React with Flux and Redux

DEVELOPING APPLICATIONS USING REACTJS



Objectives

- · To understand what the Flux architecture is
- To be able to structure an application using Flux
- To understand what the Redux framework is and what it is used for
- To be able to structure an application using Redux

What is FLUX?

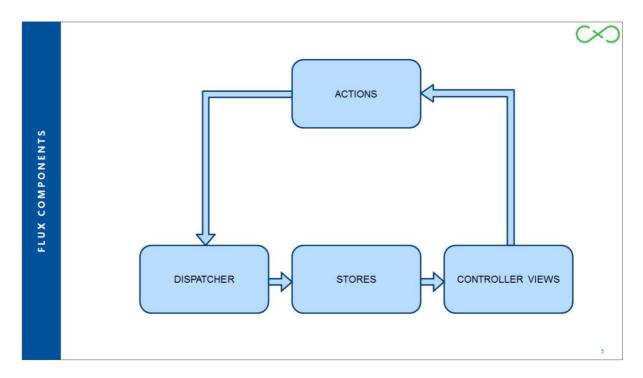
- FLUX is an architecture used and developed by Facebook
 - · NOT a framework or a library
- Compliments React and the concept of unidirectional data flow
- · Usually split into 4 distinct components
 - ACTIONS
 - DISPATCHER
 - STORES
 - CONTROLLER VIEWS
- Not to be likened to MVC this pattern is not what FLUX is about!
 - · Flux is about unidirectional data flow

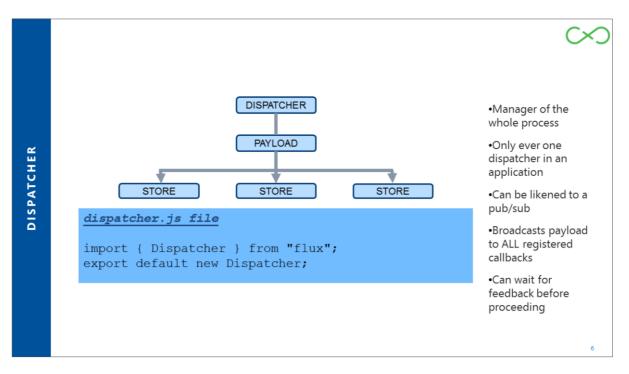
Flux Logo from: https://facebook.github.io/flux/



FLUX Components

- ACTIONS
 - · Helper methods that facilitate passing of data to Dispatcher
- DISPATCHER
 - · Receives actions and broadcasts payloads to registered callbacks
- STORES
 - Containers for application state and logic that have callbacks registered to Dispatcher
- CONTROLLER VIEWS
 - React components that get their state from Stores and pass it down via props to child components





The code show can function as a Dispatcher with nothing else. However, in more complex applications, it may be desirable to define some methods within the Dispatcher, especially if your application has view triggered actions and server/API triggered actions. In such cases, a dispatcher.js file may take the following form:

```
Import { Dispatcher } from "flux";
var AppDispatcher = new Dispatcher();

AppDispatcher.handleViewAction = (action) => {
    this.dispatch({
        source: 'VIEW_ACTION',
        action
    });
}
// Other methods could then be defined
```

export default AppDispatcher

When the AppDispatcher.handleViewAction is called, an action payload is broadcast to all registered callbacks and the action can be acted upon either the appropriate Stores and the results in Store update. Dependencies can also be defined and marshalled for callbacks on Stores. If a particular part of an application is dependent on another being updated, the dispatcher can ensure that this happens using waitFor().



STORE

- State for a particular application instance
 - Per app section, stores manage data retrieval methods and callbacks
- Data can be held in the Store, but better practice is to have a CONSTANTS file that holds the data
- Stores use Node's EventEmitter to be able to emit changes
- The Store must register with the Dispatcher to receive any actions that may be broadcast

```
stores/AppStore.js file
import { EventEmitter } from "events";
import dispatcher from "../dispatcher";
class AppStore extends EventEmitter {
    constructor() {
         super();
          this.data = [
                   {id: 1, name: "name1"},
{id: 2, name: "name2"}
         1
    getAll() {
         return this.data;
    handleActions(action) {
         switch(action.type) {
                   case "SOME_ACTION":
                             this.someOtherDefinedMethod(action.specificData);
                             break;
                   default:
                             break;
const appStore = new AppStore;
dispatcher.register(appStore.handleActions.bind(appStore));
export default appStore;
```

This code sets up a sample Store. Not all of the methods have been included but it could be used as a template for any Store. The handleActions is what will determine if the Store responds to an action that has been broadcast by the Dispatcher. If there is no CASE in the SWITCH then nothing will happen. If a CASE is found, then the appropriate method from the AppStore class will be called with any specific data that is needed from the action object supplied by the Dispatcher. The AppStore methods will be used by components to update their state to update the ControllerView.

Each method defined should emit a change so that the ControllerView can pick up on this and execute appropriately.



ACTIONS

- Collections of methods called within ControllerViews to send actions to the Dispatcher
 - · Actually payloads delivered via the dispatcher
- Action methods usually define a type and data that needs to be supplied (as an object) for the method as a parameter to a dispatcher.dispatch call

```
actions/AppActions.js file
import dispatcher from "../dispatcher";
export function filterBySearch(searchParameters) {
   dispatcher.dispatch({
      type: "FILTER_SEARCH",
      searchParameters
   });
}
```

There may be several methods defined within this file.



CONTROLLER VIEWS

- · Simply React components that listen to change events and retrieve application state from Stores
 - · State can then be passed down to dependent child components via props
 - · Lifecycle methods can be used to register/deregister listeners
 - · Other methods can be defined to handle changes
- · The base component will need to import the Store and the Actions

```
import React from "react";
import * as AppActions from "../actions/AppActions";
import AppStore from "../stores/AppStore";
```

The line:

import * as AppActions from "./actions/AppActions";

essentially means import all of the functions declared in the AppActions file and allow them to be called with the qualified name AppActions. So for a method filterBySearch(searchParams) in the AppActions file, the method could be called using AppActions.filterBySearch(searchParams).



CONTROLLER VIEWS - Initial State

- Components initial state can be set by calling the AppStore.getAll() method (if defined)
 - Used in the declaration of this.state in the constructor:

```
app/App.jsx file constructor

export default class App extends React.Component {
   constructor() {
      super();
      this.state = {
            data: AppStore.getAll(),
            otherState: "data"
      }
      this._onEventName =
            this._onEventName.bind(this);
   }
```

this._onEventName = this._onEventName.bind(this); binds the call of the _onEventName function to the App class rather than on the calling object. This will be used in the next section where reacting to changes are discussed.



CONTROLLER VIEWS – Lifecycle Methods

- Useful to register/deregister listeners on the Store
- Ensure that App responds to changes in the Store due to an Action
- Needs to have a handler function defined
- Unmounting protects against memory
 leaks

```
app/App.jsx file lifecycle methods

componentWillMount() {
    AppStore.on(eventName, this._onEventName);
}
componentWillUnmount() {
    AppStore.removeEventListener(eventName, this._onEventName);
}
_onEventName() {
    this.setState({
        data: AppStore.getFunction()
    });
}
```

12

As components exist in a virtual DOM, every time the component is created a new event listener is created and the old component stays in memory. This is bad!

To circumvent this, using the removeEventListener call when the component unmounts means that the component is fully removed from memory.



CONTROLLER VIEW - Invoking Actions

- Components on pages will invoke Actions by calling methods
 - If it is a user input then the state will be set for this before calling the appropriate Action with the input data supplied

```
app/App.jsx file action invocation

handleUserSearchInput(filterText) {
    this.setState({
        filterText
    });
    AppActions.filterBySearch(filterText);
}
```



Redux 🚱 Redux

- A predictable state container for JavaScript Apps
 - · Does not have to be used as part of a React application
 - Does fit nicely with React though
- · Evolves the ideas of Flux but avoids complexity by taking cues from Elm
 - · Is basically a way of implementing a Flux pattern with a few subtle changes

Similarities	Differences
Concentrate model update logic in certain layer of application (Stores in Flux, Reducers in Redux)	No concept of Dispatcher in Redux – relies on pure functions rather than event emitters
Both operate on the (state, action) => state	Redux assumes you never mutate data, always returning a new object

Redux logo from: http://redux.js.org/



Redux - Core Concepts

- · Application state described in a plain JavaScript object
 - · Object like a model but no setters
- State changes are from a dispatched action
 - · Action is a plain JavaScript object that describes what happened
- · State and Actions tied together with Reducers
 - Function that takes state and action as arguments and returns next state to the app
- redux.js.org claim that you can write 90% of the code needed in plain JavaScript with no use of Redux, its APIs or any magic!

1:



Redux – Three Principles

- · Single Source of Truth
 - · State of the 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
- · Changes are made with pure functions
 - To specify how the state tree is transformed by actions, you write pure reducers
 - Reducers take the previous state and an action and return the next state in a new state object rather than mutating previous state



When To Use REDUX?

- · Consider using Redux when:
 - Multiple React components need to access the same state but do not share any common ancestors
 - Passing state becomes a chain of props being passed down the component link



Redux - STORES

- · Orchestrates all the moving parts in Redux
- Place that holds the application's state Single Source of Truth!
- Uses redux package's createStore function and an appropriate reducer
- · Initial state can be passed to create store
 - · Not necessary most of the time but useful in server-side rendering

```
import { createStore } from 'redux';
import todoApp from '../reducers/reducers';
export const store = createStore(todoApp);
```



Redux - REDUCERS

- · Specify the changes that ACTION trigger to produce state
- · JavaScript function that takes in state and action and returns the updated state
- combineReducer allows for multiple reducers to be defined in multiple files and returned from a single file

```
reducers/reducers.js
import { combineReducers } from 'redux';
import { ADD_TODO } from '../actions/actions';
function todo(state, action) {
  switch(action.type) {
   case ADD TODO:
      return {id: action.id, text: action.text};
    default: return state;
function todos(state=[], action) {
  switch(action.type) {
   case ADD_TODO:
      return [...state, todo(undefined, action)];
   default: return state;
const todoApp = combineReducers ({ todos });
export default todoApp;
```

For the examples given in this course, reducers are held in a single file. In practice, individual reducers should be contained in their own files and exported. The reducers is file should import each reducer and supply it in the object passed into the combineReducers method. The const that this returns to is what is exported and used in any file that needs to access the reducers.



Redux - ACTIONS

- JavaScript objects that use type to inform about data to send to the store (dispatching)
- Example: Defining an ADD_TODO action used for adding the a new item to the list
 - addTodo is an action creator returns the action text and sets id for each new created item and the text for this todo as a payload

This could also have been written as a fat arrow function.





Redux

React and Redux

- · react-redux is a special package for binding React and Redux
- · Exposes connect method
 - Connects a React component to he Redux store
 - Has 2 or 3 arguments dependent on use case
 - · Need to know about:
 - mapStateToProps connects part of Redux state to props of React Component
 - · mapDispatchToProps connects Redux actions to React props (i.e. connected React component will be able to dispatch actions)



React and Redux

- React maintains single source of truth by only holding state in a root component
 - · Fits nicely with Redux
 - Create a special React-Redux component called Provider and have this hold state and wrap application in this
 - Store is passed as an attribute to the Provider allowing all child components access to it through props

```
main.js
import React from 'react';
import ReactDOM from 'react-dom';
import { Provider } from 'react-redux';
import { store } from './stores/TodoStore';
import App from './App';

ReactDOM.render(
<Provider store={store}><App /></Provider>,
document.querySelector("#app"));
```

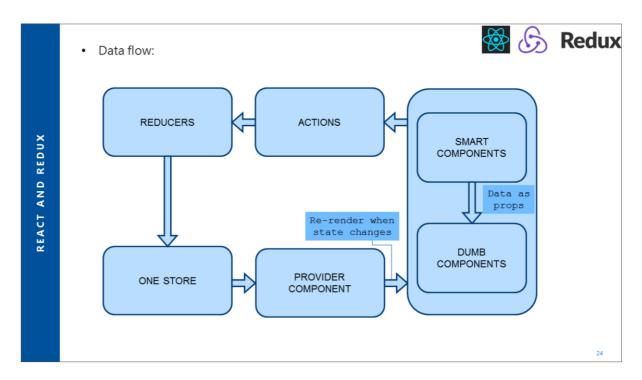
22

React logo from: https://facebook.github.io/react/



A container component is essentially a React component that uses store.subscribe() to access part of the state from the store and supply props to presentational components it renders.

In the code on the slide, the connect function is used to connect the App component to the the store. The select function is important here as it takes the state from the store and returns props (visibleTodos) to use in components.





Redux and Asynchronous Data

- By default, Redux's store and createStore() only support synchronous data flow.
- To allow asynchronous actions, createStore() can be enhanced using applyMiddleware()
 - · Asynchronous middleware like redux-thunk or redux-promise promise can be used
 - Both wrap store's dispatch() and allows dispatch of things other than actions
 - · Essentially, last middleware in chain dispatches an action as a plain object to supply data

```
stores/TodoStore.js
import { createStore, applyMiddleware } from 'redux';
Import thunk from 'redux-thunk';
import todoApp from '../reducers/reducers';
export const store = createStore(todoApp, applyMiddleware(thunk));
```



Exercise/Demo

• EG11 – Use React and Redux to populate data in an application

Objectives

- · To understand what the Flux architecture is
- To be able to structure an application using Flux
- · To understand what Redux is and what it is used for
- To be able to structure an React application using Redux