Data Flow · Redux

Data Flow

Redux architecture revolves around a **strict unidirectional data flow**.

This means that all data in an application follows the same lifecycle pattern, making the logic of your app more predictable and easier to understand. It also encourages data normalization, so that you don't end up with multiple, independent copies of the same data that are unaware of one another.

If you're still not convinced, read Motivation and The Case for Flux for a compelling argument in favor of unidirectional data flow. Although Redux is not exactly Flux, it shares the same key benefits.

The data lifecycle in any Redux app follows these 4 steps:

1. **You call** `store.dispatch(action)`.

   An action is a plain object describing *what happened*. For example:

   ```
   { type: 'LIKE_ARTICLE', articleId: 42 }
   { type: 'FETCH_USER_SUCCESS', response: { id: 3, name: 'Mary' } }
   { type: 'ADD_TODO', text: 'Read the Redux docs.' }
   ```

   Think of an action as a very brief snippet of news. “Mary liked article 42.” or “‘Read the Redux docs.' was added to the list of todos.”

   You can call `store.dispatch(action)` from anywhere in your app, including components and XHR callbacks, or even at scheduled intervals.

2. **The Redux store calls the reducer function you gave it.**
   The store will pass two arguments to the reducer: the current state tree and the action. For example, in the todo app, the root reducer might receive something like this:

   ```
   // The current application state (list of todos and chosen filter)
   let previousState = {
     visibleTodoFilter: 'SHOW_ALL',
     todos: [
       {
         text: 'Read the docs.',
         complete: false
   ```

```
    }
  ]
}

// The action being performed (adding a todo)
let action = {
  type: 'ADD_TODO',
  text: 'Understand the flow.'
}

// Your reducer returns the next application state
let nextState = todoApp(previousState, action)
```

Note that a reducer is a pure function. It only *computes* the next state. It should be completely predictable: calling it with the same inputs many times should produce the same outputs. It shouldn't perform any side effects like API calls or router transitions. These should happen before an action is dispatched.

3. **The root reducer may combine the output of multiple reducers into a single state tree.**
   How you structure the root reducer is completely up to you. Redux ships with a `combineReducers()` helper function, useful for "splitting" the root reducer into separate functions that each manage one branch of the state tree.

   Here's how `combineReducers()` works. Let's say you have two reducers, one for a list of todos, and another for the currently selected filter setting:

```
function todos(state = [], action) {
  // Somehow calculate it...
  return nextState
}

function visibleTodoFilter(state = 'SHOW_ALL', action) {
  // Somehow calculate it...
  return nextState
}

let todoApp = combineReducers({
  todos,
  visibleTodoFilter
})
```

When you emit an action, `todoApp` returned by `combineReducers` will call both reducers:

```
let nextTodos = todos(state.todos, action)
let nextVisibleTodoFilter = visibleTodoFilter(state.visibleTodoFilter, action)
```

It will then combine both sets of results into a single state tree:

```
return {
  todos: nextTodos,
  visibleTodoFilter: nextVisibleTodoFilter
}
```

While `combineReducers()` is a handy helper utility, you don't have to use it; feel free to write your own root reducer!

4. **The Redux store saves the complete state tree returned by the root reducer.**
This new tree is now the next state of your app! Every listener registered with `store.subscribe(listener)` will now be invoked; listeners may call `store.getState()` to get the current state.

Now, the UI can be updated to reflect the new state. If you use bindings like React Redux, this is the point at which `component.setState(newState)` is called.