Reducers · Redux

Reducers

Actions describe the fact that *something happened*, but don't specify how the application's state changes in response. This is the job of a reducer.

Designing the State Shape

In Redux, all application state is stored as a single object. It's a good idea to think of its shape before writing any code. What's the minimal representation of your app's state as an object?

For our todo app, we want to store two different things:

- The currently selected visibility filter;
- The actual list of todos.

You'll often find that you need to store some data, as well as some UI state, in the state tree. This is fine, but try to keep the data separate from the UI state.

```
{
  visibilityFilter: 'SHOW_ALL',
  todos: [
    {
      text: 'Consider using Redux',
      completed: true,
    },
    {
      text: 'Keep all state in a single tree',
      completed: false
    }
  ]
}
```

> **Note on Relationships**
>
> In a more complex app, you're going to want different entities to reference each other. We suggest that you keep your state as normalized as possible, without any nesting. Keep every entity in an object stored with an ID as a key, and use IDs to reference it from other entities, or lists. Think of the app's state as a database. This approach is described in normalizr's documentation in detail. For

> example, keeping `todosById: { id -> todo }` and `todos: array<id>` inside the state would be a better idea in a real app, but we're keeping the example simple.

Handling Actions

Now that we've decided what our state object looks like, we're ready to write a reducer for it. The reducer is a pure function that takes the previous state and an action, and returns the next state.

```
(previousState, action) => newState
```

It's called a reducer because it's the type of function you would pass to `Array.prototype.reduce(reducer, ?initialValue)`. It's very important that the reducer stays pure. Things you should **never** do inside a reducer:

- Mutate its arguments;
- Perform side effects like API calls and routing transitions;
- Call non-pure functions, e.g. `Date.now()` or `Math.random()`.

We'll explore how to perform side effects in the advanced walkthrough. For now, just remember that the reducer must be pure. **Given the same arguments, it should calculate the next state and return it. No surprises. No side effects. No API calls. No mutations. Just a calculation.**

With this out of the way, let's start writing our reducer by gradually teaching it to understand the actions we defined earlier.

We'll start by specifying the initial state. Redux will call our reducer with an `undefined` state for the first time. This is our chance to return the initial state of our app:

```
import { VisibilityFilters } from './actions'

const initialState = {
  visibilityFilter: VisibilityFilters.SHOW_ALL,
  todos: []
}

function todoApp(state, action) {
  if (typeof state === 'undefined') {
    return initialState
  }

  // For now, don't handle any actions
```

```
  // and just return the state given to us.
  return state
}
```

One neat trick is to use the ES6 default arguments syntax to write this in a more compact way:

```
function todoApp(state = initialState, action) {
  // For now, don't handle any actions
  // and just return the state given to us.
  return state
}
```

Now let's handle `SET_VISIBILITY_FILTER`. All it needs to do is to change `visibilityFilter` on the state. Easy:

```
function todoApp(state = initialState, action) {
  switch (action.type) {
    case SET_VISIBILITY_FILTER:
      return Object.assign({}, state, {
        visibilityFilter: action.filter
      })
    default:
      return state
  }
}
```

Note that:

1. **We don't mutate the `state`.** We create a copy with `Object.assign()`. `Object.assign(state, { visibilityFilter: action.filter })` is also wrong: it will mutate the first argument. You **must** supply an empty object as the first parameter. You can also enable the object spread operator proposal to write `{ ...state, ...newState }` instead.

2. **We return the previous `state` in the `default` case.** It's important to return the previous `state` for any unknown action.

> **Note on `Object.assign`**
>
> `Object.assign()` is a part of ES6, but is not implemented by most browsers yet. You'll need to either use a polyfill, a Babel plugin, or a helper from another library like `_.assign()`.
>
> **Note on `switch` and Boilerplate**

> The `switch` statement is *not* the real boilerplate. The real boilerplate of Flux is conceptual: the need to emit an update, the need to register the Store with a Dispatcher, the need for the Store to be an object (and the complications that arise when you want a universal app). Redux solves these problems by using pure reducers instead of event emitters.
>
> It's unfortunate that many still choose a framework based on whether it uses `switch` statements in the documentation. If you don't like `switch`, you can use a custom `createReducer` function that accepts a handler map, as shown in "reducing boilerplate".

Handling More Actions

We have two more actions to handle! Let's extend our reducer to handle `ADD_TODO`.

```
function todoApp(state = initialState, action) {
  switch (action.type) {
    case SET_VISIBILITY_FILTER:
      return Object.assign({}, state, {
        visibilityFilter: action.filter
      })
    case ADD_TODO:
      return Object.assign({}, state, {
        todos: [
          ...state.todos,
          {
            text: action.text,
            completed: false
          }
        ]
      })
    default:
      return state
  }
}
```

Just like before, we never write directly to `state` or its fields, and instead we return new objects. The new `todos` is equal to the old `todos` concatenated with a single new item at the end. The fresh todo was constructed using the data from the action.

Finally, the implementation of the `TOGGLE_TODO` handler shouldn't come as a complete surprise:

```
case TOGGLE_TODO:
```

```
  return Object.assign({}, state, {
    todos: state.todos.map((todo, index) => {
      if (index === action.index) {
        return Object.assign({}, todo, {
          completed: !todo.completed
        })
      }
      return todo
    })
  })
```

Because we want to update a specific item in the array without resorting to mutations, we have to create a new array with the same items except the item at the index. If you find yourself often writing such operations, it's a good idea to use a helper like react-addons-update, updeep, or even a library like Immutable that has native support for deep updates. Just remember to never assign to anything inside the `state` unless you clone it first.

Splitting Reducers

Here is our code so far. It is rather verbose:

```
function todoApp(state = initialState, action) {
  switch (action.type) {
    case SET_VISIBILITY_FILTER:
      return Object.assign({}, state, {
        visibilityFilter: action.filter
      })
    case ADD_TODO:
      return Object.assign({}, state, {
        todos: [
          ...state.todos,
          {
            text: action.text,
            completed: false
          }
        ]
      })
    case TOGGLE_TODO:
      return Object.assign({}, state, {
        todos: state.todos.map((todo, index) => {
          if(index === action.index) {
```

```
            return Object.assign({}, todo, {
              completed: !todo.completed
            })
          }
          return todo
        })
      })
    default:
      return state
  }
}
```

Is there a way to make it easier to comprehend? It seems like `todos` and `visibilityFilter` are updated completely independently. Sometimes state fields depend on one another and more consideration is required, but in our case we can easily split updating `todos` into a separate function:

```
function todos(state = [], action) {
  switch (action.type) {
    case ADD_TODO:
      return [
        ...state,
        {
          text: action.text,
          completed: false
        }
      ]
    case TOGGLE_TODO:
      return state.map((todo, index) => {
        if (index === action.index) {
          return Object.assign({}, todo, {
            completed: !todo.completed
          })
        }
        return todo
      })
    default:
      return state
  }
}

function todoApp(state = initialState, action) {
```

```
  switch (action.type) {
    case SET_VISIBILITY_FILTER:
      return Object.assign({}, state, {
        visibilityFilter: action.filter
      })
    case ADD_TODO:
    case TOGGLE_TODO:
      return Object.assign({}, state, {
        todos: todos(state.todos, action)
      })
    default:
      return state
  }
}
```

Note that `todos` also accepts `state`—but it's an array! Now `todoApp` just gives it the slice of the state to manage, and `todos` knows how to update just that slice. **This is called *reducer composition*, and it's the fundamental pattern of building Redux apps.**

Let's explore reducer composition more. Can we also extract a reducer managing just `visibilityFilter`? We can:

```
function visibilityFilter(state = SHOW_ALL, action) {
  switch (action.type) {
    case SET_VISIBILITY_FILTER:
      return action.filter
    default:
      return state
  }
}
```

Now we can rewrite the main reducer as a function that calls the reducers managing parts of the state, and combines them into a single object. It also doesn't need to know the complete initial state anymore. It's enough that the child reducers return their initial state when given `undefined` at first.

```
function todos(state = [], action) {
  switch (action.type) {
    case ADD_TODO:
      return [
        ...state,
        {
```

```
          text: action.text,
          completed: false
        }
      ]
    case TOGGLE_TODO:
      return state.map((todo, index) => {
        if (index === action.index) {
          return Object.assign({}, todo, {
            completed: !todo.completed
          })
        }
        return todo
      })
    default:
      return state
  }
}

function visibilityFilter(state = SHOW_ALL, action) {
  switch (action.type) {
    case SET_VISIBILITY_FILTER:
      return action.filter
    default:
      return state
  }
}

function todoApp(state = {}, action) {
  return {
    visibilityFilter: visibilityFilter(state.visibilityFilter, action),
    todos: todos(state.todos, action)
  }
}
```

Note that each of these reducers is managing its own part of the global state. The `state` parameter is different for every reducer, and corresponds to the part of the state it manages. This is already looking good! When the app is larger, we can split the reducers into separate files and keep them completely independent and managing different data domains.

Finally, Redux provides a utility called `combineReducers()` that does the same boilerplate logic that the `todoApp` above currently does. With its help, we can rewrite `todoApp` like this:

```
import { combineReducers } from 'redux'

const todoApp = combineReducers({
  visibilityFilter,
  todos
})

export default todoApp
```

Note that this is completely equivalent to:

```
export default function todoApp(state = {}, action) {
  return {
    visibilityFilter: visibilityFilter(state.visibilityFilter, action),
    todos: todos(state.todos, action)
  }
}
```

You could also give them different keys, or call functions differently. These two ways to write a combined reducer are completely equivalent:

```
const reducer = combineReducers({
  a: doSomethingWithA,
  b: processB,
  c: c
})
```

```
function reducer(state = {}, action) {
  return {
    a: doSomethingWithA(state.a, action),
    b: processB(state.b, action),
    c: c(state.c, action)
  }
}
```

All `combineReducers()` does is generate a function that calls your reducers **with the slices of state selected according to their keys**, and combining their results into a single object again. It's not magic.

> **Note for ES6 Savvy Users**
>
> Because `combineReducers` expects an object, we can put all top-level reducers into a separate file,

> `export` each reducer function, and use `import * as reducers` to get them as an object with their names as the keys:
>
> ```
> import { combineReducers } from 'redux'
> import * as reducers from './reducers'
>
> const todoApp = combineReducers(reducers)
> ```
>
> Because `import *` is still new syntax, we don't use it anymore in the documentation to avoid confusion, but you may encounter it in some community examples.

Source Code

`reducers.js`

```
import { combineReducers } from 'redux'
import { ADD_TODO, TOGGLE_TODO, SET_VISIBILITY_FILTER, VisibilityFilters } from
'./actions'
const { SHOW_ALL } = VisibilityFilters

function visibilityFilter(state = SHOW_ALL, action) {
  switch (action.type) {
    case SET_VISIBILITY_FILTER:
      return action.filter
    default:
      return state
  }
}

function todos(state = [], action) {
  switch (action.type) {
    case ADD_TODO:
      return [
        ...state,
        {
          text: action.text,
          completed: false
        }
      ]
    case TOGGLE_TODO:
      return state.map((todo, index) => {
```

```
        if (index === action.index) {
          return Object.assign({}, todo, {
            completed: !todo.completed
          })
        }
        return todo
      })
    default:
      return state
  }
}

const todoApp = combineReducers({
  visibilityFilter,
  todos
})

export default todoApp
```

Next Steps

Next, we'll explore how to create a Redux store that holds the state and takes care of calling your reducer when you dispatch an action.