Redux

An elegant way to manage application states

Problem

- As the requirements for JavaScript single-page applications have become increasingly complicated, **our code must manage more state than ever before**. This state can include server responses and cached data, as well as locally created data that has not yet been persisted to the server. UI state is also increasing in complexity, as we need to manage active routes, selected tabs, spinners, pagination controls, and so on
- Managing this ever-changing state is hard. If a model can update another model, then a view can update a model, which updates another model, and this, in turn, might cause another view to update. At some point, you no longer understand what happens in your app as you have **lost control over the when, why, and how of its state.**

Redux

- **Redux attempts to make state mutations predictable** by imposing certain restrictions on how and when updates can happen. These restrictions are reflected in the three principles of Redux.
    1. **Single source of truth**
    2. **State is read-only**
    3. **Changes are made with pure functions**

Single source of truth

- The state of your whole application is stored in an object tree within a single store.
- This makes it easy to create universal apps, as the state from your server can be serialized and hydrated into the client with no extra coding effort. A single state tree also makes it easier to debug or inspect an application; it also enables you to persist your app's state in development, for a faster development cycle.

State is read-only

- The only way to change the state is to emit an action, an object describing what happened.
- This ensures that neither the views nor the network callbacks will ever write directly to the state.
- Instead, they express an intent to transform the state.
- Because all changes are centralized and happen one by one in a strict order, there are no subtle race conditions to watch out for.
- As actions are just plain objects, they can be logged, serialized, stored, and later replayed for debugging or testing purposes.

Changes are made with pure functions

- To specify how the state tree is transformed by actions, you write pure reducers.
- Reducers are just pure functions that take the previous state and an action, and return the next state. Remember to return new state objects, instead of mutating the previous state.
- You can start with a single reducer, and as your app grows, split it off into smaller reducers that manage specific parts of the state tree.
- Because reducers are just functions, you can control the order in which they are called, pass additional data, or even make reusable reducers for common tasks such as pagination.

Actions

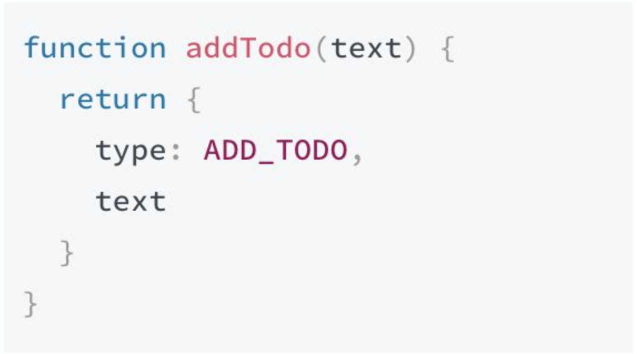
- **Actions** are payloads of information that send data from your application to your store. They are the *only* source of information for the store. You send them to the store using store.dispatch().
- Actions are plain JavaScript objects. Actions must have a type property that indicates the type of action being performed. Types should typically be defined as string constants. Once your app is large enough, you may want to move them into a separate module

```
const ADD_TODO = 'ADD_TODO'
```

```
{
  type: ADD_TODO,
  text: 'Build my first Redux app'
}
```

Action Creators

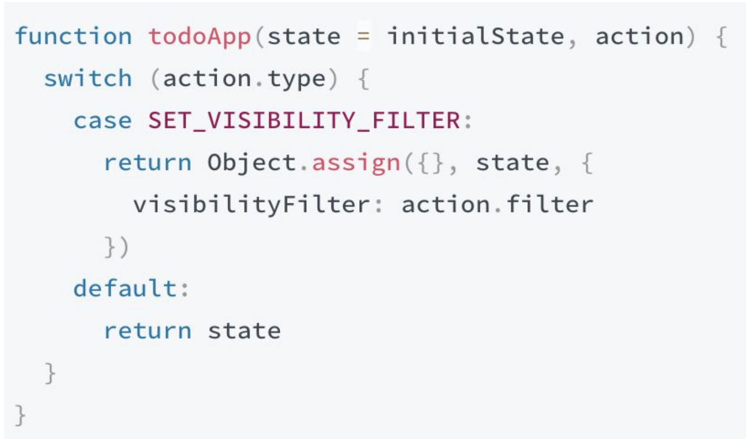
- **Action creators** are exactly that—functions that create actions. It's easy to conflate the terms “action” and “action creator”, so do your best to use the proper term.
- In Redux, action creators simply return an action.



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

Reducers

- **Reducers** specify how the application's state changes in response to actions sent to the store. Remember that actions only describe *what happened*, but don't describe how the application's state changes.



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

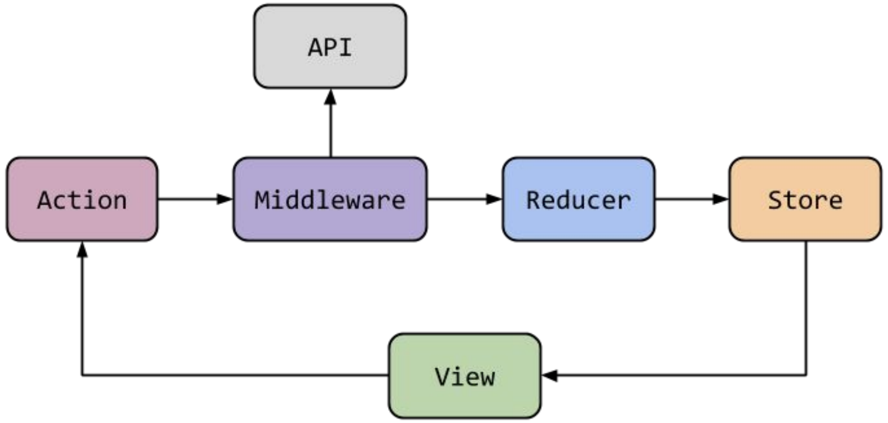
Store

Store has following responsibilities:

- Holds application state;
- Allows access to state via getState()
- Allows state to be updated via dispatch(action)
- Registers listeners via subscribe()
- Handles unregistering of listeners via the function returned by unsubscribe().

Data Flow

Redux Architecture


API
Action
Middleware
Reducer
Store
View