ReactJS-Redux

Agenda

- MVC Data State
- Problem Statement
- Solution: Flux/Redux
- Redux's Three Principles
- Pure Functions
- Actions / Action Creators
- Reducers
- React-Redux

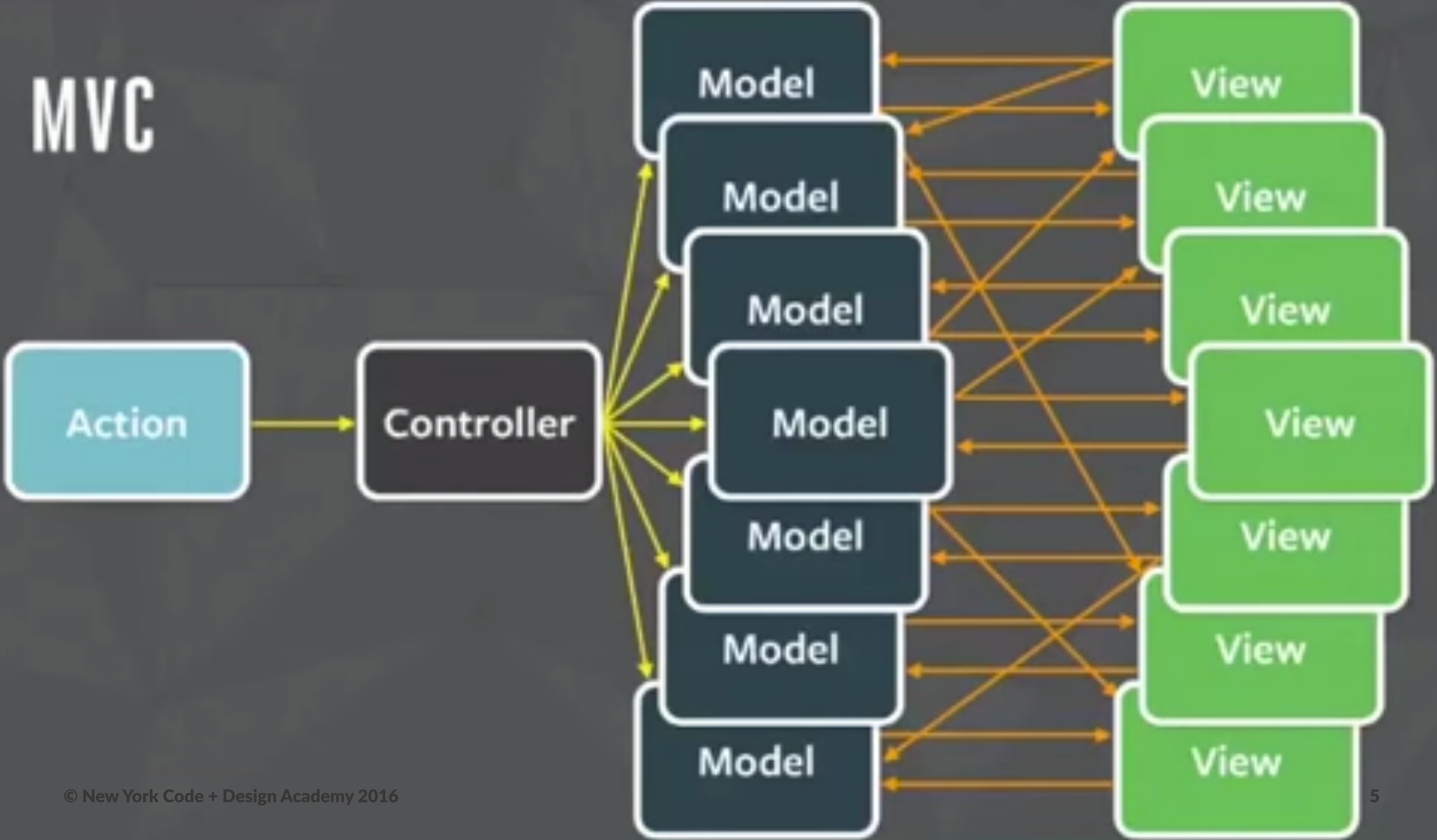
Hacker Way: Rethinking Web App Development at Facebook

https://youtu.be/nYkdrAPrdcw?t=10m24s

MVC Data State

In MVC applications, UI state is held in a number of Models, each of which has it's own internal state.

MVC


Action
Controller
Model
Model
Model
Model
Model
Model
Model
View
View
View
View
View
View
View

MVC Data State

MVC applications update by using **2-way data binding**

Two-way binding just means that:

1. When properties in the model get updated, so does the UI.
2. When UI elements get updated, the changes get propagated back to the model.

MVC Data State

In MVC architecture, both views and models both can update one another, which leads to some problems:

- Not very performant
- Hard to capture current state of the application

Problem Statement

Facebook users keep complaining because you can't keep your chat information all in sync

How do you debug something like this??

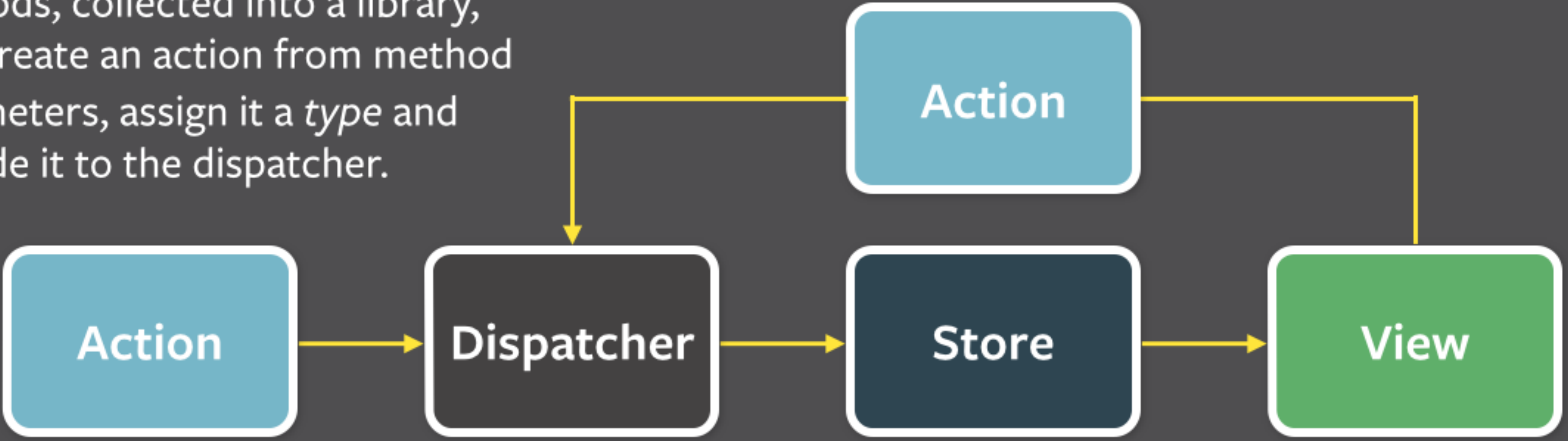
MVC architecture makes keeping track of the total state difficult, since with 2-way data binding Views can modify Models and vice versa.

Solution: Flux Architecture

Whiteboard session!

Facebook has proposed an alternative structure to the MVC data model called **Flux**, which is based on a **unidirectional data flow**:

Action creators are helper methods, collected into a library, that create an action from method parameters, assign it a *type* and provide it to the dispatcher.



Every action is sent to all stores via the *callbacks* the stores register with the dispatcher.

After stores update themselves in response to an action, they emit a *change* event.

Special views called *controller-views*, listen for *change* events, retrieve the new data from the stores and provide the new data to the entire tree of their child views.

Solution: Flux Architecture

Flux/Redux attempt to resolve the problem where bugs are difficult to trace when a model has been continuously updated as an app is being used.

The data store can only be mutated using actions, which percolate down into the data store, and trigger a re-render in the view.

Each change to the UI can be easily traced back to a series of actions, and the full state of the UI is available to view at any time.

Enter Redux

Flux is meant as more of an architecture than a specific implementation, Redux is a **minimal data store implementation** based on Facebook's Flux specification

- Redux is a predictable state container for JavaScript apps
- The data store is passed into React components, and that data will render them in a consistent way
- Any changes in the view have to be passed to Redux, which creates a new instance of the data store which re-render the React components

Redux Terminology

- `State` - The single state value that is managed by the store, it represents the entire state of a Redux application, which is often a deeply nested object (this is very similar to React state, but is stored externally)
- `Action` - A plain object that represents an intention to change the state. Actions are the **only way** to get data into the store. Actions must have a **type** property, but otherwise can have any number of data/ metadata fields
- `Reducer` - A **function** that accepts an accumulation and a value and returns a new accumulation. In Redux, the Reducer function produces new states for the data store given the previous state and an action object

Redux in a nutshell:

A *state** and **action** are passed into a **reducer** function, and a new state is returned*

Redux's Three Principles

- Single source of truth

  The state of your whole application is stored in an object tree within a single store.

- State is read-only

  The only way to change the state is to emit an action, an object describing what happened.

- Changes are made with pure functions

  To specify how the state tree is transformed by actions, you write pure reducers.

Pure Functions

Pure functions are not something specific to Redux, or even to JavaScript, but are a concept in traditional computer science.

A **pure function** is a function which:

- Given the same input, will always return the same output.
- Produces no side effects.
- Relies on no external state.

Math is made up of pure functions, for example `f(x) = 2x`

Pure Functions

Let's look at an example of an impure function:

```
var y = 2;
function badAdd(x) {
  return x + y;
}
```

This function is not pure because it references data that it hasn’t directly been given. As a result, it’s possible to call this function with the same input and get different output:

```
var y = 2;
badAdd(3) // 5
y = 3;
badAdd(3) // 6
```

Pure Functions

Now let's look at that function written in a pure way:

```
function add(x, y) {
  return x + y;
}
```

If you were to run `add(2, 4)` over and over, you'd have the same result (6), therefore it is a pure function.

Why Use Pure Functions?

The main advantage of a pure function is that **it doesn’t have any side effect**s. It doesn’t modify the state of the system outside of their scope.

Then, they just simplify and clarify the code: when you call a pure function, you just need to focus on the return value as you know you didn’t break anything elsewhere doing so.

Pure functions ensure that your functions return **consisent** results when you provide the same inputs.

Why Use Pure Functions?

Also, **it's very easy to unit test a pure function** since there is no context to consider. Just focus on inputs / outputs.

Finally, maximizing the use of pure functions **makes your code simpler, and more flexible**. This is how Redux implements the *predictable* part of being a predictable state container.

React Redux - Installation

- Grab the react-redux-boilerplate repo:

  github.com/yono38/react-redux-boilerplate

- Inside your new application run the command

```
$ npm install
$ npm start
```

Counter Example

Note: This example only uses redux, we will explain react-redux bindings later

```
// Create actions/index.js
// Namespace actions
export const INCREMENT = 'counter/INCREMENT';
export const DECREMENT = 'counter/DECREMENT';

// Create reducers/index.js
import { INCREMENT, DECREMENT } from '../actions'
const initialState = 0;
export default (state = initialState, action) => {
  switch (action.type) {
    case INCREMENT:
      return state + 1
    case DECREMENT:
      return state - 1
    default:
      return state
  }
}
```

Counter Example

```
// Create components/Counter.jsx
import React, { Component } from 'react'

const Counter = ({ value, onIncrement, onDecrement }) => (
  <div>
    <h1> Value: {value} </h1>
    <button onClick={onIncrement}> + </button>
    <button onClick={onDecrement}> - </button>
  </div>
);

export default Counter;
```

Counter Example

```
// Modify index.js

import React from 'react'
import ReactDOM from 'react-dom'
import { createStore } from 'redux'
import Counter from './components/Counter'
import counter from './reducers'
import { INCREMENT, DECREMENT } from '../actions'

const store = createStore(counter);

const render = () => ReactDOM.render(
  <Counter
    value={store.getState()}
    onIncrement={() => store.dispatch({ type: INCREMENT })}
    onDecrement={() => store.dispatch({ type: DECREMENT })}
  />,
  document.getElementById('root')
)

render();
store.subscribe(render)
```

Exercise - Clear Your Counter

- Modify your counter reducer to be able to have a `CLEAR` action that resets the state to `0`
- Modify your Counter component to add a button that will dispatch this `CLEAR` action

Actions

An action is the payload of data sent to a reducer to let it create a new state. It is an object that always contains a `type` property. The type string is often turned into a variable so it can be imported and used in other reducers and components.

```
const ADD_TODO = 'ADD_TODO'

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

There are no rules or constraints on the shape of an Action object other than the `type` field, but some developers follow the pattern in Flux Standard Action

Action Creators

Action creators are functions that return an action object. They are helper functions used when dispatching an action, for example:

```
const ADD_TODO = 'ADD_TODO';
function addTodo(text) {
  return {
    type: ADD_TODO,
    text
  }
}
```

This makes them reusable, and easy to test. You can use them by wrapping with a dispatch:

```
store.dispatch(addTodo('Build my first Redux app'))
```

Exercise - Increment By 5

- Modify your counter reducer to have the `INCREMENT` action be able to take an increment value. How are defaults handled?
- Create and export an Action Creator (in the actions file) that will take in an increment value and return an action that will increment by that amount
- Modify your Counter component to add a button that will increment the value by 5 on click using your Action Creator function

Reducers

Using actions, we can describe something that *has* happened in the application, but not what effect this has on the state. That is the job of the reducer.

When designing an application with Redux, start by thinking of what the simplest (and smallest!) representation of that state could look like.

Example

For example, let's say we wanted to write a Todo list application that has a set of Todos and a filter:

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

Exercise - Blog State Shape Design

Design a data state for a blog application which:

- Lists multiple posts, each of which has a `title`, `description` and `date`
- Allows for sorting by alphabetical, date ascending, and date descending

Exercise - Blog State Shape Design

One solution might look like this:

```
{
  sort: 'DATE_ASCENDING',
  posts: [
    {
      title: 'Intro to React',
      description: 'In this article we talk about the UI rendering library React.'
      date: '2016-10-01',
    },
    {
      title: 'React-Redux Fundamentals',
      description: 'Learn about the predictable data store, which makes your applications easier to write and debug.'
      date: '2016-10-05',
    },
  ]
}
```

Reducers

Reducers are functions that act on an action's `type`. They could be written as a set of `if-else` statements, but by convention they use a `switch` statement for readability.

The most basic reducer just returns the original state:

```
function reducer(state = initialState, action) {
  switch (action.type) {
    default:
      return state
  }
}
```

Reducers

A new requirement comes in that says we need to be able to add a new post to the data store. We modify the reducer to read in this action and modify the store:

```
function reducer(state = initialState, action) {
  switch (action.type) {
    case 'ADD_POST':
      state.posts.push(action.data);
      return state;
    default:
      return state;
  }
}
```

This is no good, it isn't a pure function! Can you explain why?

Reducers

By modifying the internals of the state rather than returning a new state, this reducer produces side effects and **is not predictable**. Running `reducer(state, action)` multiple times will grow the posts property in the state, even though it's being given the same inputs.

ES6 Review

- `spread operator (...)` - allows you to copy enumerable properties from one object to another in a more succinct way
- `Object.assign()` - creates a fresh copy of an object, bringing over all properties from the original object without referencing it

Reducers

We can use some ES6 magic in the `Object.assign` function and spread operator to rewrite this in as a pure operation:

```
function reducer(state = initialState, action) {
  switch (action.type) {
    case 'ADD_POST':
      // Create a new copy of the state which merges your new post into the state
      return Object.assign({}, state, {
        posts: [
          ...state.posts,
          action.data
        ]
      };
    default:
      return state;
  }
}
```

Reducers

`Object.assign` will create a fresh copy of the state object, with a new `posts` array that contains all the previous posts (using the spread operator) and our new post appended at the end.

Since it creates a fresh copy of the state rather than modifying the original, this reducer can be run with the same state and action inputs over and over again with the same predictable result.

Summary of Flux/Redux

"Let's zoom out a bit"

- Pure functions
- Actions
- Reducers

Exercise

Add to the previous reducer so that it can take an action that changes the sort value.

How do you write this in a way to ensure it is a pure function?

React-Redux

One thing to remember is that Redux has no relation to React. You can write Redux apps with React, Angular, Ember, jQuery, or vanilla JavaScript.

However, since React & Redux do work so well together, the React team has written some bindings to ease integration in the form of **react-redux**.

react-redux exports a `<Provider>` component that allows you to write child components that can interact more easily using the `connect()` decorator function.

React-Redux

One key to properly integrating React with Redux is separating **presentational** and **container** components.

- **Presentational Components**
  - Handle how things look (mark)
  - *Read Data* from Props
  - *Change Data* by invoking callbacks from props
- **Container Components**
  - How things work (data fetching, state updates)
  - *Read Data* by subscribing to Redux state
  - *Change Data* by dispatching Redux actions

React-Redux

A **Higher Order Function (HOF)** is a function that when called, returns another function as its result

react-redux supplies the *connect* HOF, which has the syntax:

```
connect(mapStateToProps, mapDispatchToProps)(PresentationalComponent)
```

- **mapStateToProps** - function that reads from *state* and props that are passed into the component (*ownProps*), and returns an object with props that should be passed into the component
- **mapDispatchToProps** - passes a *dispatch* function in which can be used in functions defined here. These functions are returned in an object and are passed to the presentational component, so that it can dispatch actions to the store

React-Redux

Go back to the counter application we made earlier in the class, we will modify it to use react-redux. Start by creating a container component for the Counter called `CounterApp`.

```
// components/CounterApp.jsx

import { connect } from 'react-redux'
import { INCREMENT, DECREMENT } from '../actions'
import Counter from './Counter'

const mapStateToProps = (state, ownProps) => {
  return {
    value: state.value
  }
}

const mapDispatchToProps = (dispatch, ownProps) => {
  return {
    onIncrement: () => {
      dispatch({ type: INCREMENT })
    },
    onDecrement: () => {
      dispatch({ type: DECREMENT })
    }
  }
}

const CounterApp = connect(
  mapStateToProps,
  mapDispatchToProps
)(Counter)
```

React-Redux

```
// modify index.js

...
// Add the new container component (you can also remove the Counter import)
import CounterApp from './components/CounterApp'
...

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

Exercise

Now try to modify the `CounterApp` component to include your changes for clearing the counter and incrementing by 5.

When implementing your increment by 5 action, remember to pull the action creator into your component.

Resources:

Official Redux Docs - Contains examples and full API docs

Getting Started with Redux - Free Video Course by the creator of Redux

Higher Order Components: A React Application Design Pattern