Async Actions · Redux

Async Actions

In the basics guide, we built a simple todo application. It was fully synchronous. Every time an action was dispatched, the state was updated immediately.

In this guide, we will build a different, asynchronous application. It will use the Reddit API to show the current headlines for a selected subreddit. How does asynchronicity fit into Redux flow?

Actions

When you call an asynchronous API, there are two crucial moments in time: the moment you start the call, and the moment when you receive an answer (or a timeout).

Each of these two moments usually require a change in the application state; to do that, you need to dispatch normal actions that will be processed by reducers synchronously. Usually, for any API request you'll want to dispatch at least three different kinds of actions:

- **An action informing the reducers that the request began.**
  The reducers may handle this action by toggling an `isFetching` flag in the state. This way the UI knows it's time to show a spinner.

- **An action informing the reducers that the request finished successfully.**
  The reducers may handle this action by merging the new data into the state they manage and resetting `isFetching`. The UI would hide the spinner, and display the fetched data.

- **An action informing the reducers that the request failed.**
  The reducers may handle this action by resetting `isFetching`. Additionally, some reducers may want to store the error message so the UI can display it.

You may use a dedicated `status` field in your actions:

```
{ type: 'FETCH_POSTS' }
{ type: 'FETCH_POSTS', status: 'error', error: 'Oops' }
{ type: 'FETCH_POSTS', status: 'success', response: { ... } }
```

Or you can define separate types for them:

```
{ type: 'FETCH_POSTS_REQUEST' }
{ type: 'FETCH_POSTS_FAILURE', error: 'Oops' }
{ type: 'FETCH_POSTS_SUCCESS', response: { ... } }
```

Choosing whether to use a single action type with flags, or multiple action types, is up to you. It's a convention you need to decide with your team. Multiple types leave less room for a mistake, but this is not an issue if you generate action creators and reducers with a helper library like redux-actions.

Whatever convention you choose, stick with it throughout the application.

We'll use separate types in this tutorial.

Synchronous Action Creators

Let's start by defining the several synchronous action types and action creators we need in our example app. Here, the user can select a subreddit to display:

`actions.js`

```
export const SELECT_SUBREDDIT = 'SELECT_SUBREDDIT'

export function selectSubreddit(subreddit) {
  return {
    type: SELECT_SUBREDDIT,
    subreddit
  }
}
```

They can also press a “refresh” button to update it:

```
export const INVALIDATE_SUBREDDIT = 'INVALIDATE_SUBREDDIT'

export function invalidateSubreddit(subreddit) {
  return {
    type: INVALIDATE_SUBREDDIT,
    subreddit
  }
}
```

These were the actions governed by the user interaction. We will also have another kind of action, governed by the network requests. We will see how to dispatch them later, but for now, we just want to

define them.

When it's time to fetch the posts for some subreddit, we will dispatch a `REQUEST_POSTS` action:

```
export const REQUEST_POSTS = 'REQUEST_POSTS'

export function requestPosts(subreddit) {
  return {
    type: REQUEST_POSTS,
    subreddit
  }
}
```

It is important for it to be separate from `SELECT_SUBREDDIT` or `INVALIDATE_SUBREDDIT`. While they may occur one after another, as the app grows more complex, you might want to fetch some data independently of the user action (for example, to prefetch the most popular subreddits, or to refresh stale data once in a while). You may also want to fetch in response to a route change, so it's not wise to couple fetching to some particular UI event early on.

Finally, when the network request comes through, we will dispatch `RECEIVE_POSTS`:

```
export const RECEIVE_POSTS = 'RECEIVE_POSTS'

export function receivePosts(subreddit, json) {
  return {
    type: RECEIVE_POSTS,
    subreddit,
    posts: json.data.children.map(child => child.data),
    receivedAt: Date.now()
  }
}
```

This is all we need to know for now. The particular mechanism to dispatch these actions together with network requests will be discussed later.

> **Note on Error Handling**
>
> In a real app, you'd also want to dispatch an action on request failure. We won't implement error handling in this tutorial, but the real world example shows one of the possible approaches.

Designing the State Shape

Just like in the basic tutorial, you'll need to design the shape of your application's state before rushing into the implementation. With asynchronous code, there is more state to take care of, so we need to think it through.

This part is often confusing to beginners, because it is not immediately clear what information describes the state of an asynchronous application, and how to organize it in a single tree.

We'll start with the most common use case: lists. Web applications often show lists of things. For example, a list of posts, or a list of friends. You'll need to figure out what sorts of lists your app can show. You want to store them separately in the state, because this way you can cache them and only fetch again if necessary.

Here's what the state shape for our “Reddit headlines” app might look like:

```
{
  selectedSubreddit: 'frontend',
  postsBySubreddit: {
    frontend: {
      isFetching: true,
      didInvalidate: false,
      items: []
    },
    reactjs: {
      isFetching: false,
      didInvalidate: false,
      lastUpdated: 1439478405547,
      items: [
        {
          id: 42,
          title: 'Confusion about Flux and Relay'
        },
        {
          id: 500,
          title: 'Creating a Simple Application Using React JS and Flux
Architecture'
        }
      ]
    }
  }
}
```

There are a few important bits here:

- We store each subreddit's information separately so we can cache every subreddit. When the user switches between them the second time, the update will be instant, and we won't need to refetch unless we want to. Don't worry about all these items being in memory: unless you're dealing with tens of thousands of items, and your user rarely closes the tab, you won't need any sort of cleanup.

- For every list of items, you'll want to store `isFetching` to show a spinner, `didInvalidate` so you can later toggle it when the data is stale, `lastUpdated` so you know when it was fetched the last time, and the `items` themselves. In a real app, you'll also want to store pagination state like `fetchedPageCount` and `nextPageUrl`.

> **Note on Nested Entities**
>
> In this example, we store the received items together with the pagination information. However, this approach won't work well if you have nested entities referencing each other, or if you let the user edit items. Imagine the user wants to edit a fetched post, but this post is duplicated in several places in the state tree. This would be really painful to implement.
>
> If you have nested entities, or if you let users edit received entities, you should keep them separately in the state as if it was a database. In pagination information, you would only refer to them by their IDs. This lets you always keep them up to date. The real world example shows this approach, together with normalizr to normalize the nested API responses. With this approach, your state might look like this:

```
{
  selectedSubreddit: 'frontend',
  entities: {
    users: {
      2: {
        id: 2,
        name: 'Andrew'
      }
    },
    posts: {
      42: {
        id: 42,
        title: 'Confusion about Flux and Relay',
        author: 2
      },
      100: {
        id: 100,
```

```
        title: 'Creating a Simple Application Using React JS and Flux
Architecture',
        author: 2
      }
    }
  },
  postsBySubreddit: {
    frontend: {
      isFetching: true,
      didInvalidate: false,
      items: []
    },
    reactjs: {
      isFetching: false,
      didInvalidate: false,
      lastUpdated: 1439478405547,
      items: [ 42, 100 ]
    }
  }
}
```

In this guide, we won't normalize entities, but it's something you should consider for a more dynamic application.

Handling Actions

Before going into the details of dispatching actions together with network requests, we will write the reducers for the actions we defined above.

> **Note on Reducer Composition**
>
> Here, we assume that you understand reducer composition with `combineReducers()`, as described in the Splitting Reducers section on the basics guide. If you don't, please read it first.

`reducers.js`

```
import { combineReducers } from 'redux'
import {
  SELECT_SUBREDDIT, INVALIDATE_SUBREDDIT,
  REQUEST_POSTS, RECEIVE_POSTS
} from '../actions'
```

```
function selectedSubreddit(state = 'reactjs', action) {
  switch (action.type) {
    case SELECT_SUBREDDIT:
      return action.subreddit
    default:
      return state
  }
}

function posts(state = {
  isFetching: false,
  didInvalidate: false,
  items: []
}, action) {
  switch (action.type) {
    case INVALIDATE_SUBREDDIT:
      return Object.assign({}, state, {
        didInvalidate: true
      })
    case REQUEST_POSTS:
      return Object.assign({}, state, {
        isFetching: true,
        didInvalidate: false
      })
    case RECEIVE_POSTS:
      return Object.assign({}, state, {
        isFetching: false,
        didInvalidate: false,
        items: action.posts,
        lastUpdated: action.receivedAt
      })
    default:
      return state
  }
}

function postsBySubreddit(state = {}, action) {
  switch (action.type) {
    case INVALIDATE_SUBREDDIT:
    case RECEIVE_POSTS:
    case REQUEST_POSTS:
```

```
      return Object.assign({}, state, {
        [action.subreddit]: posts(state[action.subreddit], action)
      })
    default:
      return state
  }
}

const rootReducer = combineReducers({
  postsBySubreddit,
  selectedSubreddit
})

export default rootReducer
```

In this code, there are two interesting parts:

- We use ES6 computed property syntax so we can update `state[action.subreddit]` with `Object.assign()` in a terse way. This:

  ```
  return Object.assign({}, state, {
    [action.subreddit]: posts(state[action.subreddit], action)
  })
  ```

  is equivalent to this:

  ```
  let nextState = {}
  nextState[action.subreddit] = posts(state[action.subreddit], action)
  return Object.assign({}, state, nextState)
  ```

  - We extracted `posts(state, action)` that manages the state of a specific post list. This is just reducer composition! It is our choice how to split the reducer into smaller reducers, and in this case, we're delegating updating items inside an object to a `posts` reducer. The real world example goes even further, showing how to create a reducer factory for parameterized pagination reducers.

Remember that reducers are just functions, so you can use functional composition and higher-order functions as much as you feel comfortable.

Async Action Creators

Finally, how do we use the synchronous action creators we defined earlier together with network

requests? The standard way to do it with Redux is to use the Redux Thunk middleware. It comes in a separate package called `redux-thunk`. We'll explain how middleware works in general later; for now, there is just one important thing you need to know: by using this specific middleware, an action creator can return a function instead of an action object. This way, the action creator becomes a thunk.

When an action creator returns a function, that function will get executed by the Redux Thunk middleware. This function doesn't need to be pure; it is thus allowed to have side effects, including executing asynchronous API calls. The function can also dispatch actions—like those synchronous actions we defined earlier.

We can still define these special thunk action creators inside our `actions.js` file:

`actions.js`

```
import fetch from 'isomorphic-fetch'

export const REQUEST_POSTS = 'REQUEST_POSTS'
function requestPosts(subreddit) {
  return {
    type: REQUEST_POSTS,
    subreddit
  }
}

export const RECEIVE_POSTS = 'RECEIVE_POSTS'
function receivePosts(subreddit, json) {
  return {
    type: RECEIVE_POSTS,
    subreddit,
    posts: json.data.children.map(child => child.data),
    receivedAt: Date.now()
  }
}

// Meet our first thunk action creator!
// Though its insides are different, you would use it just like any other action
creator:
// store.dispatch(fetchPosts('reactjs'))

export function fetchPosts(subreddit) {

  // Thunk middleware knows how to handle functions.
```

```
  // It passes the dispatch method as an argument to the function,
  // thus making it able to dispatch actions itself.

  return function (dispatch) {

    // First dispatch: the app state is updated to inform
    // that the API call is starting.

    dispatch(requestPosts(subreddit))

    // The function called by the thunk middleware can return a value,
    // that is passed on as the return value of the dispatch method.

    // In this case, we return a promise to wait for.
    // This is not required by thunk middleware, but it is convenient for us.

    return fetch(`http://www.reddit.com/r/${subreddit}.json`)
      .then(response => response.json())
      .then(json =>

        // We can dispatch many times!
        // Here, we update the app state with the results of the API call.

        dispatch(receivePosts(subreddit, json))
      )

      // In a real world app, you also want to
      // catch any error in the network call.
  }
}
```

Note on `fetch`

We use `fetch` API in the examples. It is a new API for making network requests that replaces `XMLHttpRequest` for most common needs. Because most browsers don't yet support it natively, we suggest that you use `isomorphic-fetch` library:

```
// Do this in every file where you use `fetch`
import fetch from 'isomorphic-fetch'
```

Internally, it uses `whatwg-fetch` polyfill on the client, and `node-fetch` on the server, so you won't

> need to change API calls if you change your app to be universal.
>
> Be aware that any `fetch` polyfill assumes a Promise polyfill is already present. The easiest way to ensure you have a Promise polyfill is to enable Babel's ES6 polyfill in your entry point before any other code runs:
>
> ```
> // Do this once before any other code in your app
> import 'babel-polyfill'
> ```

How do we include the Redux Thunk middleware in the dispatch mechanism? We use the `applyMiddleware()` store enhancer from Redux, as shown below:

`index.js`

```
import thunkMiddleware from 'redux-thunk'
import createLogger from 'redux-logger'
import { createStore, applyMiddleware } from 'redux'
import { selectSubreddit, fetchPosts } from './actions'
import rootReducer from './reducers'

const loggerMiddleware = createLogger()

const store = createStore(
  rootReducer,
  applyMiddleware(
    thunkMiddleware, // lets us dispatch() functions
    loggerMiddleware // neat middleware that logs actions
  )
)

store.dispatch(selectSubreddit('reactjs'))
store.dispatch(fetchPosts('reactjs')).then(() =>
  console.log(store.getState())
)
```

The nice thing about thunks is that they can dispatch results of each other:

`actions.js`

```
import fetch from 'isomorphic-fetch'

export const REQUEST_POSTS = 'REQUEST_POSTS'
function requestPosts(subreddit) {
```

```
  return {
    type: REQUEST_POSTS,
    subreddit
  }
}

export const RECEIVE_POSTS = 'RECEIVE_POSTS'
function receivePosts(subreddit, json) {
  return {
    type: RECEIVE_POSTS,
    subreddit,
    posts: json.data.children.map(child => child.data),
    receivedAt: Date.now()
  }
}

function fetchPosts(subreddit) {
  return dispatch => {
    dispatch(requestPosts(subreddit))
    return fetch(`http://www.reddit.com/r/${subreddit}.json`)
      .then(response => response.json())
      .then(json => dispatch(receivePosts(subreddit, json)))
  }
}

function shouldFetchPosts(state, subreddit) {
  const posts = state.postsBySubreddit[subreddit]
  if (!posts) {
    return true
  } else if (posts.isFetching) {
    return false
  } else {
    return posts.didInvalidate
  }
}

export function fetchPostsIfNeeded(subreddit) {

  // Note that the function also receives getState()
  // which lets you choose what to dispatch next.
```

```
  // This is useful for avoiding a network request if
  // a cached value is already available.

  return (dispatch, getState) => {
    if (shouldFetchPosts(getState(), subreddit)) {
      // Dispatch a thunk from thunk!
      return dispatch(fetchPosts(subreddit))
    } else {
      // Let the calling code know there's nothing to wait for.
      return Promise.resolve()
    }
  }
}
```

This lets us write more sophisticated async control flow gradually, while the consuming code can stay pretty much the same:

`index.js`

```
store.dispatch(fetchPostsIfNeeded('reactjs')).then(() =>
  console.log(store.getState())
)
```

> **Note about Server Rendering**
>
> Async action creators are especially convenient for server rendering. You can create a store, dispatch a single async action creator that dispatches other async action creators to fetch data for a whole section of your app, and only render after the Promise it returns, completes. Then your store will already be hydrated with the state you need before rendering.

Thunk middleware isn't the only way to orchestrate asynchronous actions in Redux:

- You can use redux-promise or redux-promise-middleware to dispatch Promises instead of functions.
- You can use redux-observable to dispatch Observables.
- You can use the redux-saga middleware to build more complex asynchronous actions.
- You can even write a custom middleware to describe calls to your API, like the real world example does.

It is up to you to try a few options, choose a convention you like, and follow it, whether with, or without the middleware.

Connecting to UI

Dispatching async actions is no different from dispatching synchronous actions, so we won't discuss this in detail. See Usage with React for an introduction into using Redux from React components. See Example: Reddit API for the complete source code discussed in this example.

Next Steps

Read Async Flow to recap how async actions fit into the Redux flow.