities `createEntityAdapter` provides.

By default, the `setAll`, `addMany`, and `upsertMany` CRUD methods expect an array of entities. However, they also allow you to pass in an object that is in the shape of `{ 1: { id: 1, ... }}` as an alternative, which makes it easier to insert pre-normalized data.

```
// features/articles/articlesSlice.js
import {
  createSlice,
  createEntityAdapter,
  createAsyncThunk,
  createSelector,
} from '@reduxjs/toolkit'
import fakeAPI from '../../services/fakeAPI'
import { normalize, schema } from 'normalizr'

// Define normalizr entity schemas
export const userEntity = new schema.Entity('users')
export const commentEntity = new schema.Entity('comments', {
  commenter: userEntity,
})
export const articleEntity = new schema.Entity('articles', {
  author: userEntity,
  comments: [commentEntity],
})

const articlesAdapter = createEntityAdapter()

export const fetchArticle = createAsyncThunk(
  'articles/fetchArticle',
  async (id) => {
    const data = await fakeAPI.articles.show(id)
    // Normalize the data so reducers can load a predictable payload, like:
    // `action.payload = { users: {}, articles: {}, comments: {} }`
    const normalized = normalize(data, articleEntity)
    return normalized.entities
  }
)



export const slice = createSlice({
  name: 'articles',
  initialState: articlesAdapter.getInitialState(),
  reducers: {},
  extraReducers: (builder) => {
    builder.addCase(fetchArticle.fulfilled, (state, action) => {
      // Handle the fetch result by inserting the articles here
      articlesAdapter.upsertMany(state, action.payload.articles)
    })
  },
})

const reducer = slice.reducer
export default reducer

// features/users/usersSlice.js

import { createSlice, createEntityAdapter } from '@reduxjs/toolkit'
import { fetchArticle } from '../articles/articlesSlice'

const usersAdapter = createEntityAdapter()

export const slice = createSlice({
  name: 'users',
  initialState: usersAdapter.getInitialState(),
  reducers: {},
  extraReducers: (builder) => {
    builder.addCase(fetchArticle.fulfilled, (state, action) => {
      // And handle the same fetch result by inserting the users here
      usersAdapter.upsertMany(state, action.payload.users)
    })
  },
})

const reducer = slice.reducer
export default reducer

// features/comments/commentsSlice.js

import { createSlice, createEntityAdapter } from '@reduxjs/toolkit'
import { fetchArticle } from '../articles/articlesSlice'

const commentsAdapter = createEntityAdapter()

export const slice = createSlice({
  name: 'comments',
  initialState: commentsAdapter.getInitialState(),
  reducers: {},
  extraReducers: (builder) => {
    builder.addCase(fetchArticle.fulfilled, (state, action) => {
      // Same for the comments
      commentsAdapter.upsertMany(state, action.payload.comments)
    })
  },
})

const reducer = slice.reducer
export default reducer
```

Using selectors with `createEntityAdapter`

The entity adapter provides a selector factory that generates the most common selectors for you. Taking the examples above, we can add selectors to our `usersSlice` like this:

```
// Rename the exports for readability in component usage
export const {
  selectById: selectUserById,
  selectIds: selectUserIds,
  selectEntities: selectUserEntities,
  selectAll: selectAllUsers,
  selectTotal: selectTotalUsers,
} = usersAdapter.getSelectors((state) => state.users)
```

You could then use these selectors in a component like this:

```
import React from 'react'
import { useSelector } from 'react-redux'
import { selectTotalUsers, selectAllUsers } from './usersSlice'

import styles from './UsersList.module.css'

export function UsersList() {
  const count = useSelector(selectTotalUsers)
  const users = useSelector(selectAllUsers)

  return (
    <div>
      <div className={styles.row}>
        There are <span className={styles.value}>{count}</span> users.{' '}
        {count === 0 && `Why don't you fetch some more?`}
      </div>
      {users.map((user) => (
        <div key={user.id}>
          <div>{`${user.first_name} ${user.last_name}`}</div>
        </div>
      ))}
    </div>
  )
}
```

Specifying Alternate ID Fields

By default, `createEntityAdapter` assumes that your data has unique IDs in an `entity.id` field. If your data set stores its ID in a different field, you can pass in a `selectId` argument that returns the appropriate field.

```
// In this instance, our user data always has a primary key of `idx`
const userData = {
  users: [
    { idx: 1, first_name: 'Test' },
    { idx: 2, first_name: 'Two' },
  ],
}

// Since our primary key is `idx` and not `id`,
// pass in an ID selector to return that field instead
export const usersAdapter = createEntityAdapter({
  selectId: (user) => user.idx,
})
```

Sorting Entities

`createEntityAdapter` provides a `sortComparer` argument that you can leverage to sort the collection of `ids` in state. This can be very useful for when you want to guarantee a sort order and your data doesn't come presorted.

```
// In this instance, our user data always has a primary key of `id`, so we do not need to provide
`selectId`.
const userData = {
  users: [
    { id: 1, first_name: 'Test' },
    { id: 2, first_name: 'Banana' },
  ],
}

// Sort by `first_name`. `state.ids` would be ordered as
// `ids: [ 2, 1 ]`, since 'B' comes before 'T'.
// When using the provided `selectAll` selector, the result would be sorted:
// [{ id: 2, first_name: 'Banana' }, { id: 1, first_name: 'Test' }]
export const usersAdapter = createEntityAdapter({
  sortComparer: (a, b) => a.first_name.localeCompare(b.first_name),
})
```

Working with Non-Serializable Data

One of the core usage principles for Redux is that you should not put non-serializable values in state or actions.

However, like most rules, there are exceptions. There may be occasions when you have to deal with actions that need to accept non-serializable data. This should be done very rarely and only if necessary, and these non-serializable payloads shouldn't ever make it into your application state through a reducer.

The serializability dev check middleware will automatically warn anytime it detects non-serializable values in your actions or state. We encourage you to leave this middleware active to help avoid accidentally making mistakes. However, if you *do* need to turnoff those warnings, you can customize the middleware by configuring it to ignore specific action types, or fields in actions and state:

```
configureStore({
  //...
  middleware: (getDefaultMiddleware) =>
    getDefaultMiddleware({
      serializableCheck: {
        // Ignore these action types
        ignoredActions: ['your/action/type'],
        // Ignore these field paths in all actions
        ignoredActionPaths: ['meta.arg', 'payload.timestamp'],
        // Ignore these paths in the state
        ignoredPaths: ['items.dates'],
      },
    }),
})
```

Use with Redux-Persist

If using Redux-Persist, you should specifically ignore all the action types it dispatches:

```
import { configureStore } from '@reduxjs/toolkit'
import {
  persistStore,
  persistReducer,
  FLUSH,
  REHYDRATE,
  PAUSE,
  PERSIST,
  PURGE,
  REGISTER,
} from 'redux-persist'
import storage from 'redux-persist/lib/storage'
import { PersistGate } from 'redux-persist/integration/react'

import App from './App'
import rootReducer from './reducers'
```

```
const persistConfig = {
  key: 'root',
  version: 1,
  storage,
}

const persistedReducer = persistReducer(persistConfig, rootReducer)

const store = configureStore({
  reducer: persistedReducer,
  middleware: (getDefaultMiddleware) =>
    getDefaultMiddleware({
      serializableCheck: {
        ignoredActions: [FLUSH, REHYDRATE, PAUSE, PERSIST, PURGE, REGISTER],
      },
    }),
})

let persistor = persistStore(store)

ReactDOM.render(
  <Provider store={store}>
    <PersistGate loading={null} persistor={persistor}>
      <App />
    </PersistGate>
  </Provider>,
  document.getElementById('root')
)
```

Additionally, you can purge any persisted state by adding an extra reducer to the specific slice that you would like to clear when calling persistor.purge(). This is especially helpful when you are looking to clear persisted state on a dispatched logout action.

```
import { PURGE } from "redux-persist";

...
extraReducers: (builder) => {
    builder.addCase(PURGE, (state) => {
        customEntityAdapter.removeAll(state);
    });
}
```

It is also strongly recommended to blacklist any api(s) that you have configured with RTK Query. If the api slice reducer is not blacklisted, the api cache will be automatically persisted and restored which could leave you with phantom subscriptions from components that do not exist any more. Configuring this should look something like this:

```
const persistConfig = {
  key: "root",
  version: 1,
  storage,
  blacklist: [pokemonApi.reducerPath],
};
```

See Redux Toolkit #121: How to use this with Redux-Persist? and Redux-Persist #988: non-serializable value error for further discussion.

Use with React-Redux-Firebase

RRF includes timestamp values in most actions and state as of 3.x, but there are PRs that may improve that behavior as of 4.x.

A possible configuration to work with that behavior could look like:

```
import { configureStore } from '@reduxjs/toolkit'
import {
  getFirebase,
  actionTypes as rrfActionTypes,
} from 'react-redux-firebase'
import { constants as rfConstants } from 'redux-firestore'
import rootReducer from './rootReducer'

const store = configureStore({
  reducer: rootReducer,
  middleware: (getDefaultMiddleware) =>
    getDefaultMiddleware({
      serializableCheck: {
        ignoredActions: [
          // just ignore every redux-firebase and react-redux-firebase action type
          ...Object.keys(rfConstants.actionTypes).map(
            (type) => `${rfConstants.actionsPrefix}/${type}`
          ),
          ...Object.keys(rrfActionTypes).map(
            (type) => `@@reactReduxFirebase/${type}`
          ),
        ],
        ignoredPaths: ['firebase', 'firestore'],
      },
      thunk: {
        extraArgument: {
          getFirebase,
        },
      },
    }),
})

export default store
```

*Last updated on **Apr 28, 2022***

Usage With TypeScript

Introduction

Redux Toolkit is written in TypeScript, and its API is designed to enable great integration with TypeScript applications.

This page provides specific details for each of the different APIs included in Redux Toolkit and how to type them correctly with TypeScript.

See the [TypeScript Quick Start tutorial page](#) for a brief overview of how to set up and use Redux Toolkit and React Redux to work with TypeScript.

`configureStore`

The basics of using `configureStore` are shown in [TypeScript Quick Start tutorial page](#). Here are some additional details that you might find useful.

Getting the `State` type

The easiest way of getting the `State` type is to define the root reducer in advance and extract its `ReturnType`.
It is recommended to give the type a different name like `RootState` to prevent confusion, as the type name `State` is usually overused.

```
import { combineReducers } from '@reduxjs/toolkit'

const rootReducer = combineReducers({})

export type RootState = ReturnType<typeof rootReducer>
```

Alternatively, if you choose to not create a `rootReducer` yourself and instead pass the slice reducers directly to `configureStore()`, you need to slightly modify the typing to correctly infer the root reducer:

```
import { configureStore } from '@reduxjs/toolkit'
// ...
const store = configureStore({
  reducer: {
    one: oneSlice.reducer,
    two: twoSlice.reducer,
  },
})
export type RootState = ReturnType<typeof store.getState>

export default store
```

If you pass the reducers directly to `configureStore()` and do not define the root reducer explicitly, there is no reference to `rootReducer`. Instead, you can refer to `store.getState`, in order to get the `State` type.

```
import { configureStore } from '@reduxjs/toolkit'
import rootReducer from './rootReducer'
const store = configureStore({
  reducer: rootReducer,
})
export type RootState = ReturnType<typeof store.getState>
```

Getting the `Dispatch` type

If you want to get the `Dispatch` type from your store, you can extract it after creating the store. It is recommended to give the type a different name like `AppDispatch` to prevent confusion, as the type name `Dispatch` is usually overused. You may also find it to be more convenient to export a hook like `useAppDispatch` shown below, then using it wherever you'd call `useDispatch`.

```
import { configureStore } from '@reduxjs/toolkit'
import { useDispatch } from 'react-redux'
import rootReducer from './rootReducer'

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

export type AppDispatch = typeof store.dispatch
export const useAppDispatch: () => AppDispatch = useDispatch // Export a hook that can be reused to resolve types

export default store
```

Correct typings for the `Dispatch` type

The type of the `dispatch` function type will be directly inferred from the `middleware` option. So if you add *correctly typed* middlewares, `dispatch` should already be correctly typed.

As TypeScript often widens array types when combining arrays using the spread operator, we suggest using the `.concat(...)` and `.prepend(...)` methods of the `MiddlewareArray` returned by `getDefaultMiddleware()`.

```
import { configureStore } from '@reduxjs/toolkit'
import additionalMiddleware from 'additional-middleware'
import logger from 'redux-logger'
// @ts-ignore
import untypedMiddleware from 'untyped-middleware'
import rootReducer from './rootReducer'

export type RootState = ReturnType<typeof rootReducer>
const store = configureStore({
  reducer: rootReducer,
  middleware: (getDefaultMiddleware) =>
    getDefaultMiddleware()
      .prepend(
        // correctly typed middlewares can just be used
        additionalMiddleware,
        // you can also type middlewares manually
        untypedMiddleware as Middleware<
          (action: Action<'specialAction'>) => number,
          RootState
        >
      )
      // prepend and concat calls can be chained
      .concat(logger),
})

export type AppDispatch = typeof store.dispatch

export default store
```

Using `MiddlewareArray` without `getDefaultMiddleware`

If you want to skip the usage of `getDefaultMiddleware` altogether, you can still use `MiddlewareArray` for type-safe concatenation of your `middleware` array. This class extends the default JavaScript `Array` type, only with modified typings for `.concat(...)` and the additional `.prepend(...)` method.

This is generally not required though, as you will probably not run into any array-type-widening issues as long as you are using `as const` and do not use the spread operator.

So the following two calls would be equivalent:

```
import { configureStore, MiddlewareArray } from '@reduxjs/toolkit'

configureStore({
  reducer: rootReducer,
  middleware: new MiddlewareArray().concat(additionalMiddleware, logger),
})

configureStore({
  reducer: rootReducer,
  middleware: [additionalMiddleware, logger] as const,
})
```

Using the extracted `Dispatch` type with React Redux

By default, the React Redux `useDispatch` hook does not contain any types that take middlewares into account. If you need a more specific type for the `dispatch` function when dispatching, you may specify the type of the returned `dispatch` function, or create a custom-typed version of `useSelector`. See the React Redux documentation for details.

`createAction`

For most use cases, there is no need to have a literal definition of `action.type`, so the following can be used:

```
createAction<number>('test')
```

This will result in the created action being of type `PayloadActionCreator<number, string>`.

In some setups, you will need a literal type for `action.type`, though. Unfortunately, TypeScript type definitions do not allow for a mix of manually-defined and inferred type parameters, so you'll have to specify the `type` both in the Generic definition as well as in the actual JavaScript code:

```
createAction<number, 'test'>('test')
```

If you are looking for an alternate way of writing this without the duplication, you can use a prepare callback so that both type parameters can be inferred from arguments, removing the need to specify the action type.

```
function withPayloadType<T>() {
  return (t: T) => ({ payload: t })
}
createAction('test', withPayloadType<string>())
```

Alternative to using a literally-typed `action.type`

If you are using `action.type` as a discriminator on a discriminated union, for example to correctly type your payload in `case` statements, you might be interested in this alternative:

Created action creators have a `match` method that acts as a type predicate:

```
const increment = createAction<number>('increment')
function test(action: Action) {
  if (increment.match(action)) {
    // action.payload inferred correctly here
    action.payload
  }
}
```

This `match` method is also very useful in combination with `redux-observable` and RxJS's `filter` method.

`createReducer`

The default way of calling `createReducer` would be with a "lookup table" / "map object", like this:

```
createReducer(0, {
  increment: (state, action: PayloadAction<number>) => state + action.payload,
})
```

Unfortunately, as the keys are only strings, using that API TypeScript can neither infer nor validate the action types for you:

```
{
  const increment = createAction<number, 'increment'>('increment')
  const decrement = createAction<number, 'decrement'>('decrement')
  createReducer(0, {
    [increment.type]: (state, action) => {
      // action is any here
    },
    [decrement.type]: (state, action: PayloadAction<string>) => {
      // even though action should actually be PayloadAction<number>, TypeScript can't detect that and won't give a warning here.
    },
  })
}
```

As an alternative, RTK includes a type-safe reducer builder API.

Building Type-Safe Reducer Argument Objects

Instead of using a simple object as an argument to `createReducer`, you can also use a callback that receives a `ActionReducerMapBuilder` instance:

```
const increment = createAction<number, 'increment'>('increment')
const decrement = createAction<number, 'decrement'>('decrement')
createReducer(0, (builder) =>
  builder
    .addCase(increment, (state, action) => {
      // action is inferred correctly here
    })
    .addCase(decrement, (state, action: PayloadAction<string>) => {
      // this would error out
    })
)
```

We recommend using this API if stricter type safety is necessary when defining reducer argument objects.

Typing `builder.addMatcher`

As the first `matcher` argument to `builder.addMatcher`, a type predicate function should be used. As a result, the `action` argument for the second `reducer` argument can be inferred by TypeScript:

```
function isNumberValueAction(action: AnyAction): action is PayloadAction<{ value: number }> {
  return typeof action.payload.value === 'number'
}

createReducer({ value: 0 }, builder =>
   builder.addMatcher(isNumberValueAction, (state, action) => {
      state.value += action.payload.value
   })
})
```

`createSlice`

As `createSlice` creates your actions as well as your reducer for you, you don't have to worry about type safety here. Action types can just be provided inline:

```
const slice = createSlice({
  name: 'test',
  initialState: 0,
  reducers: {
    increment: (state, action: PayloadAction<number>) => state + action.payload,
  },
})
// now available:
slice.actions.increment(2)
// also available:
slice.caseReducers.increment(0, { type: 'increment', payload: 5 })
```

If you have too many case reducers and defining them inline would be messy, or you want to reuse case reducers across slices, you can also define them outside the `createSlice` call and type them as `CaseReducer`:

```
type State = number
const increment: CaseReducer<State, PayloadAction<number>> = (state, action) =>
  state + action.payload

createSlice({
  name: 'test',
  initialState: 0,
  reducers: {
    increment,
  },
})
```

Defining the Initial State Type

You might have noticed that it is not a good idea to pass your `SliceState` type as a generic to `createSlice`. This is due to the fact that in almost all cases, follow-up generic parameters to `createSlice` need to be inferred, and TypeScript cannot mix explicit declaration and inference of generic types within the same "generic block".

The standard approach is to declare an interface or type for your state, create an initial state value that uses that type, and pass the initial state value to `createSlice`. You can also use the construct `initialState: myInitialState as SliceState`.

```
type SliceState = { state: 'loading' } | { state: 'finished'; data: string }

// First approach: define the initial state using that type
const initialState: SliceState = { state: 'loading' }

createSlice({
  name: 'test1',
  initialState, // type SliceState is inferred for the state of the slice
  reducers: {},
})

// Or, cast the initial state as necessary
createSlice({
  name: 'test2',
  initialState: { state: 'loading' } as SliceState,
  reducers: {},
})
```

which will result in a `Slice<SliceState, ...>`.

Defining Action Contents with `prepare` Callbacks

If you want to add a `meta` or `error` property to your action, or customize the `payload` of your action, you have to use the `prepare` notation.

Using this notation with TypeScript looks like this:

```
const blogSlice = createSlice({
  name: 'blogData',
  initialState,
  reducers: {
    receivedAll: {
      reducer(
        state,
        action: PayloadAction<Page[], string, { currentPage: number }>
      ) {
        state.all = action.payload
        state.meta = action.meta
      },
      prepare(payload: Page[], currentPage: number) {
        return { payload, meta: { currentPage } }
      }, }} }
```

Generated Action Types for Slices

As TS cannot combine two string literals (`slice.name` and the key of `actionMap`) into a new literal, all actionCreators created by `createSlice` are of type 'string'. This is usually not a problem, as these types are only rarely used as literals.

In most cases that `type` would be required as a literal,
the `slice.action.myAction.match` type predicate should be a viable alternative:

```
const slice = createSlice({
  name: 'test',
  initialState: 0,
  reducers: {
    increment: (state, action: PayloadAction<number>) => state + action.payload,
  },
})

function myCustomMiddleware(action: Action) {
  if (slice.actions.increment.match(action)) {
    // `action` is narrowed down to the type `PayloadAction<number>` here.
  }
}
```

If you actually *need* that type, unfortunately there is no other way than manual casting.

Type safety with `extraReducers`

Reducer lookup tables that map an action `type` string to a reducer function are not easy to fully type correctly. This affects both `createReducer` and the `extraReducers` argument for `createSlice`. So, like with `createReducer`, you may also use the "builder callback" approach for defining the reducer object argument.

This is particularly useful when a slice reducer needs to handle action types generated by other slices, or generated by specific calls to `createAction` (such as the actions generated by `createAsyncThunk`).

```
const fetchUserById = createAsyncThunk(
  'users/fetchById',
  // if you type your function argument here
  async (userId: number) => {
    const response = await fetch(`https://reqres.in/api/users/${userId}`)
    return (await response.json()) as Returned
  }
)

interface UsersState {
  entities: []
  loading: 'idle' | 'pending' | 'succeeded' | 'failed'
}
```

```
const initialState = {
  entities: [],
  loading: 'idle',
} as UsersState

const usersSlice = createSlice({
  name: 'users',
  initialState,
  reducers: {
    // fill in primary logic here
  },
  extraReducers: (builder) => {
    builder.addCase(fetchUserById.pending, (state, action) => {
      // both `state` and `action` are now correctly typed
      // based on the slice state and the `pending` action creator
    })
  },
})
```

Like the `builder` in `createReducer`, this `builder` also accepts `addMatcher` (see typing `builder.matcher`) and `addDefaultCase`.

Wrapping `createSlice`

If you need to reuse reducer logic, it is common to write "higher-order reducers" that wrap a reducer function with additional common behavior. This can be done with `createSlice` as well, but due to the complexity of the types for `createSlice`, you have to use the `SliceCaseReducers` and `ValidateSliceCaseReducers` types in a very specific way.

Here is an example of such a "generic" wrapped `createSlice` call:

```
interface GenericState<T> {
  data?: T
  status: 'loading' | 'finished' | 'error'
}


const createGenericSlice = <
  T,
  Reducers extends SliceCaseReducers<GenericState<T>>
>({
  name = '',
  initialState,
  reducers,
}: {
  name: string
  initialState: GenericState<T>
  reducers: ValidateSliceCaseReducers<GenericState<T>, Reducers>
}) => {
  return createSlice({
    name,
    initialState,
    reducers: {
      start(state) {
        state.status = 'loading'
      },
      /**
       * If you want to write to values of the state that depend on the generic
       * (in this case: `state.data`, which is T), you might need to specify the
       * State type manually here, as it defaults to `Draft<GenericState<T>>`,
       * which can sometimes be problematic with yet-unresolved generics.
       * This is a general problem when working with immer's Draft type and generics.
       */
      success(state: GenericState<T>, action: PayloadAction<T>) {
        state.data = action.payload
        state.status = 'finished'
      },
      ...reducers,
    },
  })
}

const wrappedSlice = createGenericSlice({
  name: 'test',
  initialState: { status: 'loading' } as GenericState<string>,
  reducers: {
    magic(state) {
      state.status = 'finished'
      state.data = 'hocus pocus'
    },
  },
})
```

`createAsyncThunk`

Basic `createAsyncThunk` Types

In the most common use cases, you should not need to explicitly declare any types for the `createAsyncThunk` call itself.

Just provide a type for the first argument to the `payloadCreator` argument as you would for any function argument, and the resulting thunk will accept the same type as its input parameter. The return type of the `payloadCreator` will also be reflected in all generated action types.

```
interface MyData {
  // ...
}

const fetchUserById = createAsyncThunk(
  'users/fetchById',
  // Declare the type your function argument here:
  async (userId: number) => {
    const response = await fetch(`https://reqres.in/api/users/${userId}`)
    // Inferred return type: Promise<MyData>
    return (await response.json()) as MyData
  }
)

// the parameter of `fetchUserById` is automatically inferred to `number` here
// and dispatching the resulting thunkAction will return a Promise of a correctly
// typed "fulfilled" or "rejected" action.
const lastReturnedAction = await store.dispatch(fetchUserById(3))
```

Typing the `thunkApi` Object

The second argument to the `payloadCreator`, known as `thunkApi`, is an object containing references to the `dispatch`, `getState`, and `extra` arguments from the thunk middleware as well as a utility function called `rejectWithValue`. If you want to use these from within the `payloadCreator`, you will need to define some generic arguments, as the types for these arguments cannot be inferred. Also, as TS cannot mix explicit and inferred generic parameters, from this point on you'll have to define the `Returned` and `ThunkArg` generic parameter as well.

Manually Defining `thunkApi` Types

To define the types for these arguments, pass an object as the third generic argument, with type declarations for some or all of these fields:

```
type AsyncThunkConfig = {
  /** return type for `thunkApi.getState` */
  state?: unknown
  /** type for `thunkApi.dispatch` */
  dispatch?: Dispatch
  /** type of the `extra` argument for the thunk middleware, which will be passed in as `thunkApi.extra` */
  extra?: unknown
  /** type to be passed into `rejectWithValue`'s first argument that will end up on `rejectedAction.payload` */
  rejectValue?: unknown
  /** return type of the `serializeError` option callback */
  serializedErrorType?: unknown
  /** type to be returned from the `getPendingMeta` option callback & merged into `pendingAction.meta` */
  pendingMeta?: unknown
  /** type to be passed into the second argument of `fulfillWithValue` to finally be merged into
`fulfilledAction.meta` */
  fulfilledMeta?: unknown
  /** type to be passed into the second argument of `rejectWithValue` to finally be merged into
`rejectedAction.meta` */
  rejectedMeta?: unknown
}
```

```
const fetchUserById = createAsyncThunk<
  // Return type of the payload creator
  MyData,
  // First argument to the payload creator
  number,
  {
    // Optional fields for defining thunkApi field types
    dispatch: AppDispatch
    state: State
    extra: {
      jwt: string
    }
  }
>('users/fetchById', async (userId, thunkApi) => {
  const response = await fetch(`https://reqres.in/api/users/${userId}`, {
    headers: {
      Authorization: `Bearer ${thunkApi.extra.jwt}`,
    },
  })
  return (await response.json()) as MyData
})
```

If you are performing a request that you know will typically either be a success or have an expected error format, you can pass in a type to `rejectValue` and `return rejectWithValue(knownPayload)` in the action creator. This allows you to reference the error payload in the reducer as well as in a component after dispatching the `createAsyncThunk` action.

```
interface MyKnownError {
  errorMessage: string
  // ...
}
interface UserAttributes {
  id: string
  first_name: string
  last_name: string
  email: string
}

const updateUser = createAsyncThunk<
  // Return type of the payload creator
  MyData,
  // First argument to the payload creator
  UserAttributes,
  // Types for ThunkAPI
  {
    extra: {
      jwt: string
    }
    rejectValue: MyKnownError
  }
>('users/update', async (user, thunkApi) => {
  const { id, ...userData } = user
  const response = await fetch(`https://reqres.in/api/users/${id}`, {
    method: 'PUT',
    headers: {
      Authorization: `Bearer ${thunkApi.extra.jwt}`,
    },
    body: JSON.stringify(userData),
  })
  if (response.status === 400) {
    // Return the known error for future handling
    return thunkApi.rejectWithValue((await response.json()) as MyKnownError)
  }
  return (await response.json()) as MyData
})
```

While this notation for `state`, `dispatch`, `extra` and `rejectValue` might seem uncommon at first, it allows you to provide only the types for these you actually need - so for example, if you are not accessing `getState` within your `payloadCreator`, there is no need to provide a type for `state`. The same can be said about `rejectValue` - if you don't need to access any potential error payload, you can ignore it.

In addition, you can leverage checks against `action.payload` and `match` as provided by `createAction` as a type-guard for when you want to access known properties on defined types. Example:

- In a reducer

```
const usersSlice = createSlice({
  name: 'users',
  initialState: {
    entities: {},
    error: null,
  },
  reducers: {},
  extraReducers: (builder) => {
    builder.addCase(updateUser.fulfilled, (state, { payload }) => {
      state.entities[payload.id] = payload
    })
    builder.addCase(updateUser.rejected, (state, action) => {
      if (action.payload) {
        // Since we passed in `MyKnownError` to `rejectValue` in `updateUser`, the type information will be available
here.
        state.error = action.payload.errorMessage
      } else {
        state.error = action.error
      }
    })
  },
})
```

- In a component

```
const handleUpdateUser = async (userData) => {
  const resultAction = await dispatch(updateUser(userData))
  if (updateUser.fulfilled.match(resultAction)) {
    const user = resultAction.payload
    showToast('success', `Updated ${user.name}`)
  } else {
    if (resultAction.payload) {
      // Since we passed in `MyKnownError` to `rejectValue` in `updateUser`, the type information will be available
here.
      // Note: this would also be a good place to do any handling that relies on the `rejectedWithValue` payload, such
as setting field errors
      showToast('error', `Update failed: ${resultAction.payload.errorMessage}`)
    } else {
      showToast('error', `Update failed: ${resultAction.error.message}`)
    }
  }
}
```

Defining a Pre-Typed `createAsyncThunk`

As of RTK 1.9, you can define a "pre-typed" version of `createAsyncThunk` that can have the types for `state`, `dispatch`, and `extra` built in. This lets you set up those types once, so you don't have to repeat them each time you call `createAsyncThunk`.

To do this, call `createAsyncThunk.withTypes<>()`, and pass in an object containing the field names and types for any of the fields in the `AsyncThunkConfig` type listed above. This might look like:

```
const createAppAsyncThunk = createAsyncThunk.withTypes<{
  state: RootState
  dispatch: AppDispatch
  rejectValue: string
  extra: { s: string; n: number }
}>()
```

Import and use that pre-typed `createAppAsyncThunk` instead of the original, and the types will be used automatically.

`createEntityAdapter`

Typing `createEntityAdapter` only requires you to specify the entity type as the single generic argument.

The example from the `createEntityAdapter` documentation would look like this in TypeScript:

```
interface Book {
  bookId: number
  title: string
  // ...
}
const booksAdapter = createEntityAdapter<Book>({
  selectId: (book) => book.bookId,
  sortComparer: (a, b) => a.title.localeCompare(b.title),
})
const booksSlice = createSlice({
  name: 'books',
  initialState: booksAdapter.getInitialState(),
  reducers: {
    bookAdded: booksAdapter.addOne,
    booksReceived(state, action: PayloadAction<{ books: Book[] }>) {
      booksAdapter.setAll(state, action.payload.books)
    } }, })
```

Using `createEntityAdapter` with `normalizr`

When using a library like [`normalizr`](normalizr), your normalized data will resemble this shape:

```
{
  result: 1,
  entities: {
    1: { id: 1, other: 'property' },
    2: { id: 2, other: 'property' }
  }
}
```

The methods `addMany`, `upsertMany`, and `setAll` all allow you to pass in the `entities` portion of this directly with no extra conversion steps. However, the `normalizr` TS typings currently do not correctly reflect that multiple data types may be included in the results, so you will need to specify that type structure yourself.

Here is an example of how that would look:

```
type Author = { id: number; name: string }
type Article = { id: number; title: string }
type Comment = { id: number; commenter: number }


export const fetchArticle = createAsyncThunk(
  'articles/fetchArticle',
  async (id: number) => {
    const data = await fakeAPI.articles.show(id)
    // Normalize the data so reducers can responded to a predictable payload.
    // Note: at the time of writing, normalizr does not automatically infer the result,
    // so we explicitly declare the shape of the returned normalized data as a generic arg.
    const normalized = normalize<
      any,
      {
        articles: { [key: string]: Article }
        users: { [key: string]: Author }
        comments: { [key: string]: Comment }
      }
    >(data, articleEntity)
    return normalized.entities
  }
)

export const slice = createSlice({
  name: 'articles',
  initialState: articlesAdapter.getInitialState(),
  reducers: {},
  extraReducers: (builder) => {
    builder.addCase(fetchArticle.fulfilled, (state, action) => {
      // The type signature on action.payload matches what we passed into the generic for `normalize`, allowing us to access specific properties on `payload.articles` if desired
      articlesAdapter.upsertMany(state, action.payload.articles)
    })
  },
})
```

Last updated on ***Oct 26, 2022***

Writing Reducers with Immer

Redux Toolkit's `createReducer` and `createSlice` automatically use [Immer](https://immerjs.github.io/immer/) internally to let you write simpler immutable update logic using "mutating" syntax. This helps simplify most reducer implementations.

Because Immer is itself an abstraction layer, it's important to understand why Redux Toolkit uses Immer, and how to use it correctly.

Immutability and Redux

Basics of Immutability

"Mutable" means "changeable". If something is "immutable", it can never be changed.

JavaScript objects and arrays are all mutable by default. If I create an object, I can change the contents of its fields. If I create an array, I can change the contents as well:

```
const obj = { a: 1, b: 2 }
// still the same object outside, but the contents have changed
obj.b = 3

const arr = ['a', 'b']
// In the same way, we can change the contents of this array
arr.push('c')
arr[1] = 'd'
```

This is called *mutating* the object or array. It's the same object or array reference in memory, but now the contents inside the object have changed.

In order to update values immutably, your code must make *copies* of existing objects/arrays, and then modify the copies.

We can do this by hand using JavaScript's array / object spread operators, as well as array methods that return new copies of the array instead of mutating the original array:

```
const obj = {
  a: {
    // To safely update obj.a.c, we have to copy each piece
    c: 3,
  },
  b: 2,
}

const obj2 = {
  // copy obj
  ...obj,
  // overwrite a
  a: {
    // copy obj.a
    ...obj.a,
    // overwrite c
    c: 42,
  },
}

const arr = ['a', 'b']
// Create a new copy of arr, with "c" appended to the end
const arr2 = arr.concat('c')

// or, we can make a copy of the original array:
const arr3 = arr.slice()
// and mutate the copy:
arr3.push('c')
```

Reducers and Immutable Updates

One of the primary rules of Redux is that **our reducers are *never* allowed to mutate the original / current state values!**

```
// ❌ Illegal - by default, this will mutate the state!

state.value = 123
```

There are several reasons why you must not mutate state in Redux:

- It causes bugs, such as the UI not updating properly to show the latest values
- It makes it harder to understand why and how the state has been updated
- It makes it harder to write tests
- It breaks the ability to use "time-travel debugging" correctly
- It goes against the intended spirit and usage patterns for Redux

So if we can't change the originals, how do we return an updated state?

Reducers can only make *copies* of the original values, and then they can mutate the copies.

```
// ✅ This is safe, because we made a copy

return {

  ...state,

  value: 123,

}
```

We already saw that we can write immutable updates by hand, by using JavaScript's array / object spread operators and other functions that return copies of the original values.

This becomes harder when the data is nested. **A critical rule of immutable updates is that you must make a copy of *every* level of nesting that needs to be updated.**

A typical example of this might look like:

```
function handwrittenReducer(state, action) {
  return {
    ...state,
    first: {
      ...state.first,
      second: {
        ...state.first.second,
        [action.someId]: {
          ...state.first.second[action.someId],
          fourth: action.someValue,
        },
      },
    },
  }
}
```

However, if you're thinking that "writing immutable updates by hand this way looks hard to remember and do correctly"... yeah, you're right! :)

Writing immutable update logic by hand *is* hard, and **accidentally mutating state in reducers is the single most common mistake Redux users make**.

Immutable Updates with Immer

Immer is a library that simplifies the process of writing immutable update logic.

Immer provides a function called `produce`, which accepts two arguments: your original `state`, and a callback function. The callback function is given a "draft" version of that state, and inside the callback, it is safe to write code that mutates the draft value. Immer tracks all attempts to mutate the draft value and then replays those mutations using their immutable equivalents to create a safe, immutably updated result:

```
import produce from 'immer'

const baseState = [
  {
    todo: 'Learn typescript',
    done: true,
  },
  {
    todo: 'Try immer',
    done: false,
  },
]
```

```
const nextState = produce(baseState, (draftState) => {
  // "mutate" the draft array
  draftState.push({ todo: 'Tweet about it' })
  // "mutate" the nested state
  draftState[1].done = true
})

console.log(baseState === nextState)
// false - the array was copied
console.log(baseState[0] === nextState[0])
// true - the first item was unchanged, so same reference
console.log(baseState[1] === nextState[1])
// false - the second item was copied and updated
```

Redux Toolkit and Immer

Redux Toolkit's `createReducer` API uses Immer internally automatically. So, it's already safe to "mutate" state inside of any case reducer function that is passed to `createReducer`:

```
const todosReducer = createReducer([], (builder) => {
  builder.addCase('todos/todoAdded', (state, action) => {
    // "mutate" the array by calling push()
    state.push(action.payload)
  })
})
```

In turn, `createSlice` uses `createReducer` inside, so it's also safe to "mutate" state there as well:

```
const todosSlice = createSlice({
  name: 'todos',
  initialState: [],
  reducers: {
    todoAdded(state, action) {
      state.push(action.payload)
    },
  },
})
```

This even applies if the case reducer functions are defined outside of the `createSlice/createReducer` call. For example, you could have a reusable case reducer function that expects to "mutate" its state, and include it as needed:

```
const addItemToArray = (state, action) => {
  state.push(action.payload)
}

const todosSlice = createSlice({
  name: 'todos',
  initialState: [],
  reducers: {
    todoAdded: addItemToArray,
  },
})
```

This works because the "mutating" logic is wrapped in Immer's `produce` method internally when it executes.

Immer Usage Patterns

There are several useful patterns to know about and gotchas to watch out for when using Immer in Redux Toolkit.

Mutating and Returning State

Immer works by tracking attempts to mutate an existing drafted state value, either by assigning to nested fields or by calling functions that mutate the value. That means that **the `state` must be a JS object or array in order for Immer to see the attempted changes**. (You can still have a slice's state be a primitive like a string or a boolean, but since primitives can never be mutated anyway, all you can do is just return a new value.)

In any given case reducer, **Immer expects that you will either *mutate* the existing state, *or* construct a new state value yourself and return it, but *not* both in the same function!** For example, both of these are valid reducers with Immer:

```
const todosSlice = createSlice({
  name: 'todos',
  initialState: [],
  reducers: {
    todoAdded(state, action) {
      // "Mutate" the existing state, no return value needed
      state.push(action.payload)
    },
    todoDeleted(state, action.payload) {
      // Construct a new result array immutably and return it
      return state.filter(todo => todo.id !== action.payload)
    }
  }
})
```

However, it *is* possible to use immutable updates to do part of the work and then save the results via a "mutation". An example of this might be filtering a nested array:

```
const todosSlice = createSlice({
  name: 'todos',
  initialState: {todos: [], status: 'idle'}
  reducers: {
    todoDeleted(state, action.payload) {
      // Construct a new array immutably
      const newTodos = state.todos.filter(todo => todo.id !== action.payload)
      // "Mutate" the existing state to save the new array
      state.todos = newTodos
    }
  }
})
```

Note that **mutating state in an arrow function with an implicit return breaks this rule and causes an error!** This is because statements and function calls may return a value, and Immer sees both the attempted mutation and *and* the new returned value and doesn't know which to use as the result. Some potential solutions are using the `void` keyword to skip having a return value, or using curly braces to give the arrow function a body and no return value:

```
const todosSlice = createSlice({
  name: 'todos',
  initialState: [],
  reducers: {
    // ❌ ERROR: mutates state, but also returns new array size!
    brokenReducer: (state, action) => state.push(action.payload),
    // ✅ SAFE: the `void` keyword prevents a return value
    fixedReducer1: (state, action) => void state.push(action.payload),
    // ✅ SAFE: curly braces make this a function body and no return
    fixedReducer2: (state, action) => {
      state.push(action.payload)
    },
  },
})
```

While writing nested immutable update logic is hard, there are times when it *is* simpler to do an object spread operation to update multiple fields at once, vs assigning individual fields:

```
function objectCaseReducer1(state, action) {
  const { a, b, c, d } = action.payload
  return {
    ...state,
    a,
    b,
    c,
    d,
  }
}
```

```
function objectCaseReducer2(state, action) {
  const { a, b, c, d } = action.payload
  // This works, but we keep having to repeat `state.x =`
  state.a = a
  state.b = b
  state.c = c
  state.d = d
}
```

As an alternative, you can use `Object.assign` to mutate multiple fields at once, since `Object.assign` always mutates the first object that it's given:

```
function objectCaseReducer3(state, action) {
  const { a, b, c, d } = action.payload
  Object.assign(state, { a, b, c, d })
}
```

Resetting and Replacing State

Sometimes you may want to replace the entire existing `state`, either because you've loaded some new data, or you want to reset the state back to its initial value.

DANGER

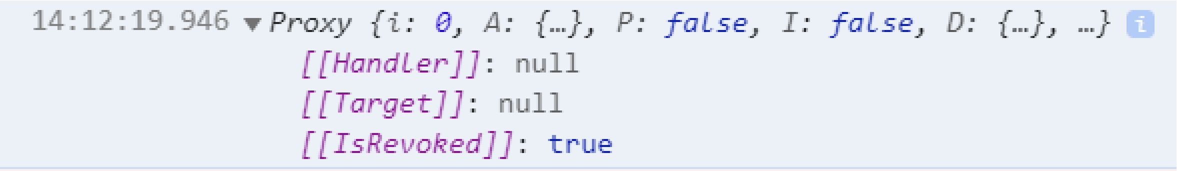
A common mistake is to try assigning `state = someValue` directly. This will not work! This only points the local `state` variable to a different reference. That is neither mutating the existing `state` object/array in memory, nor returning an entirely new value, so Immer does not make any actual changes.

Instead, to replace the existing state, you should return the new value directly:

```
const initialState = []
const todosSlice = createSlice({
  name: 'todos',
  initialState,
  reducers: {
    brokenTodosLoadedReducer(state, action) {
      // ❌ ERROR: does not actually mutate or return anything new!
      state = action.payload
    },
    fixedTodosLoadedReducer(state, action) {
      // ✅ CORRECT: returns a new value to replace the old one
      return action.payload
    },
    correctResetTodosReducer(state, action) {
      // ✅ CORRECT: returns a new value to replace the old one
      return initialState
    },
  },
})
```

Debugging and Inspecting Drafted State

It's common to want to log in-progress state from a reducer to see what it looks like as it's being updated, like `console.log(state)`. Unfortunately, browsers display logged Proxy instances in a format that is hard to read or understand:



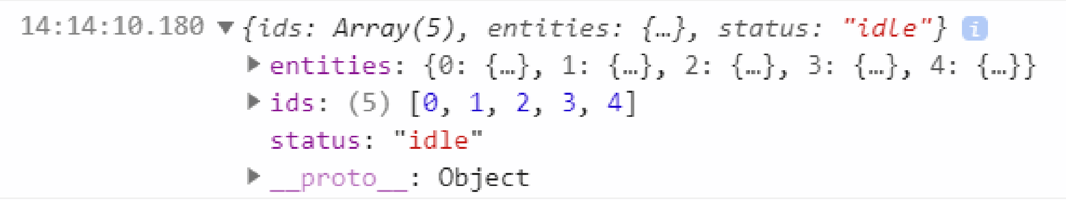
```
14:12:19.946 ▼Proxy {i: 0, A: {…}, P: false, I: false, D: {…}, …}
                 [[Handler]]: null
                 [[Target]]: null
                 [[IsRevoked]]: true
```

To work around this, [Immer includes a `current` function that extracts a copy of the wrapped data](), and RTK re-exports `current`. You can use this in your reducers if you need to log or inspect the work-in-progress state:

```
import { current } from '@reduxjs/toolkit'

const todosSlice = createSlice({
  name: 'todos',
  initialState: todosAdapter.getInitialState(),
  reducers: {
    todoToggled(state, action) {
      // ❌ ERROR: logs the Proxy-wrapped data
      console.log(state)
      // ✅ CORRECT: logs a plain JS copy of the current data
      console.log(current(state))
    },
  },
})
```

The correct output would look like this instead:



```
14:14:10.180 ▼{ids: Array(5), entities: {…}, status: "idle"}
               ▶entities: {0: {…}, 1: {…}, 2: {…}, 3: {…}, 4: {…}}
               ▶ids: (5) [0, 1, 2, 3, 4]
                status: "idle"
               ▶__proto__: Object
```

Immer also provides `original` and `isDraft` functions, which retrieves the original data without any updates applied and check to see if a given value is a Proxy-wrapped draft. As of RTK 1.5.1, both of those are re-exported from RTK as well.

Updating Nested Data

Immer greatly simplifies updating nested data. Nested objects and arrays are also wrapped in Proxies and drafted, and it's safe to pull out a nested value into its own variable and then mutate it.

However, this still only applies to objects and arrays. If we pull out a primitive value into its own variable and try to update it, Immer has nothing to wrap and cannot track any updates:

```
const todosSlice = createSlice({
  name: 'todos',
  initialState: [],
  reducers: {
    brokenTodoToggled(state, action) {
      const todo = state.find((todo) => todo.id === action.payload)
      if (todo) {
        // ❌ ERROR: Immer can't track updates to a primitive value!
        let { completed } = todo
        completed = !completed
      }
    },
    fixedTodoToggled(state, action) {
      const todo = state.find((todo) => todo.id === action.payload)
      if (todo) {
        // ✅ CORRECT: This object is still wrapped in a Proxy, so we can "mutate" it
        todo.completed = !todo.completed
      }
    },
  },
})
```

There *is* a gotcha here. Immer will not wrap objects that are newly inserted into the state. Most of the time this shouldn't matter, but there may be occasions when you want to insert a value and then make further updates to it.

Related to this, RTK's `createEntityAdapter` update functions can either be used as standalone reducers, or "mutating" update functions. These functions determine whether to "mutate" or return a new value by checking to see if the state they're given is wrapped in a draft or not. If you are calling these functions yourself inside of a case reducer, be sure you know whether you're passing them a draft value or a plain value.

Finally, it's worth noting that **Immer does not automatically create nested objects or arrays for you - you have to create them yourself**. As an example, say we have a lookup table containing nested arrays, and we want to insert an item into one of those arrays. If we unconditionally try to insert without checking for the existence of that array, the logic will crash when the array doesn't exist. Instead, you'd need to ensure the array exists first:

```
const itemsSlice = createSlice({
  name: 'items',
  initialState: { a: [], b: [] },
  reducers: {
    brokenNestedItemAdded(state, action) {
      const { id, item } = action.payload
      // ❌ ERROR: will crash if no array exists for `id`!
      state[id].push(item)
    },
    fixedNestedItemAdded(state, action) {
      const { id, item } = action.payload
      // ✅ CORRECT: ensures the nested array always exists first
      if (!state[id]) {
        state[id] = []
      }

      state[id].push(item)
    },
  },
})
```

Linting State Mutations

Many ESLint configs include the https://eslint.org/docs/rules/no-param-reassign rule, which may also warn about mutations to nested fields. That can cause the rule to warn about mutations to `state` in Immer-powered reducers, which is not helpful.

To resolve this, you can tell the ESLint rule to ignore mutations and assignment to a parameter named `state` only in slice files:

```
// @filename .eslintrc.js
module.exports = {
  // add to your ESLint config definition
  overrides: [
    {
      // feel free to replace with your preferred file pattern - eg. 'src/**/*Slice.ts'
      files: ['src/**/*.slice.ts'],
      // avoid state param assignment
      rules: { 'no-param-reassign': ['error', { props: false }] },
    },
  ],
}
```

Why Immer is Built In

We've received a number of requests over time to make Immer an optional part of RTK's `createSlice` and `createReducer` APIs, rather than strictly required.

Our answer is always the same: **Immer *is required* in RTK, and that is not going to change**.

It's worth going over the reasons why we consider Immer to be a critical part of RTK and why we will not make it optional.

Benefits of Immer

Immer has two primary benefits. First, **Immer drastically simplifies immutable update logic**. Proper immutable updates are extremely verbose. Those verbose operations are hard to read overall, and also obfuscate what the actual intent of the update statement is. Immer eliminates all the nested spreads and array slices. Not only is the code shorter and easier to read, it's much more clear what actual update is supposed to happen.

Second, writing immutable updates correctly is *hard*, and it is really easy to make mistakes (like forgetting to copy a level of nesting in a set of object spreads, copying a top-level array and not the item to be updated inside the array, or forgetting that `array.sort()` mutates the array). This is part of why accidental mutations has always been the most common cause of Redux bugs. **Immer effectively *eliminates* accidental mutations**. Not only are there no more spread operations that can be mis-written, but Immer freezes state automatically as well. This causes errors to be thrown if you do accidentally mutate, even outside of a reducer. **Eliminating the #1 cause of Redux bugs is a *huge* improvement.**

Additionally, RTK Query uses Immer's patch capabilities to enable optimistic updates and manual cache updates as well.

Tradeoffs and Concerns

Like any tool, using Immer does have tradeoffs, and users have expressed a number of concerns about using it.

Immer does add to the overall app bundle size. It's about 8K min, 3.3K min+gz (ref: Immer docs: Installation, Bundle.js.org analysis). However, that library bundle size starts to pay for itself by shrinking the amount of reducer logic in your app. Additionally, the benefits of more readable code and eliminating mutation bugs are worth the size.

Immer also adds a bit of overhead in runtime performance. However, per the Immer "Performance" docs page, the overhead is not meaningful in practice. Additionally, reducers are almost never a perf bottleneck in a Redux app anyway. Instead, the cost of updating the UI is much more important.

So, while using Immer isn't "free", the bundle and perf costs are small enough to be worth it.

The most realistic pain point with using Immer is that browser debuggers show Proxies in a confusing way, which makes it hard to inspect state variables while debugging. This is certainly an annoyance. However, this doesn't actually affect runtime behavior, and we've documented the use of `current` to create a viewable plain JS version of the data above in this page. (Given the increasingly wide use of Proxies as part of libraries like Mobx and Vue 3, this is also not unique to Immer.)

Another issue is education and understanding. Redux has always required immutability in reducers, and so seeing "mutating" code can be confusing. It's certainly possible that new Redux users might see those "mutations" in example code, assume that it's normal for Redux usage, and later try to do the same thing outside of `createSlice`. This would indeed cause real mutations and bugs, because it's outside of Immer's ability to wrap the updates.

We've addressed this by repeatedly emphasizing the important of immutability throughout our docs, including multiple highlighted sections emphasizing that the "mutations" only work right thanks to Immer's "magic" inside and adding this specific docs page you're reading now.

Architecture and Intent

There's two more reasons why Immer is not optional.

One is RTK's architecture. `createSlice` and `createReducer` are implemented by directly importing Immer. There's no easy way to create a version of either of them that would have a hypothetical `immer: false` option. You can't do optional imports, and we need Immer available immediately and synchronously during the initial load of the app.

Additionally, RTK currently calls Immer's `enableES5` plugin immediately on import, in order to ensure that Immer works correctly in environments without ES6 Proxy support (such as IE11 and older React Native versions). This is necessary because Immer split out the ES5 behavior into a plugin around version 6.0, but dropping the ES5 support would have been a major breaking change for RTK and broken our users. Because RTK itself calls `enableES5` from the entry point, Immer is *always* pulled in.

And finally: **Immer is built into RTK by default because we believe it is the best choice for our users!** We *want* our users to be using Immer, and consider it to be a critical non-negotiable component of RTK. The great benefits like simpler reducer code and preventing accidental mutations far outweigh the relatively small concerns.

Further Information

See the Immer documentation for more details on Immer's APIs, edge cases, and behavior.

For historical discussion on why Immer is required, see these issues:

- RTK #5: Why Immer inside a starter kit?
- RTK #183: Consider adding an option to remove Immer
- RTK #242: make `immer` optional for `createReducer`

Last updated on ***Dec 27, 2022***

configureStore

A friendly abstraction over the standard Redux `createStore` function that adds good defaults to the store setup for a better development experience.

Parameters

`configureStore` accepts a single configuration object parameter, with the following options:

```
type ConfigureEnhancersCallback = (
  defaultEnhancers: EnhancerArray<[StoreEnhancer]>
) => StoreEnhancer[]

interface ConfigureStoreOptions<
  S = any,
  A extends Action = AnyAction,
  M extends Middlewares<S> = Middlewares<S>
> {
  /**
   * A single reducer function that will be used as the root reducer, or an
   * object of slice reducers that will be passed to `combineReducers()`.
   */
  reducer: Reducer<S, A> | ReducersMapObject<S, A>

  /**
   * An array of Redux middleware to install. If not supplied, defaults to
   * the set of middleware returned by `getDefaultMiddleware()`.
   */
  middleware?: ((getDefaultMiddleware: CurriedGetDefaultMiddleware<S>) => M) | M

  /**
   * Whether to enable Redux DevTools integration. Defaults to `true`.
   *
   * Additional configuration can be done by passing Redux DevTools options
   */
  devTools?: boolean | DevToolsOptions

  /**
   * The initial state, same as Redux's createStore.
   * You may optionally specify it to hydrate the state
   * from the server in universal apps, or to restore a previously serialized
   * user session. If you use `combineReducers()` to produce the root reducer
   * function (either directly or indirectly by passing an object as `reducer`),
   * this must be an object with the same shape as the reducer map keys.
   */
  preloadedState?: DeepPartial<S extends any ? S : S>

  /**
   * The store enhancers to apply. See Redux's `createStore()`.
   * All enhancers will be included before the DevTools Extension enhancer.
   * If you need to customize the order of enhancers, supply a callback
   * function that will receive the original array (ie, `[applyMiddleware]`),
   * and should return a new array (such as `[applyMiddleware, offline]`).
   * If you only need to add middleware, you can use the `middleware` parameter instead.
   */
  enhancers?: StoreEnhancer[] | ConfigureEnhancersCallback
}

function configureStore<S = any, A extends Action = AnyAction>(
  options: ConfigureStoreOptions<S, A>
): EnhancedStore<S, A>
```

`reducer`

If this is a single function, it will be directly used as the root reducer for the store.

If it is an object of slice reducers, like `{users : usersReducer, posts : postsReducer}`, `configureStore` will automatically create the root reducer by passing this object to the Redux `combineReducers` utility.

`middleware`

An optional array of Redux middleware functions

If this option is provided, it should contain all the middleware functions you want added to the store. `configureStore` will automatically pass those to `applyMiddleware`.

If not provided, `configureStore` will call `getDefaultMiddleware` and use the array of middleware functions it returns.

Where you wish to add onto or customize the default middleware, you may pass a callback function that will receive `getDefaultMiddleware` as its argument, and should return a middleware array.

For more details on how the `middleware` parameter works and the list of middleware that are added by default, see the `getDefaultMiddleware` docs page.

`devTools`

If this is a boolean, it will be used to indicate whether `configureStore` should automatically enable support for the Redux DevTools browser extension.

If it is an object, then the DevTools Extension will be enabled, and the options object will be passed to `composeWithDevtools()`. See the DevTools Extension docs for `EnhancerOptions` for a list of the specific options that are available.

Defaults to `true`.

`trace`

The Redux DevTools Extension recently added support for showing action stack traces that show exactly where each action was dispatched. Capturing the traces can add a bit of overhead, so the DevTools Extension allows users to configure whether action stack traces are captured by setting the 'trace' argument. If the DevTools are enabled by passing `true` or an object, then `configureStore` will default to enabling capturing action stack traces in development mode only.

`preloadedState`

An optional initial state value to be passed to the Redux `createStore` function.

`enhancers`

An optional array of Redux store enhancers, or a callback function to customize the array of enhancers.

If defined as an array, these will be passed to the Redux `compose` function, and the combined enhancer will be passed to `createStore`.

This should *not* include `applyMiddleware()` or the Redux DevTools Extension `composeWithDevTools`, as those are already handled by `configureStore`.

Example: `enhancers: [offline]` will result in a final setup of `[applyMiddleware, offline, devToolsExtension]`.

If defined as a callback function, it will be called with the existing array of enhancers *without* the DevTools Extension (currently `[applyMiddleware]`), and should return a new array of enhancers. This is primarily useful for cases where a store enhancer needs to be added in front of `applyMiddleware`, such as `redux-first-router` or `redux-offline`.

Example: `enhancers: (defaultEnhancers) => defaultEnhancers.prepend(offline)` will result in a final setup of `[offline, applyMiddleware, devToolsExtension]`.

Usage

Basic Example

```
import { configureStore } from '@reduxjs/toolkit'

import rootReducer from './reducers'

const store = configureStore({ reducer: rootReducer })
// The store now has redux-thunk added and the Redux DevTools Extension is turned on
```

Full Example

```
// file: todos/todosReducer.ts noEmit
import type { Reducer } from '@reduxjs/toolkit'
declare const reducer: Reducer<{}>
export default reducer

// file: visibility/visibilityReducer.ts noEmit
import type { Reducer } from '@reduxjs/toolkit'
declare const reducer: Reducer<{}>
export default reducer

// file: store.ts
import { configureStore } from '@reduxjs/toolkit'

// We'll use redux-logger just as an example of adding another middleware
import logger from 'redux-logger'

// And use redux-batched-subscribe as an example of adding enhancers
import { batchedSubscribe } from 'redux-batched-subscribe'

import todosReducer from './todos/todosReducer'
import visibilityReducer from './visibility/visibilityReducer'

const reducer = {
  todos: todosReducer,
  visibility: visibilityReducer,
}

const preloadedState = {
  todos: [
    {
      text: 'Eat food',
      completed: true,
    },
    {
      text: 'Exercise',
      completed: false,
    },
  ],
  visibilityFilter: 'SHOW_COMPLETED',
}

const debounceNotify = _.debounce((notify) => notify())

const store = configureStore({
  reducer,
  middleware: (getDefaultMiddleware) => getDefaultMiddleware().concat(logger),
  devTools: process.env.NODE_ENV !== 'production',
  preloadedState,
  enhancers: [batchedSubscribe(debounceNotify)],
})

// The store has been created with these options:
// - The slice reducers were automatically passed to combineReducers()
// - redux-thunk and redux-logger were added as middleware
// - The Redux DevTools Extension is disabled for production
// - The middleware, batched subscribe, and devtools enhancers were composed together
```

Last updated on ***Feb 24, 2023***

`getDefaultMiddleware`

Returns an array containing the default list of middleware.

Intended Usage

By default, [`configureStore`]() adds some middleware to the Redux store setup automatically.

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

// Store has middleware added, because the middleware list was not customized
```

If you want to customize the list of middleware, you can supply an array of middleware functions to `configureStore`:

```
const store = configureStore({
  reducer: rootReducer,
  middleware: [thunk, logger],
})

// Store specifically has the thunk and logger middleware applied
```

However, when you supply the `middleware` option, you are responsible for defining *all* the middleware you want added to the store. `configureStore` will not add any extra middleware beyond what you listed.

`getDefaultMiddleware` is useful if you want to add some custom middleware, but also still want to have the default middleware added as well:

```
import { configureStore } from '@reduxjs/toolkit'

import logger from 'redux-logger'

import rootReducer from './reducer'

const store = configureStore({
  reducer: rootReducer,
  middleware: (getDefaultMiddleware) => getDefaultMiddleware().concat(logger),
})

// Store has all of the default middleware added, _plus_ the logger middleware
```

It is preferable to use the chainable `.concat(...)` and `.prepend(...)` methods of the returned `MiddlewareArray` instead of the array spread operator, as the latter can lose valuable type information under some circumstances.

Included Default Middleware

Development

One of the goals of Redux Toolkit is to provide opinionated defaults and prevent common mistakes. As part of that, `getDefaultMiddleware` includes some middleware that are added **in development builds of your app only** to provide runtime checks for three common issues:

- Immutability check middleware: deeply compares state values for mutations. It can detect mutations in reducers during a dispatch, and also mutations that occur between dispatches (such as in a component or a selector). When a mutation is detected, it will throw an error and indicate the key path for where the mutated value was detected in the state tree. (Forked from `redux-immutable-state-invariant`.)
- Serializability check middleware: a custom middleware created specifically for use in Redux Toolkit. Similar in concept to `immutable-state-invariant`, but deeply checks your state tree and your actions for non-serializable values such as functions, Promises, Symbols, and other non-plain-JS-data values. When a non-serializable value is detected, a console error will be printed with the key path for where the non-serializable value was detected.
- Action creator check middleware: another custom middleware created specifically for use in Redux Toolkit. Identifies when an action creator was mistakenly dispatched without being called, and warns to console with the action type.

In addition to these development tool middleware, it also adds `redux-thunk` by default, since thunks are the basic recommended side effects middleware for Redux.

Currently, the return value is:

```
const middleware = [
  actionCreatorInvariant,
  immutableStateInvariant,
  thunk,
  serializableStateInvariant,
]
```

Production

Currently, the return value is:

```
const middleware = [thunk]
```

Customizing the Included Middleware

`getDefaultMiddleware` accepts an options object that allows customizing each middleware in two ways:

- Each middleware can be excluded the result array by passing `false` for its corresponding field
- Each middleware can have its options customized by passing the matching options object for its corresponding field

This example shows excluding the serializable state check middleware, and passing a specific value for the thunk middleware's "extra argument":

```
import { configureStore } from '@reduxjs/toolkit'
import rootReducer from './reducer'
import { myCustomApiService } from './api'

const store = configureStore({
  reducer: rootReducer,
  middleware: (getDefaultMiddleware) =>
    getDefaultMiddleware({
      thunk: {
        extraArgument: myCustomApiService,
      },
      serializableCheck: false,
    }),
})
```

API Reference

```
interface ThunkOptions<E = any> {
  extraArgument: E
}

interface ImmutableStateInvariantMiddlewareOptions {
  // See "Immutability Middleware" page for definition
}

interface SerializableStateInvariantMiddlewareOptions {
  // See "Serializability Middleware" page for definition
}

interface ActionCreatorInvariantMiddlewareOptions {
  // See "Action Creator Middleware" page for definition
}

interface GetDefaultMiddlewareOptions {
  thunk?: boolean | ThunkOptions
  immutableCheck?: boolean | ImmutableStateInvariantMiddlewareOptions
  serializableCheck?: boolean | SerializableStateInvariantMiddlewareOptions
  actionCreatorCheck?: boolean | ActionCreatorInvariantMiddlewareOptions
}

function getDefaultMiddleware<S = any>(
  options: GetDefaultMiddlewareOptions = {}
): Middleware<{}, S>[]
```

Last updated on ***May 2, 2023***

Immutability Middleware

A port of the `redux-immutable-state-invariant` middleware, customized for use with Redux Toolkit. Any detected mutations will be thrown as errors.

This middleware is added to the store by default by `configureStore` and `getDefaultMiddleware`.

You can customize the behavior of this middleware by passing any of the supported options as the `immutableCheck` value for `getDefaultMiddleware`.

Options

```
type IsImmutableFunc = (value: any) => boolean

interface ImmutableStateInvariantMiddlewareOptions {
  /**
    Callback function to check if a value is considered to be immutable.
    This function is applied recursively to every value contained in the state.
    The default implementation will return true for primitive types
    (like numbers, strings, booleans, null and undefined).
   */
  isImmutable?: IsImmutableFunc
  /**
    An array of dot-separated path strings or RegExps that match named nodes from
    the root state to ignore when checking for immutability.
    Defaults to undefined
   */
  ignoredPaths?: (string | RegExp)[]
  /** Print a warning if checks take longer than N ms. Default: 32ms */
  warnAfter?: number
  // @deprecated. Use ignoredPaths
  ignore?: string[]
}
```

Exports

`createImmutableStateInvariantMiddleware`

Creates an instance of the immutability check middleware, with the given options.

You will most likely not need to call this yourself, as `getDefaultMiddleware` already does so.

Example:

```
// file: exampleSlice.ts
import { createSlice } from '@reduxjs/toolkit'

export const exampleSlice = createSlice({
  name: 'example',
  initialState: {
    user: 'will track changes',
    ignoredPath: 'single level',
    ignoredNested: {
      one: 'one',
      two: 'two',
    },
  },
  reducers: {},
})

export default exampleSlice.reducer


// file: store.ts
import {
  configureStore,
  createImmutableStateInvariantMiddleware,
} from '@reduxjs/toolkit'

import exampleSliceReducer from './exampleSlice'

const immutableInvariantMiddleware = createImmutableStateInvariantMiddleware({
  ignoredPaths: ['ignoredPath', 'ignoredNested.one', 'ignoredNested.two'],
})

const store = configureStore({
  reducer: exampleSliceReducer,
  // Note that this will replace all default middleware
  middleware: [immutableInvariantMiddleware],
})
```

doing the same without removing all other middlewares, using getDetfaultMiddleware:

```
import { configureStore } from '@reduxjs/toolkit'

import exampleSliceReducer from './exampleSlice'

const store = configureStore({
  reducer: exampleSliceReducer,
  // This replaces the original default middleware with the customized versions
  middleware: (getDefaultMiddleware) =>
    getDefaultMiddleware({
      immutableCheck: {
        ignoredPaths: ['ignoredPath', 'ignoredNested.one', 'ignoredNested.two'],
      },
    }),
})
```

`isImmutableDefault`

Default implementation of the "is this value immutable?" check. Currently implemented as:

```
return ( typeof value !== 'object' || value === null || typeof value === 'undefined')
```

This will return true for primitive types (like numbers, strings, booleans, null and undefined)

*Last updated on **Jan 28, 2023***

Serializability Middleware

A custom middleware that detects if any non-serializable values have been included in state or dispatched actions, modeled after `redux-immutable-state-invariant`. Any detected non-serializable values will be logged to the console.

This middleware is added to the store by default by [`configureStore`]() and [`getDefaultMiddleware`]().

You can customize the behavior of this middleware by passing any of the supported options as the `serializableCheck` value for `getDefaultMiddleware`.

Options

```
interface SerializableStateInvariantMiddlewareOptions {
  /**
   * The function to check if a value is considered serializable. This
   * function is applied recursively to every value contained in the
   * state. Defaults to `isPlain()`.
   */
  isSerializable?: (value: any) => boolean
  /**
   * The function that will be used to retrieve entries from each
   * value.  If unspecified, `Object.entries` will be used. Defaults
   * to `undefined`.
   */
  getEntries?: (value: any) => [string, any][]

  /**
   * An array of action types to ignore when checking for serializability.
   * Defaults to []
   */
  ignoredActions?: string[]

  /**
   * An array of dot-separated path strings or regular expressions to ignore
   * when checking for serializability, Defaults to
   * ['meta.arg', 'meta.baseQueryMeta']
   */
  ignoredActionPaths?: (string | RegExp)[]

  /**
   * An array of dot-separated path strings or regular expressions to ignore
   * when checking for serializability, Defaults to []
   */
  ignoredPaths?: (string | RegExp)[]
  /**
   * Execution time warning threshold. If the middleware takes longer
   * than `warnAfter` ms, a warning will be displayed in the console.
   * Defaults to 32ms.
   */
  warnAfter?: number

  /**
   * Opt out of checking state. When set to `true`, other state-related params will be ignored.
   */
  ignoreState?: boolean

  /**
   * Opt out of checking actions. When set to `true`, other action-related params will be ignored.
   */
  ignoreActions?: boolean
}
```

Exports

`createSerializableStateInvariantMiddleware`

Creates an instance of the serializability check middleware, with the given options.

You will most likely not need to call this yourself, as `getDefaultMiddleware` already does so.

Example:

```
import { Iterable } from 'immutable'
import {
  configureStore,
  createSerializableStateInvariantMiddleware,
  isPlain,
} from '@reduxjs/toolkit'
import reducer from './reducer'

// Augment middleware to consider Immutable.JS iterables serializable
const isSerializable = (value: any) =>
  Iterable.isIterable(value) || isPlain(value)

const getEntries = (value: any) =>
  Iterable.isIterable(value) ? value.entries() : Object.entries(value)

const serializableMiddleware = createSerializableStateInvariantMiddleware({
  isSerializable,
  getEntries,
})

const store = configureStore({
  reducer,
  middleware: [serializableMiddleware],
})
```

`isPlain`

Checks whether the given value is considered a "plain value" or not.

Currently implemented as:

```
import isPlainObject from './isPlainObject'

export function isPlain(val: any) {
  return (
    typeof val === 'undefined' ||
    val === null ||
    typeof val === 'string' ||
    typeof val === 'boolean' ||
    typeof val === 'number' ||
    Array.isArray(val) ||
    isPlainObject(val)
  )
}
```

This will accept all standard JS objects, arrays, and primitives, but return false for `Date`s, `Map`s, and other similar class instances.

*Last updated on **Jan 28, 2023***

Action Creator Middleware

A custom middleware that detects if an action creator has been mistakenly dispatched, instead of being called before dispatching.

A common mistake is to call `dispatch(actionCreator)` instead of `dispatch(actionCreator())`. This tends to "work" as the action creator has the static `type` property, but can lead to unexpected behaviour.

Options

```
export interface ActionCreatorInvariantMiddlewareOptions {
  /**
   * The function to identify whether a value is an action creator.
   * The default checks for a function with a static type property and match method.
   */
  isActionCreator?: (action: unknown) => action is Function & { type?: unknown }
}
```

Exports

`createActionCreatorInvariantMiddleware`

Creates an instance of the action creator check middleware, with the given options.

You will most likely not need to call this yourself, as `getDefaultMiddleware` already does so. Example:

```
import {
  configureStore,
  createActionCreatorInvariantMiddleware,
} from '@reduxjs/toolkit'
import reducer from './reducer'

// Augment middleware to consider all functions with a static type property to be action creators
const isActionCreator = (
  action: unknown
): action is Function & { type: unknown } =>
  typeof action === 'function' && 'type' in action

const actionCreatorMiddleware = createActionCreatorInvariantMiddleware({
  isActionCreator,
})

const store = configureStore({
  reducer,
  middleware: [actionCreatorMiddleware],
})
```

*Last updated on **May 2, 2023***

`createListenerMiddleware`

Overview

A Redux middleware that lets you define "listener" entries that contain an "effect" callback with additional logic, and a way to specify when that callback should run based on dispatched actions or state changes.

It's intended to be a lightweight alternative to more widely used Redux async middleware like sagas and observables. While similar to thunks in level of complexity and concept, it can be used to replicate some common saga usage patterns.

Conceptually, you can think of this as being similar to React's `useEffect` hook, except that it runs logic in response to Redux store updates instead of component props/state updates.

Listener effect callbacks have access to `dispatch` and `getState`, similar to thunks. The listener also receives a set of async workflow functions like `take`, `condition`, `pause`, `fork`, and `unsubscribe`, which allow writing more complex async logic.

Listeners can be defined statically by calling `listenerMiddleware.startListening()` during setup, or added and removed dynamically at runtime with special `dispatch(addListener())` and `dispatch(removeListener())` actions.

Basic Usage

```
import { configureStore, createListenerMiddleware } from '@reduxjs/toolkit'

import todosReducer, {
  todoAdded,
  todoToggled,
  todoDeleted,
} from '../features/todos/todosSlice'

// Create the middleware instance and methods
const listenerMiddleware = createListenerMiddleware()

// Add one or more listener entries that look for specific actions.
// They may contain any sync or async logic, similar to thunks.
listenerMiddleware.startListening({
  actionCreator: todoAdded,
  effect: async (action, listenerApi) => {
    // Run whatever additional side-effect-y logic you want here
    console.log('Todo added: ', action.payload.text)

    // Can cancel other running instances
    listenerApi.cancelActiveListeners()

    // Run async logic
    const data = await fetchData()

    // Pause until action dispatched or state changed
    if (await listenerApi.condition(matchSomeAction)) {
      // Use the listener API methods to dispatch, get state,
      // unsubscribe the listener, start child tasks, and more
      listenerApi.dispatch(todoAdded('Buy pet food'))
      // Spawn "child tasks" that can do more work and return results
      const task = listenerApi.fork(async (forkApi) => {
        // Can pause execution
        await forkApi.delay(5)
        // Complete the child by returning a value
        return 42
      })

      const result = await task.result
      // Unwrap the child result in the listener
      if (result.status === 'ok') {
        // Logs the `42` result value that was returned
        console.log('Child succeeded: ', result.value)
      }
    }
  },
})

const store = configureStore({
  reducer: {
    todos: todosReducer,
  },
  // Add the listener middleware to the store.
  // NOTE: Since this can receive actions with functions inside,
  // it should go before the serializability check middleware
  middleware: (getDefaultMiddleware) =>
    getDefaultMiddleware().prepend(listenerMiddleware.middleware),
})
```

`createListenerMiddleware`

Creates an instance of the middleware, which should then be added to the store via `configureStore`'s `middleware` parameter.

```
const createListenerMiddleware = (options?: CreateMiddlewareOptions) =>
  ListenerMiddlewareInstance

interface CreateListenerMiddlewareOptions<ExtraArgument = unknown> {
  extra?: ExtraArgument
  onError?: ListenerErrorHandler
}

type ListenerErrorHandler = (
  error: unknown,
  errorInfo: ListenerErrorInfo
) => void

interface ListenerErrorInfo {
  raisedBy: 'effect' | 'predicate'
}
```

Middleware Options

- `extra`: an optional "extra argument" that will be injected into the `listenerApi` parameter of each listener. Equivalent to the "extra argument" in the Redux Thunk middleware
- `onError`: an optional error handler that gets called with synchronous and async errors raised by `listener` and synchronous errors thrown by `predicate`.

Listener Middleware Instance

The "listener middleware instance" returned from `createListenerMiddleware` is an object similar to the "slice" objects generated by `createSlice`. The instance object is *not* the actual Redux middleware itself. Rather, it contains the middleware and some instance methods used to add and remove listener entries within the middleware.

```
interface ListenerMiddlewareInstance<
  State = unknown,
  Dispatch extends ThunkDispatch<State, unknown, AnyAction> = ThunkDispatch<
    State,
    unknown,
    AnyAction
  >,
  ExtraArgument = unknown
> {
  middleware: ListenerMiddleware<State, Dispatch, ExtraArgument>
  startListening: (options: AddListenerOptions) => Unsubscribe
  stopListening: (
    options: AddListenerOptions & UnsubscribeListenerOptions
  ) => boolean
  clearListeners: () => void
}
```

`middleware`

The actual Redux middleware. Add this to the Redux store via the `configureStore.middleware` option.

Since the listener middleware can receive "add" and "remove" actions containing functions, this should normally be added as the first middleware in the chain so that it is before the serializability check middleware.

```
const store = configureStore({
  reducer: {
    todos: todosReducer,
  },
  // Add the listener middleware to the store.
  // NOTE: Since this can receive actions with functions inside,
  // it should go before the serializability check middleware
  middleware: (getDefaultMiddleware) =>
    getDefaultMiddleware().prepend(listenerMiddleware.middleware),
})
```

startListening

Adds a new listener entry to the middleware. Typically used to "statically" add new listeners during application setup.

```
const startListening = (options: AddListenerOptions) => UnsubscribeListener

interface AddListenerOptions {
  // Four options for deciding when the listener will run:

  // 1) Exact action type string match
  type?: string

  // 2) Exact action type match based on the RTK action creator
  actionCreator?: ActionCreator

  // 3) Match one of many actions using an RTK matcher
  matcher?: Matcher

  // 4) Return true based on a combination of action + state
  predicate?: ListenerPredicate

  // The actual callback to run when the action is matched
  effect: (action: Action, listenerApi: ListenerApi) => void | Promise<void>
}

type ListenerPredicate<Action extends AnyAction, State> = (
  action: Action,
  currentState?: State,
  originalState?: State
) => boolean

type UnsubscribeListener = (
  unsubscribeOptions?: UnsubscribeListenerOptions
) => void

interface UnsubscribeListenerOptions {
  cancelActive?: true
}
```

You must provide exactly *one* of the four options for deciding when the listener will run: `type`, `actionCreator`, `matcher`, or `predicate`. Every time an action is dispatched, each listener will be checked to see if it should run based on the current action vs the comparison option provided.

These are all acceptable:

```
// 1) Action type string
listenerMiddleware.startListening({ type: 'todos/todoAdded', effect })
// 2) RTK action creator
listenerMiddleware.startListening({ actionCreator: todoAdded, effect })
// 3) RTK matcher function
listenerMiddleware.startListening({
  matcher: isAnyOf(todoAdded, todoToggled),
  effect,})
```

```
// 4) Listener predicate
listenerMiddleware.startListening({
  predicate: (action, currentState, previousState) => {
    // return true when the listener should run
  },
  effect,
})
```

Note that the `predicate` option actually allows matching solely against state-related checks, such as "did `state.x` change" or "the current value of `state.x` matches some criteria", regardless of the actual action.

The "matcher" utility functions included in RTK are acceptable as either the `matcher` or `predicate` option.

The return value is an `unsubscribe()` callback that will remove this listener. By default, unsubscribing will *not* cancel any active instances of the listener. However, you may also pass in `{cancelActive: true}` to cancel running instances.

If you try to add a listener entry but another entry with this exact function reference already exists, no new entry will be added, and the existing `unsubscribe` method will be returned.

The `effect` callback will receive the current action as its first argument, as well as a "listener API" object similar to the "thunk API" object in `createAsyncThunk`.

All listener predicates and callbacks are checked *after* the root reducer has already processed the action and updated the state.
The `listenerApi.getOriginalState()` method can be used to get the state value that existed before the action that triggered this listener was processed.

`stopListening`

Removes a given listener entry.

It accepts the same arguments as `startListening()`. It checks for an existing listener entry by comparing the function references of `listener` and the provided `actionCreator/matcher/predicate` function or `type` string.

By default, this does *not* cancel any active running instances. However, you may also pass in `{cancelActive: true}` to cancel running instances.

```
const stopListening = (
  options: AddListenerOptions & UnsubscribeListenerOptions
) => boolean

interface UnsubscribeListenerOptions {
  cancelActive?: true
}
```

Returns `true` if the listener entry has been removed, or `false` if no subscription matching the input provided has been found.

```
// Examples:
// 1) Action type string
listenerMiddleware.stopListening({
  type: 'todos/todoAdded',
  listener,
  cancelActive: true,
})
// 2) RTK action creator
listenerMiddleware.stopListening({ actionCreator: todoAdded, effect })
// 3) RTK matcher function
listenerMiddleware.stopListening({ matcher, effect, cancelActive: true })
// 4) Listener predicate
listenerMiddleware.stopListening({ predicate, effect })
```

`clearListeners`

Removes all current listener entries. It also cancels all active running instances of those listeners as well.

This is most likely useful for test scenarios where a single middleware or store instance might be used in multiple tests, as well as some app cleanup situations.

```
const clearListeners = () => void;
```

Action Creators

In addition to adding and removing listeners by directly calling methods on the listener instance, you can dynamically add and remove listeners at runtime by dispatching special "add" and "remove" actions. These are exported from the main RTK package as standard RTK-generated action creators.

`addListener`

A standard RTK action creator, imported from the package. Dispatching this action tells the middleware to dynamically add a new listener at runtime. It accepts exactly the same options as `startListening()`

Dispatching this action returns an `unsubscribe()` callback from `dispatch`.

```
// Per above, provide `predicate` or any of the other comparison options
const unsubscribe = store.dispatch(addListener({ predicate, effect }))
```

`removeListener`

A standard RTK action creator, imported from the package. Dispatching this action tells the middleware to dynamically remove a listener at runtime. Accepts the same arguments as `stopListening()`.

By default, this does *not* cancel any active running instances. However, you may also pass in `{cancelActive: true}` to cancel running instances.

Returns `true` if the listener entry has been removed, `false` if no subscription matching the input provided has been found.

```
const wasRemoved = store.dispatch(
  removeListener({ predicate, effect, cancelActive: true })
)
```

`clearAllListeners`

A standard RTK action creator, imported from the package. Dispatching this action tells the middleware to remove all current listener entries. It also cancels all active running instances of those listeners as well.

```
store.dispatch(clearAllListeners())
```

Listener API

The `listenerApi` object is the second argument to each listener callback. It contains several utility functions that may be called anywhere inside the listener's logic.

```
export interface ListenerEffectAPI<
  State,
  Dispatch extends ReduxDispatch<AnyAction>,
  ExtraArgument = unknown
> extends MiddlewareAPI<Dispatch, State> {
  // NOTE: MiddlewareAPI contains `dispatch` and `getState` already

  /**
   * Returns the store state as it existed when the action was originally dispatched, _before_ the reducers ran.
   * This function can **only** be invoked **synchronously**, it throws error otherwise.
   */
  getOriginalState: () => State
  /**
   * Removes the listener entry from the middleware and prevent future instances of the listener from running.
   * It does **not** cancel any active instances.
   */
  unsubscribe(): void
  /**
   * It will subscribe a listener if it was previously removed, noop otherwise.
   */
  subscribe(): void
  /**
   * Returns a promise that resolves when the input predicate returns `true` or
   * rejects if the listener has been cancelled or is completed.
   *
   * The return value is `true` if the predicate succeeds or `false` if a timeout is provided and expires first.
   */
  condition: ConditionFunction<State>
  /**
   * Returns a promise that resolves when the input predicate returns `true` or
   * rejects if the listener has been cancelled or is completed.
   *
   * The return value is the `[action, currentState, previousState]` combination that the predicate saw as arguments.
   *
   * The promise resolves to null if a timeout is provided and expires first.
   */
  take: TakePattern<State>
  /**
   * Cancels all other running instances of this same listener except for the one that made this call.
   */
  cancelActiveListeners: () => void
  /**
   * An abort signal whose `aborted` property is set to `true`
   * if the listener execution is either aborted or completed.
   * @see https://developer.mozilla.org/en-US/docs/Web/API/AbortSignal
   */
  signal: AbortSignal
  /**
   * Returns a promise that resolves after `timeoutMs` or
   * rejects if the listener has been cancelled or is completed.
   */
  delay(timeoutMs: number): Promise<void>
  /**
   * Queues in the next microtask the execution of a task.
   */
  fork<T>(executor: ForkedTaskExecutor<T>): ForkedTask<T>
  /**
   * Returns a promise that resolves when `waitFor` resolves or
   * rejects if the listener has been cancelled or is completed.
   * @param promise
   */
  pause<M>(promise: Promise<M>): Promise<M>
  extra: ExtraArgument
}
```

These can be divided into several categories.

Store Interaction Methods

- `dispatch: Dispatch`: the standard `store.dispatch` method
- `getState: () => State`: the standard `store.getState` method

- `getOriginalState: () => State`: returns the store state as it existed when the action was originally dispatched, *before* the reducers ran. (**Note**: this method can only be called synchronously, during the initial dispatch call stack, to avoid memory leaks. Calling it asynchronously will throw an error.)
- `extra: unknown`: the "extra argument" that was provided as part of the middleware setup, if any

`dispatch` and `getState` are exactly the same as in a thunk. `getOriginalState` can be used to compare the original state before the listener was started.

`extra` can be used to inject a value such as an API service layer into the middleware at creation time, and is accessible here.

Listener Subscription Management

- `unsubscribe: () => void`: removes the listener entry from the middleware, and prevent future instances of the listener from running. (This does *not* cancel any active instances.)
- `subscribe: () => void`: will re-subscribe the listener entry if it was previously removed, or no-op if currently subscribed
- `cancelActiveListeners: () => void`: cancels all other running instances of this same listener *except* for the one that made this call. (The cancellation will only have a meaningful effect if the other instances are paused using one of the cancellation-aware APIs like `take/cancel/pause/delay` - see "Cancelation and Task Management" in the "Usage" section for more details)
- `signal: AbortSignal`: An `AbortSignal` whose `aborted` property will be set to `true` if the listener execution is aborted or completed.

Dynamically unsubscribing and re-subscribing this listener allows for more complex async workflows, such as avoiding duplicate running instances by calling `listenerApi.unsubscribe()` at the start of a listener, or calling `listenerApi.cancelActiveListeners()` to ensure that only the most recent instance is allowed to complete.

Conditional Workflow Execution

- `take: (predicate: ListenerPredicate, timeout?: number) => Promise<[Action, State, State] | null>`: returns a promise that will resolve when the `predicate` returns `true`. The return value is the `[action, currentState, previousState]` combination that the predicate saw as arguments. If a `timeout` is provided and expires first, the promise resolves to `null`.
- `condition: (predicate: ListenerPredicate, timeout?: number) => Promise<boolean>`: Similar to `take`, but resolves to `true` if the predicate succeeds, and `false` if a `timeout` is provided and expires first. This allows async logic to pause and wait for some condition to occur before continuing. See "Writing Async Workflows" below for details on usage.
- `delay: (timeoutMs: number) => Promise<void>`: returns a cancellation-aware promise that resolves after the timeout, or rejects if cancelled before the expiration

- `pause: (promise: Promise<T>) => Promise<T>`: accepts any promise, and returns a cancellation-aware promise that either resolves with the argument promise or rejects if cancelled before the resolution

These methods provide the ability to write conditional logic based on future dispatched actions and state changes. Both also accept an optional `timeout` in milliseconds.

`take` resolves to a `[action, currentState, previousState]` tuple or `null` if it timed out, whereas `condition` resolves to `true` if it succeeded or `false` if timed out.

`take` is meant for "wait for an action and get its contents", while `condition` is meant for checks like `if (await condition(predicate))`.

Both these methods are cancellation-aware, and will throw a `TaskAbortError` if the listener instance is cancelled while paused.

Note that both `take` and `condition` will only resolve **after the next action** has been dispatched. They do not resolve immediately even if their predicate would return true for the current state.

Child Tasks

- `fork: (executor: (forkApi: ForkApi) => T | Promise<T>) => ForkedTask<T>`: Launches a "child task" that may be used to accomplish additional work. Accepts any sync or async function as its argument, and returns a `{result, cancel}` object that can be used to check the final status and return value of the child task, or cancel it while in-progress.

Child tasks can be launched, and waited on to collect their return values. The provided `executor` function will be called asynchronously with a `forkApi` object containing `{pause, delay, signal}`, allowing it to pause or check cancellation status. It can also make use of the `listenerApi` from the listener's scope.

An example of this might be a listener that forks a child task containing an infinite loop that listens for events from a server. The parent then uses `listenerApi.condition()` to wait for a "stop" action, and cancels the child task.

The task and result types are:

```
interface ForkedTaskAPI {
  pause<W>(waitFor: Promise<W>): Promise<W>
  delay(timeoutMs: number): Promise<void>
  signal: AbortSignal
}

export type TaskResolved<T> = {
  readonly status: 'ok'
  readonly value: T
}
```

```
export type TaskRejected = {
  readonly status: 'rejected'
  readonly error: unknown
}

export type TaskCancelled = {
  readonly status: 'cancelled'
  readonly error: TaskAbortError
}

export type TaskResult<Value> =
  | TaskResolved<Value>
  | TaskRejected
  | TaskCancelled

export interface ForkedTask<T> {
  result: Promise<TaskResult<T>>
  cancel(): void
}
```

TypeScript Usage

The middleware code is fully TS-typed. However, the `startListening` and `addListener` functions do not know what the store's `RootState` type looks like by default, so `getState()` will return `unknown`.

To fix this, the middleware provides types for defining "pre-typed" versions of those methods, similar to the pattern used for defing pre-typed React-Redux hooks. We specifically recommend creating the middleware instance in a separate file from the actual `configureStore()` call:

```
// listenerMiddleware.ts
import { createListenerMiddleware, addListener } from '@reduxjs/toolkit'
import type { TypedStartListening, TypedAddListener } from '@reduxjs/toolkit'

import type { RootState, AppDispatch } from './store'

export const listenerMiddleware = createListenerMiddleware()

export type AppStartListening = TypedStartListening<RootState, AppDispatch>

export const startAppListening =
  listenerMiddleware.startListening as AppStartListening

export const addAppListener = addListener as TypedAddListener<
  RootState,
  AppDispatch
>
```

Then import and use those pre-typed methods in your components.

Usage Guide

Overall Purpose

This middleware lets you run additional logic when some action is dispatched, as a lighter-weight alternative to middleware like sagas and observables that have both a heavy runtime bundle cost and a large conceptual overhead.

This middleware is not intended to handle all possible use cases. Like thunks, it provides you with a basic set of primitives (including access to `dispatch` and `getState`), and gives you freedom to write any sync or async logic you want. This is both a strength (you can do anything!) and a weakness (you can do anything, with no guard rails!).

The middleware includes several async workflow primitives that are sufficient to write equivalents to many Redux-Saga effects operators like `takeLatest`, `takeLeading`, and `debounce`, although none of those methods are directly included. (See the listener middleware tests file for examples of how to write code equivalent to those effects.)

Standard Usage Patterns

The most common expected usage is "run some logic after a given action was dispatched". For example, you could set up a simple analytics tracker by looking for certain actions and sending extracted data to the server, including pulling user details from the store:

```
listenerMiddleware.startListening({
  matcher: isAnyOf(action1, action2, action3),
  effect: (action, listenerApi) => {
    const user = selectUserDetails(listenerApi.getState())

    const { specialData } = action.meta

    analyticsApi.trackUsage(action.type, user, specialData)
  },
})
```

However, the `predicate` option also allows triggering logic when some state value has changed, or when the state matches a particular condition:

```
listenerMiddleware.startListening({
  predicate: (action, currentState, previousState) => {
    // Trigger logic whenever this field changes
    return currentState.counter.value !== previousState.counter.value
  },
  effect,
})

listenerMiddleware.startListening({
  predicate: (action, currentState, previousState) => {
    // Trigger logic after every action if this condition is true
    return currentState.counter.value > 3
  },
  effect,})
```

You could also implement a generic API fetching capability, where the UI dispatches a plain action describing the type of resource to be requested, and the middleware automatically fetches it and dispatches a result action:

```
listenerMiddleware.startListening({
  actionCreator: resourceRequested,
  effect: async (action, listenerApi) => {
    const { name, args } = action.payload
    listenerApi.dispatch(resourceLoading())

    const res = await serverApi.fetch(`/api/${name}`, ...args)
    listenerApi.dispatch(resourceLoaded(res.data))
  },
})
```

(That said, we would recommend use of RTK Query for any meaningful data fetching behavior - this is primarily an example of what you *could* do in a listener.)

The `listenerApi.unsubscribe` method may be used at any time, and will remove the listener from handling any future actions. As an example, you could create a one-shot listener by unconditionally calling `unsubscribe()` in the body - the effect callback would run the first time the relevant action is seen, then immediately unsubscribe and never run again. (The middleware actually uses this technique internally for the `take/condition` methods)

Writing Async Workflows with Conditions

One of the great strengths of both sagas and observables is their support for complex async workflows, including stopping and starting behavior based on specific dispatched actions. However, the weakness is that both require mastering a complex API with many unique operators (effects methods like `call()` and `fork()` for sagas, RxJS operators for observables), and both add a significant amount to application bundle size.

While the listener middleware is *not* meant to fully replace sagas or observables, it does provide a carefully chosen set of APIs to implement long-running async workflows as well.

Listeners can use the `condition` and `take` methods in `listenerApi` to wait until some action is dispatched or state check is met. The `condition` method is directly inspired by the `condition` function in Temporal.io's workflow API (credit to @swyx for the suggestion!), and `take` is inspired by the `take` effect from Redux-Saga.

The signatures are:

```
type ConditionFunction<Action extends AnyAction, State> = (
  predicate: ListenerPredicate<Action, State> | (() => boolean),
  timeout?: number
) => Promise<boolean>

type TakeFunction<Action extends AnyAction, State> = (
  predicate: ListenerPredicate<Action, State> | (() => boolean),
  timeout?: number
) => Promise<[Action, State, State] | null>
```

You can use `await condition(somePredicate)` as a way to pause execution of your listener callback until some criteria is met.

The `predicate` will be called after every action is processed by the reducers, and should return `true` when the condition should resolve. (It is effectively a one-shot listener itself.) If a `timeout` number (in ms) is provided, the promise will resolve `true` if the `predicate` returns first, or `false` if the timeout expires. This allows you to write comparisons like `if (await condition(predicate, timeout))`.

This should enable writing longer-running workflows with more complex async logic, such as the "cancellable counter" example from Redux-Saga.

An example of `condition` usage, from the test suite:

```
test('condition method resolves promise when there is a timeout', async () => {
  let finalCount = 0
  let listenerStarted = false

  listenerMiddleware.startListening({
    predicate: (action, currentState: CounterState) => {
      return increment.match(action) && currentState.value === 0
    },
    effect: async (action, listenerApi) => {
      listenerStarted = true
      // Wait for either the counter to hit 3, or 50ms to elapse
      const result = await listenerApi.condition(
        (action, currentState: CounterState) => {
          return currentState.value === 3
        },
        50
      )

      // In this test, we expect the timeout to happen first
      expect(result).toBe(false)
      // Save the state for comparison outside the listener
      const latestState = listenerApi.getState()
      finalCount = latestState.value
    },
  })

  store.dispatch(increment())
  // The listener should have started right away
  expect(listenerStarted).toBe(true)

  store.dispatch(increment())

  // If we wait 150ms, the condition timeout will expire first
  await delay(150)
  // Update the state one more time to confirm the listener isn't checking it
  store.dispatch(increment())

  // Handled the state update before the delay, but not after
  expect(finalCount).toBe(2)
})
```

Cancellation and Task Management

The listener middleware supports cancellation of running listener instances, `take/condition/pause/delay` functions, and "child tasks", with an implementation based on [`AbortController`](AbortController).

The `listenerApi.pause/delay()` functions provide a cancellation-aware way to have the current listener sleep. `pause()` accepts a promise, while `delay` accepts a timeout value. If the listener is cancelled while waiting, a `TaskAbortError` will be thrown. In addition, both `take` and `condition` support cancellation interruption as well.

`listenerApi.fork()` can used to launch "child tasks" that can do additional work. These can be waited on to collect their results. An example of this might look like:

```
listenerMiddleware.startListening({
  actionCreator: increment,
  effect: async (action, listenerApi) => {
    // Spawn a child task and start it immediately
    const task = listenerApi.fork(async (forkApi) => {
      // Artificially wait a bit inside the child
      await forkApi.delay(5)
      // Complete the child by returning a value
      return 42
    })

    const result = await task.result
    // Unwrap the child result in the listener
    if (result.status === 'ok') {
      // Logs the `42` result value that was returned
      console.log('Child succeeded: ', result.value)
    }
  },
})
```

Complex Async Workflows

The provided async workflow primitives (`cancelActiveListeners`, `unsubscribe`, `subscribe`, `take`, `condition`, `pause`, `delay`) can be used to implement behavior that is equivalent to many of the more complex async workflow capabilities found in the Redux-Saga library. This includes effects such as `throttle`, `debounce`, `takeLatest`, `takeLeading`, and `fork/join`. Some examples from the test suite:

```
test('debounce / takeLatest', async () => {
  // Repeated calls cancel previous ones, no work performed
  // until the specified delay elapses without another call
  // NOTE: This is also basically identical to `takeLatest`.
  // Ref: https://redux-saga.js.org/docs/api#debouncems-pattern-saga-args
  // Ref: https://redux-saga.js.org/docs/api#takelatestpattern-saga-args

  listenerMiddleware.startListening({
    actionCreator: increment,
    effect: async (action, listenerApi) => {
      // Cancel any in-progress instances of this listener
      listenerApi.cancelActiveListeners()

      // Delay before starting actual work
      await listenerApi.delay(15)

      // do work here
    },
  })
}

test('takeLeading', async () => {
  // Starts listener on first action, ignores others until task completes
  // Ref: https://redux-saga.js.org/docs/api#takeleadingpattern-saga-args

  listenerMiddleware.startListening({
    actionCreator: increment,
    effect: async (action, listenerApi) => {
      listenerCalls++

      // Stop listening for this action
      listenerApi.unsubscribe()

      // Pretend we're doing expensive work

      // Re-enable the listener
      listenerApi.subscribe()
    },
  })
})

test('cancelled', async () => {
  // cancelled allows checking if the current task was cancelled
  // Ref: https://redux-saga.js.org/docs/api#cancelled

  let canceledAndCaught = false
  let canceledCheck = false

  // Example of canceling prior instances conditionally and checking cancellation
  listenerMiddleware.startListening({
    matcher: isAnyOf(increment, decrement, incrementByAmount),
    effect: async (action, listenerApi) => {
      if (increment.match(action)) {
        // Have this branch wait around to be cancelled by the other
        try {
          await listenerApi.delay(10)
        } catch (err) {
          // Can check cancellation based on the exception and its reason
          if (err instanceof TaskAbortError) {
            canceledAndCaught = true
          }
        }
      } else if (incrementByAmount.match(action)) {
        // do a non-cancellation-aware wait
        await delay(15)
        if (listenerApi.signal.aborted) {
          canceledCheck = true
        }
      } else if (decrement.match(action)) {
        listenerApi.cancelActiveListeners()
      }
    },
  })
})
```

As a more practical example: this saga-based "long polling" loop repeatedly asks the server for a message and then processes each response. The child loop is started on demand when a "start polling" action is dispatched, and the loop is cancelled when a "stop polling" action is dispatched.

That approach can be implemented via the listener middleware:

```
// Track how many times each message was processed by the loop
const receivedMessages = {
  a: 0,
  b: 0,
  c: 0,
}

const eventPollingStarted = createAction('serverPolling/started')
const eventPollingStopped = createAction('serverPolling/stopped')

listenerMiddleware.startListening({
  actionCreator: eventPollingStarted,
  effect: async (action, listenerApi) => {
    // Only allow one instance of this listener to run at a time
    listenerApi.unsubscribe()

    // Start a child job that will infinitely loop receiving messages
    const pollingTask = listenerApi.fork(async (forkApi) => {
      try {
        while (true) {
          // Cancellation-aware pause for a new server message
          const serverEvent = await forkApi.pause(pollForEvent())
          // Process the message. In this case, just count the times we've seen this message.
          if (serverEvent.type in receivedMessages) {
            receivedMessages[
              serverEvent.type as keyof typeof receivedMessages
            ]++
          }
        }
      } catch (err) {
        if (err instanceof TaskAbortError) {
          // could do something here to track that the task was cancelled
        }
      }
    })

    // Wait for the "stop polling" action
    await listenerApi.condition(eventPollingStopped.match)
    pollingTask.cancel()
  },
})
```

Adding Listeners Inside Components

Listeners can be added at runtime via `dispatch(addListener())`. This means that you can add listeners anywhere you have access to `dispatch`, and that includes React components.

Since dispatching `addListener` returns an `unsubscribe` callback, this naturally maps to the behavior of React `useEffect` hooks, which let you return a cleanup function. You can add a listener in an effect, and remove the listener when the hook is cleaned up.

The basic pattern might look like:

```
useEffect(() => {
  // Could also just `return dispatch(addListener())` directly, but showing this
  // as a separate variable to be clear on what's happening
  const unsubscribe = dispatch(
    addListener({
      actionCreator: todoAdded,
      effect: (action, listenerApi) => {
        // do some useful logic here
      },
    })
  )
  return unsubscribe
}, [])
```

While this pattern is *possible*, **we do not necessarily *recommend* doing this!** The React and Redux communities have always tried to emphasize basing behavior on *state* as much as possible. Having React components directly tie into the Redux action dispatch pipeline could potentialy lead to codebases that are more difficult to maintain.

At the same time, this *is* a valid technique, both in terms of API behavior and potential use cases. It's been common to lazy-load sagas as part of a code-split app, and that has often required some complex additional setup work to "inject" sagas. In contrast, `dispatch(addListener())` fits naturally into a React component's lifecycle.

So, while we're not specifically encouraging use of this pattern, it's worth documenting here so that users are aware of it as a possibility.

Organizing Listeners in Files

As a starting point, **it's best to create the listener middleware in a separate file, such as `app/listenerMiddleware.ts`, rather than in the same file as the store**. This avoids any potential circular import problems from other files trying to import `middleware.addListener`.

From there, so far we've come up with three different ways to organize listener functions and setup.

First, you can import effect callbacks from slice files into the middleware file, and add the listeners:

```
import { action1, listener1 } from '../features/feature1/feature1Slice'
import { action2, listener2 } from '../features/feature2/feature2Slice'

listenerMiddleware.startListening({ actionCreator: action1, effect: listener1 })
listenerMiddleware.startListening({ actionCreator: action2, effect: listener2 })
```

This is probably the simplest option, and mirrors how the store setup pulls together all the slice reducers to create the app.

The second option is the opposite: have the slice files import the middleware and directly add their listeners:

```
import { listenerMiddleware } from '../../app/listenerMiddleware'

const feature1Slice = createSlice(/* */)
const { action1 } = feature1Slice.actions

export default feature1Slice.reducer

listenerMiddleware.startListening({
  actionCreator: action1,
  effect: () => {},
})
```

This keeps all the logic in the slice, although it does lock the setup into a single middleware instance.

The third option is to create a setup function in the slice, but let the listener file call that on startup:

```
import type { AppStartListening } from '../../app/listenerMiddleware'

const feature1Slice = createSlice(/* */)
const { action1 } = feature1Slice.actions

export default feature1Slice.reducer

export const addFeature1Listeners = (startListening: AppStartListening) => {
  startListening({
    actionCreator: action1,
    effect: () => {},
  })
}
```

```
import { addFeature1Listeners } from '../features/feature1/feature1Slice'

addFeature1Listeners(listenerMiddleware.startListening)
```

Feel free to use whichever of these approaches works best in your app.

Last updated on ***Apr 7, 2023***

autoBatchEnhancer

A Redux store enhancer that looks for one or more "low-priority" dispatched actions in a row, and queues a callback to run subscriber notifications on a delay. It then notifies subscribers either when the queued callback runs, or when the next "normal-priority" action is dispatched, whichever is first.

Basic Usage

```
import {
  createSlice,
  configureStore,
  autoBatchEnhancer,
  prepareAutoBatched,
} from '@reduxjs/toolkit'

interface CounterState {
  value: number
}

const counterSlice = createSlice({
  name: 'counter',
  initialState: { value: 0 } as CounterState,
  reducers: {
    incrementBatched: {
      // Batched, low-priority
      reducer(state) {
        state.value += 1
      },
      // Use the `prepareAutoBatched` utility to automatically
      // add the `action.meta[SHOULD_AUTOBATCH]` field the enhancer needs
      prepare: prepareAutoBatched<void>(),
    },
    // Not batched, normal priority
    decrementUnbatched(state) {
      state.value -= 1
    },
  },
})
const { incrementBatched, decrementUnbatched } = counterSlice.actions

const store = configureStore({
  reducer: counterSlice.reducer,
  enhancers: (existingEnhancers) => {
    // Add the autobatch enhancer to the store setup
    return existingEnhancers.concat(autoBatchEnhancer())
  },
})
```

API

`autoBatchEnhancer`

```
export type SHOULD_AUTOBATCH = string
type AutoBatchOptions =
  | { type: 'tick' }
  | { type: 'timer'; timeout: number }
  | { type: 'raf' }
  | { type: 'callback'; queueNotification: (notify: () => void) => void }

export type autoBatchEnhancer = (options?: AutoBatchOptions) => StoreEnhancer
```

Creates a new instance of the autobatch store enhancer.

Any action that is tagged with `action.meta[SHOULD_AUTOBATCH] = true` will be treated as "low-priority", and a notification callback will be queued. The enhancer will delay notifying subscribers until either:

- The queued callback runs and triggers the notifications
- A "normal-priority" action (any action *without* `action.meta[SHOULD_AUTOBATCH] = true`) is dispatched in the same tick

`autoBatchEnhancer` accepts options to configure how the notification callback is queued:

- `{type: 'raf'}`: queues using `requestAnimationFrame` (default)
- `{type: 'tick'}`: queues using `queueMicrotask`
- `{type: 'timer, timeout: number}`: queues using `setTimeout`

- `{type: 'callback', queueNotification: (notify: () => void) => void}`: lets you provide your own callback, such as a debounced or throttled function

The default behavior is to queue the notifications using `requestAnimationFrame`.

The `SHOULD_AUTOBATCH` value is meant to be opaque - it's currently a string for simplicity, but could be a `Symbol` in the future.

`prepareAutoBatched`

```
type prepareAutoBatched = <T>() => (payload: T) => { payload: T; meta: unknown }
```

Creates a function that accepts a `payload` value, and returns an object with `{payload, meta: {[SHOULD_AUTOBATCH]: true}}`. This is meant to be used with RTK's `createSlice` and its "`prepare` callback" syntax:

Batching Approach and Background

The post A Comparison of Redux Batching Techniques describes four different approaches for "batching Redux actions/dispatches"

- a higher-order reducer that accepts multiple actions nested inside one real action, and iterates over them together
- an enhancer that wraps `dispatch` and debounces the notification callback
- an enhancer that wraps `dispatch` to accept an array of actions
- React's `unstable_batchedUpdates()`, which just combines multiple queued renders into one but doesn't affect subscriber notifications

This enhancer is a variation of the "debounce" approach, but with a twist.

Instead of *just* debouncing *all* subscriber notifications, it watches for any actions with a specific `action.meta[SHOULD_AUTOBATCH]: true` field attached.

When it sees an action with that field, it queues a callback. The reducer is updated immediately, but the enhancer does *not* notify subscribers right way. If other actions with the same field are dispatched in succession, the enhancer will continue to *not* notify subscribers. Then, when the queued callback runs, it finally notifies all subscribers, similar to how React batches re-renders.

The additional twist is also inspired by React's separation of updates into "low-priority" and "immediate" behavior (such as a render queued by an AJAX request vs a render queued by a user input that should be handled synchronously).

If some low-pri actions have been dispatched and a notification microtask is queued, then a *normal* priority action (without the field) is dispatched, the enhancer will go ahead and notify all subscribers synchronously as usual, and *not* notify them at the end of the tick.

This allows Redux users to selectively tag certain actions for effective batching behavior, making this purely opt-in on a per-action basis, while retaining normal notification behavior for all other actions.

RTK Query and Batching

RTK Query already marks several of its key internal action types as batchable. If you add the `autoBatchEnhancer` to the store setup, it will improve the overall UI performance, especially when rendering large lists of components that use the RTKQ query hooks.

*Last updated on **May 3, 2023***

`createReducer()`

Overview

A utility that simplifies creating Redux reducer functions. It uses Immer internally to drastically simplify immutable update logic by writing "mutative" code in your reducers, and supports directly mapping specific action types to case reducer functions that will update the state when that action is dispatched.

Redux reducers are often implemented using a `switch` statement, with one `case` for every handled action type.

```
const initialState = { value: 0 }

function counterReducer(state = initialState, action) {
  switch (action.type) {
    case 'increment':
      return { ...state, value: state.value + 1 }
    case 'decrement':
      return { ...state, value: state.value - 1 }
    case 'incrementByAmount':
      return { ...state, value: state.value + action.payload }
    default:
      return state
  }
}
```

This approach works well, but is a bit boilerplate-y and error-prone. For instance, it is easy to forget the `default` case or setting the initial state.

The `createReducer` helper streamlines the implementation of such reducers. It supports two different forms of defining case reducers to handle actions: a "builder callback" notation and a "map object" notation. Both are equivalent, but the "builder callback" notation is preferred.

With `createReducer`, your reducers instead look like:

```
import { createAction, createReducer } from '@reduxjs/toolkit'

interface CounterState {
  value: number
}

const increment = createAction('counter/increment')
const decrement = createAction('counter/decrement')
const incrementByAmount = createAction<number>('counter/incrementByAmount')

const initialState = { value: 0 } as CounterState

const counterReducer = createReducer(initialState, (builder) => {
  builder
    .addCase(increment, (state, action) => {
      state.value++
    })
    .addCase(decrement, (state, action) => {
      state.value--
    })
    .addCase(incrementByAmount, (state, action) => {
      state.value += action.payload
    })
})
```

Usage with the "Builder Callback" Notation

This overload accepts a callback function that receives a `builder` object as its argument. That builder provides `addCase`, `addMatcher` and `addDefaultCase` functions that may be called to define what actions this reducer will handle.

The recommended way of using `createReducer` is the builder callback notation, as it works best with TypeScript and most IDEs.

Parameters

- **initialState** `State | (() => State)`: The initial state that should be used when the reducer is called the first time. This may also be a "lazy initializer" function, which should return an initial state value when called. This will be used whenever the reducer is called with `undefined` as its state value, and is primarily useful for cases like reading initial state from `localStorage`.
- **builderCallback** `(builder: Builder) => void` A callback that receives a *builder* object to define case reducers via calls to `builder.addCase(actionCreatorOrType, reducer)`.

Example Usage

```
import {
  createAction,
  createReducer,
  AnyAction,
  PayloadAction,
} from '@reduxjs/toolkit'

const increment = createAction<number>('increment')
const decrement = createAction<number>('decrement')

function isActionWithNumberPayload(
  action: AnyAction
): action is PayloadAction<number> {
  return typeof action.payload === 'number'
}

const reducer = createReducer(
  {
    counter: 0,
    sumOfNumberPayloads: 0,
    unhandledActions: 0,
  },
  (builder) => {
    builder
      .addCase(increment, (state, action) => {
        // action is inferred correctly here
        state.counter += action.payload
      })
      // You can chain calls, or have separate `builder.addCase()` lines each time
      .addCase(decrement, (state, action) => {
        state.counter -= action.payload
      })
      // You can apply a "matcher function" to incoming actions
      .addMatcher(isActionWithNumberPayload, (state, action) => {})
      // and provide a default case if no other handlers matched
      .addDefaultCase((state, action) => {})
  }
)
```

Builder Methods

`builder.addCase`

Adds a case reducer to handle a single exact action type.

All calls to `builder.addCase` must come before any calls to `builder.addMatcher` or `builder.addDefaultCase`.

Parameters

- **actionCreator** Either a plain action type string, or an action creator generated by `createAction` that can be used to determine the action type.
- **reducer** The actual case reducer function.

`builder.addMatcher`

Allows you to match your incoming actions against your own filter function instead of only the `action.type` property.

If multiple matcher reducers match, all of them will be executed in the order they were defined in - even if a case reducer already matched. All calls to `builder.addMatcher` must come after any calls to `builder.addCase` and before any calls to `builder.addDefaultCase`.

Parameters

- **matcher** A matcher function. In TypeScript, this should be a type predicate function
- **reducer** The actual case reducer function.

`builder.addDefaultCase`

Adds a "default case" reducer that is executed if no case reducer and no matcher reducer was executed for this action.

Parameters

- **reducer** The fallback "default case" reducer function.

```
import { createReducer } from '@reduxjs/toolkit'
const initialState = { otherActions: 0 }
const reducer = createReducer(initialState, (builder) => {
  builder
    // .addCase(...)
    // .addMatcher(...)
    .addDefaultCase((state, action) => {
      state.otherActions++
    })
})
```

Usage with the "Map Object" Notation

This overload accepts an object where the keys are string action types, and the values are case reducer functions to handle those action types.

While this notation is a bit shorter, it works only in JavaScript, not TypeScript and has less integration with IDEs, so we recommend the "builder callback" notation in most cases.

Parameters

- **initialState** `State | (() => State)`: The initial state that should be used when the reducer is called the first time. This may also be a "lazy initializer" function, which should return an initial state value when called. This will be used whenever the reducer is called with `undefined` as its state value, and is primarily useful for cases like reading initial state from `localStorage`.
- **actionsMap** An object mapping from action types to *case reducers*, each of which handles one specific action type.

- **actionMatchers** An array of matcher definitions in the form `{matcher, reducer}`. All matching reducers will be executed in order, independently if a case reducer matched or not.
- **defaultCaseReducer** A "default case" reducer that is executed if no case reducer and no matcher reducer was executed for this action.

Returns

The generated reducer function.

The reducer will have a `getInitialState` function attached that will return the initial state when called. This may be useful for tests or usage with React's `useReducer` hook:

```
const counterReducer = createReducer(0, {
  increment: (state, action) => state + action.payload,
  decrement: (state, action) => state - action.payload,
})

console.log(counterReducer.getInitialState()) // 0
```

Example Usage

```
const counterReducer = createReducer(0, {
  increment: (state, action) => state + action.payload,
  decrement: (state, action) => state - action.payload
})

// Alternately, use a "lazy initializer" to provide the initial state
// (works with either form of createReducer)
const initialState = () => 0
const counterReducer = createReducer(initialState, {
  increment: (state, action) => state + action.payload,
  decrement: (state, action) => state - action.payload
})
```

Action creators that were generated using `createAction` may be used directly as the keys here, using computed property syntax:

```
const increment = createAction('increment')
const decrement = createAction('decrement')

const counterReducer = createReducer(0, {
  [increment]: (state, action) => state + action.payload,
  [decrement.type]: (state, action) => state - action.payload
})
```

Matchers and Default Cases as Arguments

The most readable approach to define matcher cases and default cases is by using the `builder.addMatcher` and `builder.addDefaultCase` methods described above, but it is also possible to use these with the object notation by passing an array of `{matcher, reducer}` objects as the third argument, and a default case reducer as the fourth argument:

```
const isStringPayloadAction = (action) => typeof action.payload === 'string'

const lengthOfAllStringsReducer = createReducer(
  // initial state
  { strLen: 0, nonStringActions: 0 },
  // normal reducers
  {
    /*...*/
  },
  //  array of matcher reducers
  [
    {
      matcher: isStringPayloadAction,
      reducer(state, action) {
        state.strLen += action.payload.length
      },
    },
  ],
  // default reducer
  (state) => {
    state.nonStringActions++
  }
)
```

Direct State Mutation

Redux requires reducer functions to be pure and treat state values as immutable. While this is essential for making state updates predictable and observable, it can sometimes make the implementation of such updates awkward. Consider the following example:

```
import { createAction, createReducer } from '@reduxjs/toolkit'

interface Todo {
  text: string
  completed: boolean
}

const addTodo = createAction<Todo>('todos/add')
const toggleTodo = createAction<number>('todos/toggle')

const todosReducer = createReducer([] as Todo[], (builder) => {
  builder
    .addCase(addTodo, (state, action) => {
      const todo = action.payload
      return [...state, todo]
    })
    .addCase(toggleTodo, (state, action) => {
      const index = action.payload
      const todo = state[index]
      return [
        ...state.slice(0, index),
        { ...todo, completed: !todo.completed },
        ...state.slice(index + 1),
      ]
    })
})
```

The `addTodo` reducer is straightforward if you know the ES6 spread syntax. However, the code for `toggleTodo` is much less straightforward, especially considering that it only sets a single flag.

To make things easier, `createReducer` uses immer to let you write reducers as if they were mutating the state directly. In reality, the reducer receives a proxy state that translates all mutations into equivalent copy operations.

```
import { createAction, createReducer } from '@reduxjs/toolkit'

interface Todo {
  text: string
  completed: boolean
}

const addTodo = createAction<Todo>('todos/add')
const toggleTodo = createAction<number>('todos/toggle')

const todosReducer = createReducer([] as Todo[], (builder) => {
  builder
    .addCase(addTodo, (state, action) => {
      // This push() operation gets translated into the same
      // extended-array creation as in the previous example.
      const todo = action.payload
      state.push(todo)
    })
    .addCase(toggleTodo, (state, action) => {
      // The "mutating" version of this case reducer is much
      //  more direct than the explicitly pure one.
      const index = action.payload
      const todo = state[index]
      todo.completed = !todo.completed
    })
})
```

Writing "mutating" reducers simplifies the code. It's shorter, there's less indirection, and it eliminates common mistakes made while spreading nested state. However, the use of Immer does add some "magic", and Immer has its own nuances in behavior. You should read through pitfalls mentioned in the immer docs . Most importantly, **you need to ensure that you either mutate the `state` argument or return a new state, *but not both***. For example, the following reducer would throw an exception if a `toggleTodo` action is passed:

```
import { createAction, createReducer } from '@reduxjs/toolkit'

interface Todo {
  text: string
  completed: boolean
}

const toggleTodo = createAction<number>('todos/toggle')

const todosReducer = createReducer([] as Todo[], (builder) => {
  builder.addCase(toggleTodo, (state, action) => {
    const index = action.payload
    const todo = state[index]

    // This case reducer both mutates the passed-in state...
    todo.completed = !todo.completed

    // ... and returns a new value. This will throw an
    // exception. In this example, the easiest fix is
    // to remove the `return` statement.
    return [...state.slice(0, index), todo, ...state.slice(index + 1)]
  })
})
```

Multiple Case Reducer Execution

Originally, `createReducer` always matched a given action type to a single case reducer, and only that one case reducer would execute for a given action.

Using action matchers changes that behavior, as multiple matchers may handle a single action.

For any dispatched action, the behavior is:

- If there is an exact match for the action type, the corresponding case reducer will execute first
- Any matchers that return `true` will execute in the order they were defined
- If a default case reducer is provided, and *no* case or matcher reducers ran, the default case reducer will execute
- If no case or matcher reducers ran, the original existing state value will be returned unchanged

The executing reducers form a pipeline, and each of them will receive the output of the previous reducer:

```
import { createReducer } from '@reduxjs/toolkit'

const reducer = createReducer(0, (builder) => {
  builder
    .addCase('increment', (state) => state + 1)
    .addMatcher(
      (action) => action.type.startsWith('i'),
      (state) => state * 5
    )
    .addMatcher(
      (action) => action.type.endsWith('t'),
      (state) => state + 2
    )
})

console.log(reducer(0, { type: 'increment' }))
// Returns 7, as the 'increment' case and both matchers all ran in sequence:
// - case 'increment": 0 => 1
// - matcher starts with 'i': 1 => 5
// - matcher ends with 't': 5 => 7
```

Logging Draft State Values

It's very common for a developer to call `console.log(state)` during the development process. However, browsers display Proxies in a format that is hard to read, which can make console logging of Immer-based state difficult.

When using either `createSlice` or `createReducer`, you may use the [`current`] utility that we re-export from the [`immer` library]. This utility creates a separate plain copy of the current Immer `Draft` state value, which can then be logged for viewing as normal.

```
import { createSlice, current } from '@reduxjs/toolkit'

const slice = createSlice({
  name: 'todos',
  initialState: [{ id: 1, title: 'Example todo' }],
  reducers: {
    addTodo: (state, action) => {
      console.log('before', current(state))
      state.push(action.payload)
      console.log('after', current(state))
    },
  },
})
```

Last updated on ***Nov 26, 2021***

createAction

A helper function for defining a Redux action type and creator.

```
function createAction(type, prepareAction?)
```

The usual way to define an action in Redux is to separately declare an *action type* constant and an *action creator* function for constructing actions of that type.

```
const INCREMENT = 'counter/increment'

function increment(amount: number) {
  return {
    type: INCREMENT,
    payload: amount,
  }
}

const action = increment(3)
// { type: 'counter/increment', payload: 3 }
```

The `createAction` helper combines these two declarations into one. It takes an action type and returns an action creator for that type. The action creator can be called either without arguments or with a `payload` to be attached to the action. Also, the action creator overrides toString() so that the action type becomes its string representation.

```
import { createAction } from '@reduxjs/toolkit'

const increment = createAction<number | undefined>('counter/increment')

let action = increment()
// { type: 'counter/increment' }

action = increment(3)
// returns { type: 'counter/increment', payload: 3 }

console.log(increment.toString())
// 'counter/increment'

console.log(`The action type is: ${increment}`)
// 'The action type is: counter/increment'
```

Using Prepare Callbacks to Customize Action Contents

By default, the generated action creators accept a single argument, which becomes `action.payload`. This requires the caller to construct the entire payload correctly and pass it in.

In many cases, you may want to write additional logic to customize the creation of the `payload` value, such as accepting multiple parameters for the action creator, generating a random ID, or getting the current timestamp. To do this, `createAction` accepts an optional second argument: a "prepare callback" that will be used to construct the payload value.

```
import { createAction, nanoid } from '@reduxjs/toolkit'

const addTodo = createAction('todos/add', function prepare(text: string) {
  return {
    payload: {
      text,
      id: nanoid(),
      createdAt: new Date().toISOString(),
    },
  }
})

console.log(addTodo('Write more docs'))
/**
 * {
 *   type: 'todos/add',
 *   payload: {
 *     text: 'Write more docs',
 *     id: '4AJvwMSWEHCchcWYga3dj',
 *     createdAt: '2019-10-03T07:53:36.581Z'
 *   }
 * }
 **/
```

If provided, all arguments from the action creator will be passed to the prepare callback, and it should return an object with the `payload` field (otherwise the payload of created actions will be `undefined`). Additionally, the object can have a `meta` and/or an `error` field that will also be added to created actions. `meta` may contain extra information about the action, `error` may contain details about the action failure. These three fields (`payload`, `meta` and `error`) adhere to the specification of Flux Standard Actions.

Note: The type field will be added automatically.

Usage with createReducer()

Because of their `toString()` override, action creators returned by `createAction()` can be used directly as keys for the case reducers passed to createReducer().

```
import { createAction, createReducer } from '@reduxjs/toolkit'

const increment = createAction<number>('counter/increment')
const decrement = createAction<number>('counter/decrement')

const counterReducer = createReducer(0, (builder) => {
  builder.addCase(increment, (state, action) => state + action.payload)
  builder.addCase(decrement, (state, action) => state - action.payload)})
```

Non-String Action Types

In principle, Redux lets you use any kind of value as an action type. Instead of strings, you could theoretically use numbers, symbols, or anything else (although it's recommended that the value should at least be serializable).

However, Redux Toolkit rests on the assumption that you use string action types. Specifically, some of its features rely on the fact that with strings, the `toString()` method of an `createAction()` action creator returns the matching action type. This is not the case for non-string action types because `toString()` will return the string-converted type value rather than the type itself.

```
const INCREMENT = Symbol('increment')
const increment = createAction(INCREMENT)

increment.toString()
// returns the string 'Symbol(increment)',
// not the INCREMENT symbol itself

increment.toString() === INCREMENT
// false
```

This means that, for instance, you cannot use a non-string-type action creator as a case reducer key for createReducer().

```
const INCREMENT = Symbol('increment')
const increment = createAction(INCREMENT)

const counterReducer = createReducer(0, {
  // The following case reducer will NOT trigger for
  // increment() actions because `increment` will be
  // interpreted as a string, rather than being evaluated
  // to the INCREMENT symbol.
  [increment]: (state, action) => state + action.payload,

  // You would need to use the action type explicitly instead.
  [INCREMENT]: (state, action) => state + action.payload,
})
```

For this reason, **we strongly recommend you to only use string action types**.

actionCreator.match

Every generated actionCreator has a `.match(action)` method that can be used to determine if the passed action is of the same type as an action that would be created by the action creator.

This has different uses:

As a TypeScript Type Guard

This `match` method is a TypeScript type guard and can be used to discriminate the `payload` type of an action.

This behavior can be particularly useful when used in custom middlewares, where manual casts might be neccessary otherwise.

```
import { createAction } from '@reduxjs/toolkit'
import type { Action } from '@reduxjs/toolkit'

const increment = createAction<number>('INCREMENT')

function someFunction(action: Action) {
  // accessing action.payload would result in an error here
  if (increment.match(action)) {
    // action.payload can be used as `number` here
  }
}
```

With redux-observable

The `match` method can also be used as a filter method, which makes it powerful when used with redux-observable:

```
import { createAction } from '@reduxjs/toolkit'
import type { Action } from '@reduxjs/toolkit'
import type { Observable } from 'rxjs'
import { map, filter } from 'rxjs/operators'

const increment = createAction<number>('INCREMENT')

export const epic = (actions$: Observable<Action>) =>
  actions$.pipe(
    filter(increment.match),
    map((action) => {
      // action.payload can be safely used as number here (and will also be correctly inferred by TypeScript)
      // ...
    })
  )
```

*Last updated on **Jun 24, 2022***

createSlice

A function that accepts an initial state, an object of reducer functions, and a "slice name", and automatically generates action creators and action types that correspond to the reducers and state.

This API is the standard approach for writing Redux logic.

Internally, it uses `createAction` and `createReducer`, so you may also use Immer to write "mutating" immutable updates:

```
import { createSlice } from '@reduxjs/toolkit'
import type { PayloadAction } from '@reduxjs/toolkit'

interface CounterState {
  value: number
}

const initialState = { value: 0 } as CounterState

const counterSlice = createSlice({
  name: 'counter',
  initialState,
  reducers: {
    increment(state) {
      state.value++
    },
    decrement(state) {
      state.value--
    },
    incrementByAmount(state, action: PayloadAction<number>) {
      state.value += action.payload
    },
  },
})

export const { increment, decrement, incrementByAmount } = counterSlice.actions
export default counterSlice.reducer
```

Parameters

`createSlice` accepts a single configuration object parameter, with the following options:

```
function createSlice({
    // A name, used in action types
    name: string,
    // The initial state for the reducer
    initialState: any,
    // An object of "case reducers". Key names will be used to generate actions.
    reducers: Object<string, ReducerFunction | ReducerAndPrepareObject>
    // A "builder callback" function used to add more reducers, or
    // an additional object of "case reducers", where the keys should be other
    // action types
```

```
  extraReducers?:
  | Object<string, ReducerFunction>
  | ((builder: ActionReducerMapBuilder<State>) => void)
})
```

`initialState`

The initial state value for this slice of state.

This may also be a "lazy initializer" function, which should return an initial state value when called. This will be used whenever the reducer is called with `undefined` as its state value, and is primarily useful for cases like reading initial state from `localStorage`.

`name`

A string name for this slice of state. Generated action type constants will use this as a prefix.

`reducers`

An object containing Redux "case reducer" functions (functions intended to handle a specific action type, equivalent to a single case statement in a switch).

The keys in the object will be used to generate string action type constants, and these will show up in the Redux DevTools Extension when they are dispatched. Also, if any other part of the application happens to dispatch an action with the exact same type string, the corresponding reducer will be run. Therefore, you should give the functions descriptive names.

This object will be passed to `createReducer`, so the reducers may safely "mutate" the state they are given.

```
import { createSlice } from '@reduxjs/toolkit'

const counterSlice = createSlice({
 name: 'counter',
 initialState: 0,
 reducers: {
  increment: (state) => state + 1,
 },
})
// Will handle the action type `'counter/increment'`
```

Customizing Generated Action Creators

If you need to customize the creation of the payload value of an action creator by means of a `prepare callback`, the value of the appropriate field of the `reducers` argument object should be an object instead of a function. This object must contain two properties: `reducer` and `prepare`. The value of the `reducer` field should be the case reducer function while the value of the `prepare` field should be the prepare callback function:

```
import { createSlice, nanoid } from '@reduxjs/toolkit'
import type { PayloadAction } from '@reduxjs/toolkit'

interface Item {
  id: string
  text: string
}

const todosSlice = createSlice({
  name: 'todos',
  initialState: [] as Item[],
  reducers: {
    addTodo: {
      reducer: (state, action: PayloadAction<Item>) => {
        state.push(action.payload)
      },
      prepare: (text: string) => {
        const id = nanoid()
        return { payload: { id, text } }
      },
    },
  },
})
```

`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.

```
import { createAction, createSlice, Action, AnyAction } from '@reduxjs/toolkit'
const incrementBy = createAction<number>('incrementBy')
const decrement = createAction('decrement')

interface RejectedAction extends Action {
  error: Error
}

function isRejectedAction(action: AnyAction): action is RejectedAction {
  return action.type.endsWith('rejected')
}

createSlice({
  name: 'counter',
  initialState: 0,
  reducers: {},
  extraReducers: (builder) => {
    builder
      .addCase(incrementBy, (state, action) => {
        // action is inferred correctly here if using TS
      })
      // You can chain calls, or have separate `builder.addCase()` lines each time
      .addCase(decrement, (state, action) => {})
      // You can match a range of action types
      .addMatcher(
        isRejectedAction,
        // `action` will be inferred as a RejectedAction due to isRejectedAction being defined as a type guard
        (state, action) => {}
      )
      // and provide a default case if no other handlers matched
      .addDefaultCase((state, action) => {})
  },
})
```

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.

```
const incrementBy = createAction('incrementBy')

createSlice({
  name: 'counter',
  initialState: 0,
  reducers: {},
  extraReducers: {
    [incrementBy]: (state, action) => {
      return state + action.payload
    },
    'some/other/action': (state, action) => {},
  },
})
```

We recommend using the `builder callback` API as the default, especially if you are using TypeScript. If you do not use the `builder callback` and are using TypeScript, you will need to use `actionCreator.type` or `actionCreator.toString()` to force the TS compiler to accept the computed property. Please see Usage With TypeScript for further details.

Return Value

`createSlice` will return an object that looks like:

```
{
    name : string,
    reducer : ReducerFunction,
    actions : Record<string, ActionCreator>,
    caseReducers: Record<string, CaseReducer>.
    getInitialState: () => State
}
```

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.

The functions passed to the `reducers` parameter can be accessed through the `caseReducers` return field. This can be particularly useful for testing or direct access to reducers created inline.

Result's function `getInitialState` provides access to the initial state value given to the slice. If a lazy state initializer was provided, it will be called and a fresh value returned.

Note: the result object is conceptually similar to a "Redux duck" code structure. The actual code structure you use is up to you, but there are a couple caveats to keep in mind:

- Actions are not exclusively limited to a single slice. Any part of the reducer logic can (and should!) respond to any dispatched action.
- At the same time, circular references can cause import problems. If slices A and B are defined in separate files, and each file tries to import the other so it can listen to other actions, unexpected behavior may occur.

Examples

```
import { createSlice, createAction } from '@reduxjs/toolkit'
import type { PayloadAction } from '@reduxjs/toolkit'
import { createStore, combineReducers } from 'redux'

const incrementBy = createAction<number>('incrementBy')
const decrementBy = createAction<number>('decrementBy')

const counter = createSlice({
  name: 'counter',
  initialState: 0 as number,
  reducers: {
    increment: (state) => state + 1,
    decrement: (state) => state - 1,
    multiply: {
      reducer: (state, action: PayloadAction<number>) => state * action.payload,
      prepare: (value?: number) => ({ payload: value || 2 }), // fallback if the payload is a falsy value
    },
  },
  // "builder callback API", recommended for TypeScript users
  extraReducers: (builder) => {
    builder.addCase(incrementBy, (state, action) => {
      return state + action.payload
    })
    builder.addCase(decrementBy, (state, action) => {
      return state - action.payload
    })
  },
})

const user = createSlice({
  name: 'user',
  initialState: { name: '', age: 20 },
  reducers: {
    setUserName: (state, action) => {
      state.name = action.payload // mutate the state all you want with immer
    },
  },
  // "map object API"
  extraReducers: {
    // @ts-expect-error in TypeScript, this would need to be [counter.actions.increment.type]
    [counter.actions.increment]: (
      state,
      action /* action will be inferred as "any", as the map notation does not contain type information */
    ) => {
      state.age += 1
    },
  },
})

const reducer = combineReducers({
  counter: counter.reducer,
  user: user.reducer,
})

const store = createStore(reducer)

store.dispatch(counter.actions.increment())
// -> { counter: 1, user: {name : '', age: 21} }
store.dispatch(counter.actions.increment())
// -> { counter: 2, user: {name: '', age: 22} }
store.dispatch(counter.actions.multiply(3))
// -> { counter: 6, user: {name: '', age: 22} }
store.dispatch(counter.actions.multiply())
// -> { counter: 12, user: {name: '', age: 22} }
console.log(`${counter.actions.decrement}`)
// -> "counter/decrement"
store.dispatch(user.actions.setUserName('eric'))
// -> { counter: 12, user: { name: 'eric', age: 22} }
```

*Last updated on **Aug 17, 2022***

createAsyncThunk

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.

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.

TIP

Redux Toolkit's **RTK Query data fetching API** is a purpose built data fetching and caching solution for Redux apps, and can **eliminate the need to write *any* thunks or reducers to manage data fetching**. We encourage you to try it out and see if it can help simplify the data fetching code in your own apps!

Sample usage:

```
import { createAsyncThunk, createSlice } from '@reduxjs/toolkit'
import { userAPI } from './userAPI'

// First, create the thunk
const fetchUserById = createAsyncThunk(
  'users/fetchByIdStatus',
  async (userId: number, thunkAPI) => {
    const response = await userAPI.fetchById(userId)
    return response.data
  }
)

interface UsersState {
  entities: []
  loading: 'idle' | 'pending' | 'succeeded' | 'failed'
}

const initialState = {
  entities: [],
  loading: 'idle',
} as UsersState

// Then, handle actions in your reducers:
const usersSlice = createSlice({
  name: 'users',
  initialState,
  reducers: {
    // standard reducer logic, with auto-generated action types per reducer
  },
  extraReducers: (builder) => {
    // Add reducers for additional action types here, and handle loading state as needed
    builder.addCase(fetchUserById.fulfilled, (state, action) => {
      // Add user to the state array
      state.entities.push(action.payload)
    })
  },
})

// Later, dispatch the thunk as needed in the app
dispatch(fetchUserById(123))
```

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. 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'}))`.
- `thunkAPI`: an object containing all of the parameters that are normally passed to a Redux thunk function, as well as additional options:
    - `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 cancelation.
    - `rejectWithValue(value, [meta])`: rejectWithValue is a utility function that you can `return` (or `throw`) in your action creator to return a rejected response with a defined payload and meta. It will pass whatever value you give it and return it in the payload of the rejected action. If you also pass in a `meta`, it will be merged with the existing `rejectedAction.meta`.
    - `fulfillWithValue(value, meta)`: fulfillWithValue is a utility function that you can `return` in your action creator to `fulfill` with a value while having the ability of adding to `fulfilledAction.meta`.

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(arg, { getState, extra } ): boolean | Promise<boolean>`: 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(arg): string`: a function to use when generating the `requestId` for the request sequence. Defaults to use nanoid, but you can implement your own ID generation logic.
- `serializeError(error: unknown) => any` to replace the internal `miniSerializeError` method with your own serialization logic.
- `getPendingMeta({ arg, requestId }, { getState, extra }): any`: a function to create an object that will be merged into the `pendingAction.meta` field.

Return Value

`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
    - `fetchUserById.rejected`, an action creator that dispatches an `'users/fetchByIdStatus/rejected'` action

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 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)

Promise Lifecycle Actions

`createAsyncThunk` will generate three Redux action creators using [`createAction`](): `pending`, `fulfilled`, and `rejected`. Each lifecycle action creator will be attached to the returned thunk action creator so that your reducer logic can reference the action types and respond to the actions when dispatched. Each action object will contain the current unique `requestId` and `arg` values under `action.meta`.

The action creators will have these signatures:

```
interface SerializedError {
  name?: string
  message?: string
  code?: string
  stack?: string
}

interface PendingAction<ThunkArg> {
  type: string
  payload: undefined
  meta: {
    requestId: string
    arg: ThunkArg
  }
}

interface FulfilledAction<ThunkArg, PromiseResult> {
  type: string
  payload: PromiseResult
  meta: {
    requestId: string
    arg: ThunkArg
  }
}

interface RejectedAction<ThunkArg> {
  type: string
  payload: undefined
  error: SerializedError | any
  meta: {
    requestId: string
    arg: ThunkArg
    aborted: boolean
    condition: boolean
  }
}

interface RejectedWithValueAction<ThunkArg, RejectedValue> {
  type: string
  payload: RejectedValue
  error: { message: 'Rejected' }
  meta: {
    requestId: string
    arg: ThunkArg
    aborted: boolean
  }
}

type Pending = <ThunkArg>(
  requestId: string,
  arg: ThunkArg
) => PendingAction<ThunkArg>

type Fulfilled = <ThunkArg, PromiseResult>(
  payload: PromiseResult,
  requestId: string,
  arg: ThunkArg
) => FulfilledAction<ThunkArg, PromiseResult>

type Rejected = <ThunkArg>(
  requestId: string,
  arg: ThunkArg
) => RejectedAction<ThunkArg>

type RejectedWithValue = <ThunkArg, RejectedValue>(
  requestId: string,
  arg: ThunkArg
) => RejectedWithValueAction<ThunkArg, RejectedValue>
```

To handle these actions in your reducers, reference the action creators in `createReducer` or `createSlice` using either the object key notation or the "builder callback" notation. (Note that if you use TypeScript, you should use the "builder callback" notation to ensure the types are inferred correctly):

```
const reducer1 = createReducer(initialState, {
  [fetchUserById.fulfilled]: (state, action) => {},
})

const reducer2 = createReducer(initialState, (builder) => {
  builder.addCase(fetchUserById.fulfilled, (state, action) => {})
})

const reducer3 = createSlice({
  name: 'users',
  initialState,
  reducers: {},
  extraReducers: {
    [fetchUserById.fulfilled]: (state, action) => {},
  },
})

const reducer4 = createSlice({
  name: 'users',
  initialState,
  reducers: {},
  extraReducers: (builder) => {
    builder.addCase(fetchUserById.fulfilled, (state, action) => {})
  },
})
```

Handling Thunk Results

Unwrapping Result Actions

Thunks may return a value when dispatched. A common use case is to return a promise from the thunk, dispatch the thunk from a component, and then wait for the promise to resolve before doing additional work:

```
const onClick = () => {
  dispatch(fetchUserById(userId)).then(() => {
    // do additional work
  })
}
```

The thunks generated by `createAsyncThunk` **will always return a resolved promise** with either the `fulfilled` action object or `rejected` action object inside, as appropriate.

The calling logic may wish to treat these actions as if they were the original promise contents. The promise returned by the dispatched thunk has an `unwrap` property which can be called to extract the `payload` of a `fulfilled` action or to throw either the `error` or, if available, `payload` created by `rejectWithValue` from a `rejected` action:

```
// in the component

const onClick = () => {
  dispatch(fetchUserById(userId))
    .unwrap()
    .then((originalPromiseResult) => {
      // handle result here
    })
    .catch((rejectedValueOrSerializedError) => {
      // handle error here
    })
}
```

Or with async/await syntax:

```
// in the component

const onClick = async () => {
  try {
    const originalPromiseResult = await dispatch(fetchUserById(userId)).unwrap()
    // handle result here
  } catch (rejectedValueOrSerializedError) {
    // handle error here
  }
}
```

Using the attached `.unwrap()` property is preferred in most cases, however Redux Toolkit also exports an `unwrapResult` function that can be used for a similar purpose:

```
import { unwrapResult } from '@reduxjs/toolkit'

// in the component
const onClick = () => {
  dispatch(fetchUserById(userId))
    .then(unwrapResult)
    .then((originalPromiseResult) => {
      // handle result here
    })
    .catch((rejectedValueOrSerializedError) => {
      // handle result here
    })
}
```

Or with async/await syntax:

```
import { unwrapResult } from '@reduxjs/toolkit'
```

```
// in the component
const onClick = async () => {
  try {
    const resultAction = await dispatch(fetchUserById(userId))
    const originalPromiseResult = unwrapResult(resultAction)
    // handle result here
  } catch (rejectedValueOrSerializedError) {
    // handle error here
  }
}
```

Checking Errors After Dispatching

Note that this means **a failed request or error in a thunk will *never* return a *rejected* promise**. We assume that any failure is more of a handled error than an unhandled exception at this point. This is due to the fact that we want to prevent uncaught promise rejections for those who do not use the result of `dispatch`.

If your component needs to know if the request failed, use `.unwrap` or `unwrapResult` and handle the re-thrown error accordingly.

Handling Thunk Errors

When your `payloadCreator` returns a rejected promise (such as a thrown error in an `async` function), the thunk will dispatch a `rejected` action containing an automatically-serialized version of the error as `action.error`. However, to ensure serializability, everything that does not match the `SerializedError` interface will have been removed from it:

```
export interface SerializedError {
  name?: string
  message?: string
  stack?: string
  code?: string
}
```

If you need to customize the contents of the `rejected` action, you should catch any errors yourself, and then **return** a new value using the `thunkAPI.rejectWithValue` utility. Doing `return rejectWithValue(errorPayload)` will cause the `rejected` action to use that value as `action.payload`.

The `rejectWithValue` approach should also be used if your API response "succeeds", but contains some kind of additional error details that the reducer should know about. This is particularly common when expecting field-level validation errors from an API.

```
const updateUser = createAsyncThunk(
  'users/update',
  async (userData, { rejectWithValue }) => {
    const { id, ...fields } = userData
    try {
      const response = await userAPI.updateById(id, fields)
      return response.data.user
    } catch (err) {
      // Use `err.response.data` as `action.payload` for a `rejected` action,
      // by explicitly returning it using the `rejectWithValue()` utility
      return rejectWithValue(err.response.data)
    }
  }
)
```

Cancellation

Canceling Before Execution

If you need to cancel a thunk before the payload creator is called, you may provide a `condition` callback as an option after the payload creator. The callback will receive the thunk argument and an object with `{getState, extra}` as parameters, and use those to decide whether to continue or not. If the execution should be canceled, the `condition` callback should return a literal `false` value or a promise that should resolve to `false`. If a promise is returned, the thunk waits for it to get fulfilled before dispatching the `pending` action, otherwise it proceeds with dispatching synchronously.

```
const fetchUserById = createAsyncThunk(
  'users/fetchByIdStatus',
  async (userId: number, thunkAPI) => {
    const response = await userAPI.fetchById(userId)
    return response.data
  },
  {
    condition: (userId, { getState, extra }) => {
      const { users } = getState()
      const fetchStatus = users.requests[userId]
      if (fetchStatus === 'fulfilled' || fetchStatus === 'loading') {
        // Already fetched or in progress, don't need to re-fetch
        return false
      }
    },
  }
)
```

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, pass in `{condition, dispatchConditionRejection: true}`.

Canceling While Running

If you want to cancel your running thunk before it has finished, you can use the `abort` method of the promise returned by `dispatch(fetchUserById(userId))`.

A real-life example of that would look like this:

```
// file: store.ts noEmit
import { configureStore } from '@reduxjs/toolkit'
import type { Reducer } from '@reduxjs/toolkit'
import { useDispatch } from 'react-redux'

declare const reducer: Reducer<{}>
const store = configureStore({ reducer })
export const useAppDispatch = () => useDispatch<typeof store.dispatch>()

// file: slice.ts noEmit
import { createAsyncThunk } from '@reduxjs/toolkit'
export const fetchUserById = createAsyncThunk(
  'fetchUserById',
  (userId: string) => {
    /* ... */
  }
)

// file: MyComponent.ts
```

```
import { fetchUserById } from './slice'
import { useAppDispatch } from './store'
import React from 'react'

function MyComponent(props: { userId: string }) {
  const dispatch = useAppDispatch()
  React.useEffect(() => {
    // Dispatching the thunk returns a promise
    const promise = dispatch(fetchUserById(props.userId))
    return () => {
      // `createAsyncThunk` attaches an `abort()` method to the promise
      promise.abort()
    }
  }, [props.userId])
}
```

After a thunk has been cancelled this way, it will dispatch (and return) a `"thunkName/rejected"` action with an `AbortError` on the `error` property. The thunk will not dispatch any further actions.

Additionally, your `payloadCreator` can use the `AbortSignal` it is passed via `thunkAPI.signal` to actually cancel a costly asynchronous action.

The `fetch` api of modern browsers already comes with support for an `AbortSignal`:

```
import { createAsyncThunk } from '@reduxjs/toolkit'

const fetchUserById = createAsyncThunk(
  'users/fetchById',
  async (userId: string, thunkAPI) => {
    const response = await fetch(`https://reqres.in/api/users/${userId}`, {
      signal: thunkAPI.signal,
    })
    return await response.json()
  }
)
```

Checking Cancellation Status

Reading the Signal Value

You can use the `signal.aborted` property to regularly check if the thunk has been aborted and in that case stop costly long-running work:

```
import { createAsyncThunk } from '@reduxjs/toolkit'

const readStream = createAsyncThunk(
  'readStream',
  async (stream: ReadableStream, { signal }) => {
    const reader = stream.getReader()

    let done = false
    let result = ''

    while (!done) {
      if (signal.aborted) {
        throw new Error('stop the work, this has been aborted!')
      }
      const read = await reader.read()
      result += read.value
      done = read.done
    }
    return result
  }
)
```

Listening for Abort Events

You can also call `signal.addEventListener('abort', callback)` to have logic inside the thunk be notified when `promise.abort()` was called. This can for example be used in conjunction with an axios `CancelToken`:

```
import { createAsyncThunk } from '@reduxjs/toolkit'
import axios from 'axios'

const fetchUserById = createAsyncThunk(
  'users/fetchById',
  async (userId: string, { signal }) => {
    const source = axios.CancelToken.source()
    signal.addEventListener('abort', () => {
      source.cancel()
    })
    const response = await axios.get(`https://reqres.in/api/users/${userId}`, {
      cancelToken: source.token,
    })
    return response.data
  }
)
```

Checking if a Promise Rejection was from an Error or Cancellation

To investigate behavior around thunk cancellation, you can inspect various properties on the `meta` object of the dispatched action. If a thunk was cancelled, the result of the promise will be a `rejected` action (regardless of whether that action was actually dispatched to the store).

- If it was cancelled before execution, `meta.condition` will be true.
- If it was aborted while running, `meta.aborted` will be true.
- If neither of those is true, the thunk was not cancelled, it was simply rejected, either by a Promise rejection or `rejectWithValue`.
- If the thunk was not rejected, both `meta.aborted` and `meta.condition` will be `undefined`.

So if you wanted to test that a thunk was cancelled before executing, you can do the following:

```
import { createAsyncThunk, isRejected } from '@reduxjs/toolkit'

test('this thunk should always be skipped', async () => {
  const thunk = createAsyncThunk(
    'users/fetchById',
    async () => throw new Error('This promise should never be entered'),
    {
      condition: () => false,
    }
  )
  const result = await thunk()(dispatch, getState, null)

  expect(result.meta.condition).toBe(true)
  expect(result.meta.aborted).toBe(false)
})
```

Examples

- Requesting a user by ID, with loading state, and only one request at a time:

```
import { createAsyncThunk, createSlice } from '@reduxjs/toolkit'
import { userAPI, User } from './userAPI'

const fetchUserById = createAsyncThunk<User, string, {
    state: { users: { loading: string, currentRequestId: string } }
}> (
  'users/fetchByIdStatus',
  async (userId: string, { getState, requestId }) => {
    const { currentRequestId, loading } = getState().users
    if (loading !== 'pending' || requestId !== currentRequestId) {
      return
    }
    const response = await userAPI.fetchById(userId)
    return response.data
  }
)

const usersSlice = createSlice({
  name: 'users',
  initialState: {
    entities: [],
    loading: 'idle',
    currentRequestId: undefined,
    error: null,
  },
  reducers: {},
  extraReducers: (builder) => {
    builder
      .addCase(fetchUserById.pending, (state, action) => {
        if (state.loading === 'idle') {
          state.loading = 'pending'
          state.currentRequestId = action.meta.requestId
        }
      })
      .addCase(fetchUserById.fulfilled, (state, action) => {
        const { requestId } = action.meta
        if (
          state.loading === 'pending' &&
          state.currentRequestId === requestId
        ) {
          state.loading = 'idle'
          state.entities.push(action.payload)
          state.currentRequestId = undefined
        }
      })
      .addCase(fetchUserById.rejected, (state, action) => {
        const { requestId } = action.meta
        if (
          state.loading === 'pending' &&
          state.currentRequestId === requestId
        ) {
          state.loading = 'idle'
          state.error = action.error
          state.currentRequestId = undefined
        }
      })
  },
})

const UsersComponent = () => {
  const { entities, loading, error } = useSelector((state) => state.users)
  const dispatch = useDispatch()

  const fetchOneUser = async (userId) => {
    try {
      const user = await dispatch(fetchUserById(userId)).unwrap()
      showToast('success', `Fetched ${user.name}`)
    } catch (err) {
      showToast('error', `Fetch failed: ${err.message}`)
    }
  }

  // render UI here}
```

- Using rejectWithValue to access a custom rejected payload in a component

  Note: this is a contrived example assuming our userAPI only ever throws validation-specific errors

```
// file: store.ts noEmit
import { configureStore } from '@reduxjs/toolkit'
import type { Reducer } from '@reduxjs/toolkit'
import { useDispatch } from 'react-redux'
import usersReducer from './user/slice'

const store = configureStore({ reducer: { users: usersReducer } })
export const useAppDispatch = () => useDispatch<typeof store.dispatch>()
export type RootState = ReturnType<typeof store.getState>

// file: user/userAPI.ts noEmit

export declare const userAPI: {
  updateById<Response>(id: string, fields: {}): { data: Response }
}

// file: user/slice.ts
import { createAsyncThunk, createSlice } from '@reduxjs/toolkit'
import { userAPI } from './userAPI'
import type { AxiosError } from 'axios'

// Sample types that will be used
export interface User {
  id: string
  first_name: string
  last_name: string
  email: string
}

interface ValidationErrors {
  errorMessage: string
  field_errors: Record<string, string>
}

interface UpdateUserResponse {
  user: User
  success: boolean
}

export const updateUser = createAsyncThunk<
  User,
  { id: string } & Partial<User>,
  {
    rejectValue: ValidationErrors
  }
>('users/update', async (userData, { rejectWithValue }) => {
  try {
    const { id, ...fields } = userData
    const response = await userAPI.updateById<UpdateUserResponse>(id, fields)
    return response.data.user
  } catch (err) {
    let error: AxiosError<ValidationErrors> = err // cast the error for access
    if (!error.response) {
      throw err
    }
    // We got validation errors, let's return those so we can reference in our component and set form
errors
    return rejectWithValue(error.response.data)
  }
})

interface UsersState {
  error: string | null | undefined
  entities: Record<string, User>
}

const initialState = {
  entities: {},
  error: null,
} as UsersState

const usersSlice = createSlice({
  name: 'users',
  initialState,
  reducers: {},
  extraReducers: (builder) => {
    // The `builder` callback form is used here because it provides correctly typed reducers from the action
creators
    builder.addCase(updateUser.fulfilled, (state, { payload }) => {
      state.entities[payload.id] = payload
    })
    builder.addCase(updateUser.rejected, (state, action) => {
      if (action.payload) {
        // Being that we passed in ValidationErrors to rejectType in `createAsyncThunk`, the payload will be
available here.
        state.error = action.payload.errorMessage
      } else {
        state.error = action.error.message
      }
    })
  },
})

export default usersSlice.reducer

// file: externalModules.d.ts noEmit

declare module 'some-toast-library' {
  export function showToast(type: string, message: string)
}

// file: user/UsersComponent.ts

import React from 'react'
import { useAppDispatch } from '../store'
import type { RootState } from '../store'
import { useSelector } from 'react-redux'
import { updateUser } from './slice'
import type { User } from './slice'
import type { FormikHelpers } from 'formik'
import { showToast } from 'some-toast-library'
```

```
interface FormValues extends Omit<User, 'id'> {}

const UsersComponent = (props: { id: string }) => {
  const { entities, error } = useSelector((state: RootState) => state.users)
  const dispatch = useAppDispatch()

  // This is an example of an onSubmit handler using Formik meant to demonstrate accessing the payload of the rejected action
  const handleUpdateUser = async (
    values: FormValues,
    formikHelpers: FormikHelpers<FormValues>
  ) => {
    const resultAction = await dispatch(updateUser({ id: props.id, ...values }))
    if (updateUser.fulfilled.match(resultAction)) {
      // user will have a type signature of User as we passed that as the Returned parameter in createAsyncThunk
      const user = resultAction.payload
      showToast('success', `Updated ${user.first_name} ${user.last_name}`)
    } else {
      if (resultAction.payload) {
        // Being that we passed in ValidationErrors to rejectType in `createAsyncThunk`, those types will be available here.
        formikHelpers.setErrors(resultAction.payload.field_errors)
      } else {
        showToast('error', `Update failed: ${resultAction.error}`)
      }
    }
  }

  // render UI here
}
```

*Last updated on **Jun 24, 2022***

`createEntityAdapter`

Overview

A function that generates a set of prebuilt reducers and selectors for performing CRUD operations on a normalized state structure containing instances of a particular type of data object. These reducer functions may be passed as case reducers to `createReducer` and `createSlice`. They may also be used as "mutating" helper functions inside of `createReducer` and `createSlice`.

This API was ported from the `@ngrx/entity` library created by the NgRx maintainers, but has been significantly modified for use with Redux Toolkit. We'd like to thank the NgRx team for originally creating this API and allowing us to port and adapt it for our needs.

Note: The term "Entity" is used to refer to a unique type of data object in an application. For example, in a blogging application, you might have `User`, `Post`, and `Comment` data objects, with many instances of each being stored in the client and persisted on the server. `User` is an "entity" - a unique type of data object that the application uses. Each unique instance of an entity is assumed to have a unique ID value in a specific field.

As with all Redux logic, *only* plain JS objects and arrays should be passed in to the store - **no class instances!**

For purposes of this reference, we will use `Entity` to refer to the specific data type that is being managed by a copy of the reducer logic in a specific portion of the Redux state tree, and `entity` to refer to a single instance of that type. Example:
in `state.users`, `Entity` would refer to the `User` type,
and `state.users.entities[123]` would be a single `entity`.

The methods generated by `createEntityAdapter` will all manipulate an "entity state" structure that looks like:

```
{
  // The unique IDs of each item. Must be strings or numbers
  ids: [ ]
  // A lookup table mapping entity IDs to the corresponding entity objects
  entities: {
  }
}
```

`createEntityAdapter` may be called multiple times in an application. If you are using it with plain JavaScript, you may be able to reuse a single adapter definition with multiple entity types if they're similar enough (such as all having an `entity.id` field). For TypeScript usage, you will need to call `createEntityAdapter` a separate time for each distinct `Entity` type, so that the type definitions are inferred correctly.

Sample usage:

```
import {
  createEntityAdapter,
  createSlice,
  configureStore,

} from '@reduxjs/toolkit'

type Book = { bookId: string; title: string }

const booksAdapter = createEntityAdapter<Book>({
  // Assume IDs are stored in a field other than `book.id`
  selectId: (book) => book.bookId,
  // Keep the "all IDs" array sorted based on book titles
  sortComparer: (a, b) => a.title.localeCompare(b.title),
})

const booksSlice = createSlice({
  name: 'books',
  initialState: booksAdapter.getInitialState(),
  reducers: {
    // Can pass adapter functions directly as case reducers.  Because we're passing this
    // as a value, `createSlice` will auto-generate the `bookAdded` action type / creator
    bookAdded: booksAdapter.addOne,
    booksReceived(state, action) {
      // Or, call them as "mutating" helpers in a case reducer
      booksAdapter.setAll(state, action.payload.books)
    },
  },
})

const store = configureStore({
  reducer: {
    books: booksSlice.reducer,
  },
})

type RootState = ReturnType<typeof store.getState>

console.log(store.getState().books)
// { ids: [], entities: {} }

// Can create a set of memoized selectors based on the location of this entity state
const booksSelectors = booksAdapter.getSelectors<RootState>(
  (state) => state.books
)

// And then use the selectors to retrieve values
const allBooks = booksSelectors.selectAll(store.getState())
```

Parameters

`createEntityAdapter` accepts a single options object parameter, with two optional fields inside.

`selectId`

A function that accepts a single `Entity` instance, and returns the value of whatever unique ID field is inside. If not provided, the default implementation is `entity => entity.id`. If your `Entity` type keeps its unique ID values in a field other than `entity.id`, you **must** provide a `selectId` function.

`sortComparer`

A callback function that accepts two `Entity` instances, and should return a standard `Array.sort()` numeric result (1, 0, -1) to indicate their relative order for sorting.

If provided, the `state.ids` array will be kept in sorted order based on comparisons of the entity objects, so that mapping over the IDs array to retrieve entities by ID should result in a sorted array of entities.

If not provided, the `state.ids` array will not be sorted, and no guarantees are made about the ordering. In other words, `state.ids` can be expected to behave like a standard Javascript array.

Note that sorting only kicks in when state is changed via one of the CRUD functions below (for example, `addOne()`, `updateMany()`).

Return Value

A "entity adapter" instance. An entity adapter is a plain JS object (not a class) containing the generated reducer functions, the original provided `selectId` and `sortComparer` callbacks, a method to generate an initial "entity state" value, and functions to generate a set of globalized and non-globalized memoized selector functions for this entity type.

The adapter instance will include the following methods (additional referenced TypeScript types included):

```
export type EntityId = number | string

export type Comparer<T> = (a: T, b: T) => number

export type IdSelector<T> = (model: T) => EntityId

export interface DictionaryNum<T> {
  [id: number]: T | undefined
}

export interface Dictionary<T> extends DictionaryNum<T> {
  [id: string]: T | undefined
}

export type Update<T> = { id: EntityId; changes: Partial<T> }

export interface EntityState<T> {
  ids: EntityId[]
  entities: Dictionary<T>
}

export interface EntityDefinition<T> {
  selectId: IdSelector<T>
  sortComparer: false | Comparer<T>
}

export interface EntityStateAdapter<T> {
  addOne<S extends EntityState<T>>(state: S, entity: T): S
  addOne<S extends EntityState<T>>(state: S, action: PayloadAction<T>): S

  addMany<S extends EntityState<T>>(state: S, entities: T[]): S
  addMany<S extends EntityState<T>>(state: S, entities: PayloadAction<T[]>): S

  setAll<S extends EntityState<T>>(state: S, entities: T[]): S
  setAll<S extends EntityState<T>>(state: S, entities: PayloadAction<T[]>): S

  removeOne<S extends EntityState<T>>(state: S, key: EntityId): S
  removeOne<S extends EntityState<T>>(state: S, key: PayloadAction<EntityId>): S

  removeMany<S extends EntityState<T>>(state: S, keys: EntityId[]): S
  removeMany<S extends EntityState<T>>(
    state: S,
    keys: PayloadAction<EntityId[]>
  ): S

  removeAll<S extends EntityState<T>>(state: S): S

  updateOne<S extends EntityState<T>>(state: S, update: Update<T>): S
  updateOne<S extends EntityState<T>>(
    state: S,
    update: PayloadAction<Update<T>>
  ): S

  updateMany<S extends EntityState<T>>(state: S, updates: Update<T>[]): S
  updateMany<S extends EntityState<T>>(
    state: S,
    updates: PayloadAction<Update<T>[]>
  ): S

  upsertOne<S extends EntityState<T>>(state: S, entity: T): S
  upsertOne<S extends EntityState<T>>(state: S, entity: PayloadAction<T>): S

  upsertMany<S extends EntityState<T>>(state: S, entities: T[]): S
  upsertMany<S extends EntityState<T>>(
    state: S,
    entities: PayloadAction<T[]>
  ): S
}

export interface EntitySelectors<T, V> {
  selectIds: (state: V) => EntityId[]
  selectEntities: (state: V) => Dictionary<T>
  selectAll: (state: V) => T[]
  selectTotal: (state: V) => number
  selectById: (state: V, id: EntityId) => T | undefined
}

export interface EntityAdapter<T> extends EntityStateAdapter<T> {
  selectId: IdSelector<T>
  sortComparer: false | Comparer<T>
  getInitialState(): EntityState<T>
  getInitialState<S extends object>(state: S): EntityState<T> & S
  getSelectors(): EntitySelectors<T, EntityState<T>>
  getSelectors<V>(
    selectState: (state: V) => EntityState<T>
  ): EntitySelectors<T, V>
}
```

CRUD Functions

The primary content of an entity adapter is a set of generated reducer functions for adding, updating, and removing entity instances from an entity state object:

- `addOne`: accepts a single entity, and adds it if it's not already present.
- `addMany`: accepts an array of entities or an object in the shape of `Record<EntityId, T>`, and adds them if not already present.
- `setOne`: accepts a single entity and adds or replaces it

- `setMany`: accepts an array of entities or an object in the shape of `Record<EntityId, T>`, and adds or replaces them.
- `setAll`: accepts an array of entities or an object in the shape of `Record<EntityId, T>`, and replaces all existing entities with the values in the array.
- `removeOne`: accepts a single entity ID value, and removes the entity with that ID if it exists.
- `removeMany`: accepts an array of entity ID values, and removes each entity with those IDs if they exist.
- `removeAll`: removes all entities from the entity state object.
- `updateOne`: accepts an "update object" containing an entity ID and an object containing one or more new field values to update inside a `changes` field, and performs a shallow update on the corresponding entity.
- `updateMany`: accepts an array of update objects, and performs shallow updates on all corresponding entities.
- `upsertOne`: accepts a single entity. If an entity with that ID exists, it will perform a shallow update and the specified fields will be merged into the existing entity, with any matching fields overwriting the existing values. If the entity does not exist, it will be added.
- `upsertMany`: accepts an array of entities or an object in the shape of `Record<EntityId, T>` that will be shallowly upserted.

SHOULD I ADD, SET OR UPSERT MY ENTITY?

All three options will insert *new* entities into the list. However they differ in how they handle entities that already exist. If an entity **already exists**:

- `addOne` and `addMany` will do nothing with the new entity
- `setOne` and `setMany` will completely replace the old entity with the new one. This will also get rid of any properties on the entity that are not present in the new version of said entity.
- `upsertOne` and `upsertMany` will do a shallow copy to merge the old and new entities overwriting existing values, adding any that were not there and not touching properties not provided in the new entity.

Each method has a signature that looks like:

```
(state: EntityState<T>, argument: TypeOrPayloadAction<Argument<T>>) =>
EntityState<T>
```

In other words, they accept a state that looks like `{ids: [], entities: {}}`, and calculate and return a new state.

These CRUD methods may be used in multiple ways:

- They may be passed as case reducers directly to `createReducer` and `createSlice`.
- They may be used as "mutating" helper methods when called manually, such as a separate hand-written call to `addOne()` inside of an existing case reducer, if the `state` argument is actually an Immer `Draft` value.

- They may be used as immutable update methods when called manually, if the `state` argument is actually a plain JS object or array.

Note: These methods do *not* have corresponding Redux actions created - they are just standalone reducers / update logic. **It is entirely up to you to decide where and how to use these methods!** Most of the time, you will want to pass them to `createSlice` or use them inside another reducer.

Each method will check to see if the `state` argument is an Immer `Draft` or not. If it is a draft, the method will assume that it's safe to continue mutating that draft further. If it is not a draft, the method will pass the plain JS value to Immer's `createNextState()`, and return the immutably updated result value.

The `argument` may be either a plain value (such as a single `Entity` object for `addOne()` or an `Entity[]` array for `addMany()`, or a `PayloadAction` action object with that same value as `action.payload`. This enables using them as both helper functions and reducers.

Note on shallow updates: `updateOne`, `updateMany`, `upsertOne`, and `upsertMany` only perform shallow updates in a mutable manner. This means that if your update/upsert consists of an object that includes nested properties, the value of the incoming change will overwrite the **entire** existing nested object. This may be unintended behavior for your application. As a general rule, these methods are best used with normalized data that *do not* have nested properties.

`getInitialState`

Returns a new entity state object like `{ids: [], entities: {}}`.

It accepts an optional object as an argument. The fields in that object will be merged into the returned initial state value. For example, perhaps you want your slice to also track some loading state:

```
const booksSlice = createSlice({
  name: 'books',
  initialState: booksAdapter.getInitialState({
    loading: 'idle',
  }),
  reducers: {
    booksLoadingStarted(state, action) {
      // Can update the additional state field
      state.loading = 'pending'
    },
  },
})
```

Selector Functions

The entity adapter will contain a `getSelectors()` function that returns a set of selectors that know how to read the contents of an entity state object:

- `selectIds`: returns the `state.ids` array.
- `selectEntities`: returns the `state.entities` lookup table.
- `selectAll`: maps over the `state.ids` array, and returns an array of entities in the same order.
- `selectTotal`: returns the total number of entities being stored in this state.
- `selectById`: given the state and an entity ID, returns the entity with that ID or `undefined`.

Each selector function will be created using the `createSelector` function from Reselect, to enable memoizing calculation of the results.

Because selector functions are dependent on knowing where in the state tree this specific entity state object is kept, `getSelectors()` can be called in two ways:

- If called without any arguments, it returns an "unglobalized" set of selector functions that assume their `state` argument is the actual entity state object to read from.
- It may also be called with a selector function that accepts the entire Redux state tree and returns the correct entity state object.

For example, the entity state for a `Book` type might be kept in the Redux state tree as `state.books`. You can use `getSelectors()` to read from that state in two ways:

```
const store = configureStore({
  reducer: {
    books: booksReducer,
  },
})

const simpleSelectors = booksAdapter.getSelectors()
const globalizedSelectors = booksAdapter.getSelectors((state) => state.books)

// Need to manually pass the correct entity state object in to this selector
const bookIds = simpleSelectors.selectIds(store.getState().books)

// This selector already knows how to find the books entity state
const allBooks = globalizedSelectors.selectAll(store.getState())
```

Notes

Applying Multiple Updates

If `updateMany()` is called with multiple updates targeted to the same ID, they will be merged into a single update, with later updates overwriting the earlier ones.

For both `updateOne()` and `updateMany()`, changing the ID of one existing entity to match the ID of a second existing entity will cause the first to replace the second completely.

Examples

Exercising several of the CRUD methods and selectors:

```
import {
  createEntityAdapter,
  createSlice,
  configureStore,
} from '@reduxjs/toolkit'

// Since we don't provide `selectId`, it defaults to assuming `entity.id` is the right field
const booksAdapter = createEntityAdapter({
  // Keep the "all IDs" array sorted based on book titles
  sortComparer: (a, b) => a.title.localeCompare(b.title),
})

const booksSlice = createSlice({
  name: 'books',
  initialState: booksAdapter.getInitialState({
    loading: 'idle',
  }),
  reducers: {
    // Can pass adapter functions directly as case reducers.  Because we're passing this
    // as a value, `createSlice` will auto-generate the `bookAdded` action type / creator
    bookAdded: booksAdapter.addOne,
    booksLoading(state, action) {
      if (state.loading === 'idle') {
        state.loading = 'pending'
      }
    },
    booksReceived(state, action) {
      if (state.loading === 'pending') {
        // Or, call them as "mutating" helpers in a case reducer
        booksAdapter.setAll(state, action.payload)
        state.loading = 'idle'
      }
    },
    bookUpdated: booksAdapter.updateOne,
  },
})

const {
  bookAdded,
  booksLoading,
  booksReceived,
  bookUpdated,
} = booksSlice.actions

const store = configureStore({
  reducer: {
    books: booksSlice.reducer,
  },
})

// Check the initial state:
console.log(store.getState().books)
// {ids: [], entities: {}, loading: 'idle' }

const booksSelectors = booksAdapter.getSelectors((state) => state.books)

store.dispatch(bookAdded({ id: 'a', title: 'First' }))
```

```
console.log(store.getState().books)
// {ids: ["a"], entities: {a: {id: "a", title: "First"}}, loading: 'idle' }

store.dispatch(bookUpdated({ id: 'a', changes: { title: 'First (altered)' } }))
store.dispatch(booksLoading())
console.log(store.getState().books)
// {ids: ["a"], entities: {a: {id: "a", title: "First (altered)"}}, loading: 'pending' }

store.dispatch(
  booksReceived([
    { id: 'b', title: 'Book 3' },
    { id: 'c', title: 'Book 2' },
  ])
)

console.log(booksSelectors.selectIds(store.getState()))
// "a" was removed due to the `setAll()` call
// Since they're sorted by title, "Book 2" comes before "Book 3"
// ["c", "b"]

console.log(booksSelectors.selectAll(store.getState()))
// All book entries in sorted order
// [{id: "c", title: "Book 2"}, {id: "b", title: "Book 3"}]
```

*Last updated on **Apr 25, 2022***

`createSelector`

The `createSelector` utility from the Reselect library, re-exported for ease of use.

For more details on using `createSelector`, see:

- The Reselect API documentation
- React-Redux docs: Hooks API - Using memoizing selectors
- Idiomatic Redux: Using Reselect Selectors for Encapsulation and Performance
- React/Redux Links: Reducers and Selectors

Note: Prior to v0.7, RTK re-exported `createSelector` from `selectorator`, which allowed using string keypaths as input selectors. This was removed, as it ultimately did not provide enough benefits, and the string keypaths made static typing for selectors difficult.

`createDraftSafeSelector`

In general, we recommend against using selectors inside of reducers:

- Selectors typically expect the entire Redux state object as an argument, while slice reducers only have access to a specific subset of the entire Redux state
- Reselect's `createSelector` relies on reference comparisons to determine if inputs have changed, and if an Immer Proxy-wrapped draft value is passed in to a selector, the selector may see the same reference and think nothing has changed.

However, some users have requested the ability to create selectors that will work correctly inside of Immer-powered reducers. One use case for this might be collecting an ordered set of items when using `createEntityAdapter`, such as `const orderedTodos = todosSelectors.selectAll(todosState)`, and then using `orderedTodos` in the rest of the reducer logic.

Besides re-exporting `createSelector`, RTK also exports a wrapped version of `createSelector` named `createDraftSafeSelector` that allows you to create selectors that can safely be used inside of `createReducer` and `createSlice` reducers with Immer-powered mutable logic. When used with plain state values, the selector will still memoize normally based on the inputs. But, when used with Immer draft values, the selector will err on the side of recalculating the results, just to be safe.

All selectors created by `entityAdapter.getSelectors` are "draft safe" selectors by default.

Example:

```
const selectSelf = (state: State) => state
const unsafeSelector = createSelector(selectSelf, (state) => state.value)
const draftSafeSelector = createDraftSafeSelector(
  selectSelf,
  (state) => state.value
)

// in your reducer:

state.value = 1

const unsafe1 = unsafeSelector(state)
const safe1 = draftSafeSelector(state)

state.value = 2

const unsafe2 = unsafeSelector(state)
const safe2 = draftSafeSelector(state)
```

After executing that, `unsafe1` and `unsafe2` will be of the same value, because the memoized selector was executed on the same object - but `safe2` will actually be different from `safe1` (with the updated value of `2`), because the safe selector detected that it was executed on a Immer draft object and recalculated using the current value instead of returning a cached value.

Last updated on ***May 30, 2021***

Matching Utilities

Redux Toolkit exports several type-safe action matching utilities that you can leverage when checking for specific kinds of actions. These are primarily useful for the `builder.addMatcher()` cases in `createSlice` and `createReducer`, as well as when writing custom middleware.

General Purpose

- `isAllOf` - returns true when **all** conditions are met
- `isAnyOf` - returns true when **at least one of** the conditions are met

`createAsyncThunk`-specific matchers

All these matchers can either be called with one or more thunks as arguments, in which case they will return a matcher function for that condition and thunks, or with one actions, in which case they will match for any thunk action with said condition.

- `isAsyncThunkAction` - accepts one or more action creators and returns true when all match
- `isPending` - accepts one or more action creators and returns true when all match
- `isFulfilled` - accepts one or more action creators and returns true when all match
- `isRejected` - accepts one or more action creators and returns true when all match
- `isRejectedWithValue` - accepts one or more action creators and returns true when all match

`isAllOf`

A higher-order function that accepts one or more of:

- `redux-toolkit` action creator functions such as the ones produced by:
  - `createAction`
  - `createSlice`
  - `createAsyncThunk`
- type guard functions
- custom action creator functions that have a `.match` property that is a type guard

It will return a type guard function that returns `true` if *all* of the provided functions match.

`isAnyOf`

Accepts the same inputs as `isAllOf` and will return a type guard function that returns `true` if at least one of the provided functions match.

`isAsyncThunkAction`

A higher-order function that returns a type guard function that may be used to check whether an action was created by `createAsyncThunk`.

```
import { isAsyncThunkAction } from '@reduxjs/toolkit'
import type { AnyAction } from '@reduxjs/toolkit'
import { requestThunk1, requestThunk2 } from '@virtual/matchers'

const isARequestAction = isAsyncThunkAction(requestThunk1, requestThunk2)

function handleRequestAction(action: AnyAction) {
  if (isARequestAction(action)) {
    // action is an action dispatched by either `requestThunk1` or `requestThunk2`
  }
}
```

isPending

A higher-order function that returns a type guard function that may be used to check whether an action is a 'pending' action creator from the `createAsyncThunk` promise lifecycle.

```
import { isPending } from '@reduxjs/toolkit'
import type { AnyAction } from '@reduxjs/toolkit'
import { requestThunk1, requestThunk2 } from '@virtual/matchers'

const isAPendingAction = isPending(requestThunk1, requestThunk2)

function handlePendingAction(action: AnyAction) {
  if (isAPendingAction(action)) {
    // action is a pending action dispatched by either `requestThunk1` or `requestThunk2`
  }
}
```

isFulfilled

A higher-order function that returns a type guard function that may be used to check whether an action is a 'fulfilled'' action creator from the `createAsyncThunk` promise lifecycle.

```
import { isFulfilled } from '@reduxjs/toolkit'
import type { AnyAction } from '@reduxjs/toolkit'
import { requestThunk1, requestThunk2 } from '@virtual/matchers'

const isAFulfilledAction = isFulfilled(requestThunk1, requestThunk2)

function handleFulfilledAction(action: AnyAction) {
  if (isAFulfilledAction(action)) {
    // action is a fulfilled action dispatched by either `requestThunk1` or `requestThunk2`
  }
}
```

isRejected

A higher-order function that returns a type guard function that may be used to check whether an action is a 'rejected' action creator from the `createAsyncThunk` promise lifecycle.

```
import { isRejected } from '@reduxjs/toolkit'
import type { AnyAction } from '@reduxjs/toolkit'
import { requestThunk1, requestThunk2 } from '@virtual/matchers'

const isARejectedAction = isRejected(requestThunk1, requestThunk2)

function handleRejectedAction(action: AnyAction) {
  if (isARejectedAction(action)) {
    // action is a rejected action dispatched by either `requestThunk1` or `requestThunk2`
  }
}
```

`isRejectedWithValue`

A higher-order function that returns a type guard function that may be used to check whether an action is a 'rejected' action creator from the `createAsyncThunk` promise lifecycle that was created by `rejectWithValue`.

```
import { isRejectedWithValue } from '@reduxjs/toolkit'
import type { AnyAction } from '@reduxjs/toolkit'
import { requestThunk1, requestThunk2 } from '@virtual/matchers'

const isARejectedWithValueAction = isRejectedWithValue(
  requestThunk1,
  requestThunk2
)

function handleRejectedWithValueAction(action: AnyAction) {
  if (isARejectedWithValueAction(action)) {
    // action is a rejected action dispatched by either `requestThunk1` or `requestThunk2`
    // where rejectWithValue was used
  }
}
```

Using matchers to reduce code complexity, duplication and boilerplate

When using the `builder` pattern to construct a reducer, we add cases or matchers one at a time. However, by using `isAnyOf` or `isAllOf`, we're able to easily use the same matcher for several cases in a type-safe manner.

First, let's examine an unnecessarily complex example:

```
import { createAsyncThunk, createReducer } from '@reduxjs/toolkit'
import type { PayloadAction } from '@reduxjs/toolkit'

interface Data {
  isInteresting: boolean
  isSpecial: boolean
}

interface Special extends Data {
  isSpecial: true}
```

```
interface Interesting extends Data {
  isInteresting: true
}

function isSpecial(
  action: PayloadAction<Data>
): action is PayloadAction<Special> {
  return action.payload.isSpecial
}

function isInteresting(
  action: PayloadAction<Data>
): action is PayloadAction<Interesting> {
  return action.payload.isInteresting
}

interface ExampleState {
  isSpecial: boolean
  isInteresting: boolean
}

const initialState = {
  isSpecial: false,
  isInteresting: false,
} as ExampleState

export const isSpecialAndInterestingThunk = createAsyncThunk(
  'isSpecialAndInterestingThunk',
  () => {
    return {
      isSpecial: true,
      isInteresting: true,
    }
  }
)

// This has unnecessary complexity
const loadingReducer = createReducer(initialState, (builder) => {
  builder.addCase(isSpecialAndInterestingThunk.fulfilled, (state, action) => {
    if (isSpecial(action)) {
      state.isSpecial = true
    }
    if (isInteresting(action)) {
      state.isInteresting = true
    }
  })
})
```

In this scenario, we can use `isAllOf` to simplify our code and reduce some of the boilerplate.

```
import { createReducer, isAllOf } from '@reduxjs/toolkit'
import {
  isSpecialAndInterestingThunk,
  initialState,
  isSpecial,
  isInteresting,
} from '@virtual/matchers' // This is a fake pkg that provides the types shown above
import type { Data } from '@virtual/matchers' // This is a fake pkg that provides the types shown above

const loadingReducer = createReducer(initialState, (builder) => {
  builder
    .addMatcher(
      isAllOf(isSpecialAndInterestingThunk.fulfilled, isSpecial),
      (state, action) => {
        state.isSpecial = true
      }
    )
    .addMatcher(
      isAllOf(isSpecialAndInterestingThunk.fulfilled, isInteresting),
      (state, action) => {
        state.isInteresting = true
      }
    )
})
```

Using matchers as a TypeScript Type Guard

The function returned by `isAllOf` and `isAnyOf` can also be used as a TypeScript type guard in other contexts.

```
import { isAllOf } from '@reduxjs/toolkit'
import type { PayloadAction } from '@reduxjs/toolkit'
import { isSpecial, isInteresting } from '@virtual/matchers' // This is a fake pkg that provides the types shown above
import type { Data } from '@virtual/matchers' // This is a fake pkg that provides the types shown above

const isSpecialAndInteresting = isAllOf(isSpecial, isInteresting)

function someFunction(action: PayloadAction<Data>) {
  if (isSpecialAndInteresting(action)) {
    // "action" will be correctly typed as:
    // `PayloadAction<Special> & PayloadAction<Interesting>`
  }
}
```

```
import { isAnyOf } from '@reduxjs/toolkit'
import type { PayloadAction } from '@reduxjs/toolkit'
import { Data, isSpecial, isInteresting } from '@virtual/matchers' // this is a fake pkg that provides the types shown
above

const isSpecialOrInteresting = isAnyOf(isSpecial, isInteresting)

function someFunction(action: PayloadAction<Data>) {
  if (isSpecialOrInteresting(action)) {
    // "action" will be correctly typed as:
    // `PayloadAction<Special> | PayloadAction<Interesting>`
  }
}
```

*Last updated on **Jun 24, 2022***

Other Exports

Redux Toolkit exports some of its internal utilities, and re-exports additional functions from other dependencies as well.

`nanoid`

An inlined copy of `nanoid/nonsecure`. Generates a non-cryptographically-secure random ID string. `createAsyncThunk` uses this by default for request IDs. May also be useful for other cases as well.

```
import { nanoid } from '@reduxjs/toolkit'

console.log(nanoid())
// 'dgPXxUz_6fWIQBD8XmiSy'
```

`miniSerializeError`

The default error serialization function used by `createAsyncThunk`, based on https://github.com/sindresorhus/serialize-error. If its argument is an object (such as an `Error` instance), it returns a plain JS `SerializedError` object that copies over any of the listed fields. Otherwise, it returns a stringified form of the value: `{ message: String(value) }`.

```
export interface SerializedError {
  name?: string
  message?: string
  stack?: string
  code?: string
}

export function miniSerializeError(value: any): SerializedError {}
```

`copyWithStructuralSharing`

A utility that will recursively merge two similar objects together, preserving existing references if the values appear to be the same. This is used internally to help ensure that re-fetched data keeps using the same references unless the new data has actually changed, to avoid unnecessary re-renders. Otherwise, every re-fetch would likely cause the entire dataset to be replaced and all consuming components to always re-render.

If either of the inputs are not plain JS objects or arrays, the new value is returned.

```
export function copyWithStructuralSharing<T>(oldObj: any, newObj: T): T
export function copyWithStructuralSharing(oldObj: any, newObj: any): any {}
```

Exports from Other Libraries

`createNextState`

The default immutable update function from the `immer` library, re-exported here as `createNextState` (also commonly referred to as `produce`)

`current`

The `current` function from the `immer` library, which takes a snapshot of the current state of a draft and finalizes it (but without freezing). Current is a great utility to print the current state during debugging, and the output of `current` can also be safely leaked outside the producer.

Exports from Other Libraries

`createNextState`

The default immutable update function from the `immer` library, re-exported here as `createNextState` (also commonly referred to as `produce`)

`current`

The `current` function from the `immer` library, which takes a snapshot of the current state of a draft and finalizes it (but without freezing). Current is a great utility to print the current state during debugging, and the output of `current` can also be safely leaked outside the producer.

```
import { createReducer, createAction, current } from '@reduxjs/toolkit'

interface Todo {
  //...
}
const addTodo = createAction<Todo>('addTodo')

const initialState = [] as Todo[]

const todosReducer = createReducer(initialState, (builder) => {
  builder.addCase(addTodo, (state, action) => {
    state.push(action.payload)
    console.log(current(state))
  })
})
```

`original`

The `original` function from the `immer` library, which returns the original object. This is particularly useful for referential equality check in reducers.

`isDraft`

The `isDraft` function from the `immer` library, which checks to see if a given value is a Proxy-wrapped "draft" state.

`freeze`

The `freeze` function from the `immer` library, which freezes draftable objects.

`combineReducers`

Redux's `combineReducers`, re-exported for convenience. While `configureStore` calls this internally, you may wish to call it yourself to compose multiple levels of slice reducers.

`compose`

Redux's `compose`. It composes functions from right to left. This is a functional programming utility. You might want to use it to apply several store custom enhancers/ functions in a row.

`bindActionCreators`

Redux's `bindActionCreators`. It wraps action creators with `dispatch()` so that they dispatch immediately when called.

`createStore`

Redux's `createStore`. You should not need to use this directly.

`applyMiddleware`

Redux's `applyMiddleware`. You should not need to use this directly.

*Last updated on **Jan 30, 2023***

Codemods

Per the description in `1.9.0-alpha.0`, we plan to remove the "object" argument from `createReducer` and `createSlice.extraReducers` in the future RTK 2.0 major version. In `1.9.0-alpha.0`, we added a one-shot runtime warning to each of those APIs.

To simplify upgrading codebases, we've published a set of codemods that will automatically transform the deprecated "object" syntax into the equivalent "builder" syntax.

The codemods package is available on NPM as **`@reduxjs/rtk-codemods`**. It currently contains two codemods: `createReducerBuilder` and `createSliceBuilder`.

To run the codemods against your codebase, run `npx @reduxjs/rtk-codemods <TRANSFORM NAME> path/of/files/ or/some**/*glob.js`.

Examples:

```
npx @reduxjs/rtk-codemods createReducerBuilder ./src

npx @reduxjs/rtk-codemods createSliceBuilder ./packages/my-app/**/*.ts
```

We also recommend re-running Prettier on the codebase before committing the changes.

These codemods *should* work, but we would greatly appreciate testing and feedback on more real-world codebases!

Before:

```
createReducer(initialState, {
  [todoAdded1a]: (state, action) => {
    // stuff
  },
  [todoAdded1b]: (state, action) => action.payload,
})

const slice1 = createSlice({
  name: 'a',
  initialState: {},
  extraReducers: {
    [todoAdded1a]: (state, action) => {
      // stuff
    },
    [todoAdded1b]: (state, action) => action.payload,
  },
})
```

After:

```
createReducer(initialState, (builder) => {
  builder.addCase(todoAdded1a, (state, action) => {
    // stuff
  })

  builder.addCase(todoAdded1b, (state, action) => action.payload)
})

const slice1 = createSlice({
  name: 'a',
  initialState: {},

  extraReducers: (builder) => {
    builder.addCase(todoAdded1a, (state, action) => {
      // stuff
    })

    builder.addCase(todoAdded1b, (state, action) => action.payload)
  },
})
```

Last updated on ***Oct 26, 2022***