# 31 Redux

## Objectives

- Students will learn to use redux with react
- Students will learn to design redux reducer's for controlling application state
- Students will learn to design action creator functions for working with redux

# Redux: Three Principles

## Single Source of Truth

- A single state tree makes things easier to debug
- Enables you to persist app state in development
- Makes undo/redo state management timeline possible

## State is read-only

- All changes are centralized and happen one by one in a strict order
- No subtle race conditions to watch out for.
- Actions can be logged, serialized, stored, and later replayed for debugging or testing purposes.

## Changes are made with pure functions

- Return new state objects, instead of mutating the previous state.

# Actions, Reducers, Store

Redux uses three things together to manage state:

- **Actions** - payloads of info sent from the application to the store.
- **Reducers** - specify how app state changes in response to actions.
- **Store** - holds all app state.
  - Allows access to state via `getState()`
  - Allows state to update via `dispatch(action)`
  - Registers listeners via `subscribe(listener)`

# Actions

All `actions` are a plain JavaScript object. They must have a `.type` property.

```
{
  type: TOGGLE_TODO,
  index: 5
}
```

# Actions: Prefer Constants over Raw Strings

It's a good idea to define variables for String constants, then to always refer to that constant. This makes it easy to update the value of the String in the future, saves you from making hard-to-find typos typing raw strings elsewhere, and allows tools to pick up actions that are actually defined in your files.

**actions.js**

```
export const INCREMENT = 'INCREMENT';
export const DECREMENT = 'DECREMENT';
export const INCREMENT-BY = 'INCREMENT-BY';
```

# Action Creators

Action Creators are functions that create actions. They use the constants you defined earlier in the file. Action Creators can accept parameters to include value properties in the action.

Later you'll `dispatch` actions:

```
dispatch(incrementBy(12));
```

### actions.js

```
export function increment() {
  return {type: INCREMENT};
}

export function incrementBy(value) {
  return {type: INCREMENT, value: value};
}


export function decrement() {
  return {type: DECREMENT};
}
```

# Reducers

The reducer is a pure function that takes the previous state and an action, and returns the next state. A pure function is a function that always produces the **same output** given the **same input**. The original state should be unchanged.

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

```
import {INCREMENT, DECREMENT} from './actions';
const initialState = {data: 0};

function myAppReducer(state, action) {
  if (state === undefined) {
    return initialState;
  }

  // a switch block compares a value to the value associated with each case.
  switch(action.type) {
    case INCREMENT: return {data: state.data + 1};
    case DECREMENT: return {data: state.data - 1};
    default: return state;

  }
}
```

# Presentational and Container Components

## Presentational Components

- Are concerned with how things look.
- Have no dependencies on the rest of the app.
- Don’t specify how the data is loaded or mutated.
- Receive data and callbacks exclusively via props.
- Rarely have their own state (when they do, it’s UI state rather than data).
- Examples: Page, Sidebar, Story, UserInfo, List.

# Presentational and Container Components

## Container Components

- Are concerned with how things work.
- May contain both presentational and container components inside
- Usually don’t have any DOM markup of their own except for some wrapping divs
- Never have any styles.
- Provide the data and behavior to presentational or other container components.
- Examples: UserPage, FollowersSidebar, StoryContainer, FollowedUserList.

# Presentational and Container Components

| | **Presentational Components** | **Container Components** |
|---|---|---|
| Purpose | How things look (markup, styles) | How things work (data fetching, state updates) |
| Aware of Redux | No | Yes |
| To read data | Read data from props | Subscribe to Redux state |
| To change data | Invoke callbacks from props | Dispatch Redux actions |
| Are written | By hand | Usually generated by React Redux |

# Providing the Store

The `store` is where all the data for the whole app lives. Create the store with `createStore` provided by Redux.

Wrap the whole `<App/>` inside a `<Provider>` that's had `store` passed into it. This will make the store available everywhere in your application.

```
import React from 'react'
import { render } from 'react-dom'
import { Provider } from 'react-redux'
import { createStore } from 'redux'
import todoApp from './reducers'
import App from './components/App'

const store = createStore(todoApp)

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

# Overall File Structure

```
src
├── main.js
├── action
|   └── category-actions.js
├── component
|   ├── app.js
|   ├── category
|   |   └── category-form
|   |       ├── _category-form.scss
|   |       └── category-form.js
|   └── dashboard
|       ├── _dashboard.scss
|       └── dashboard.js
├── reducer
|   └── category.js
└── style
    └── main.scss
```

# Dependencies

Version numbers here aren't important.

```
"dependencies": {
  "babel-core": "^6.26.3",
  "babel-loader": "^7.1.4",
  "babel-preset-env": "^1.6.1",
  "babel-preset-react": "^6.24.1",
  "css-loader": "^0.28.11",
  "html-webpack-plugin": "^3.2.0",
  "node-sass": "^4.9.0",
  "react": "^16.3.2",
  "react-dom": "^16.3.2",
  "react-redux": "^5.0.7",
  "react-router-dom": "^4.2.2",
  "redux": "^4.0.0",
  "sass-loader": "^7.0.1",
  "style-loader": "^0.21.0",
  "webpack": "^4.6.0",
  "webpack-cli": "^2.1.2",
  "webpack-dev-server": "^3.1.4"
}
```

# Passing State to dispatch()

In order to pass state from a component to a dispatch that's been mapped to props you must change the `onClick` reference to a arrow function that executes itself when someone clicks and passes the state as a parameter.

```
<button onClick={() => this.props.decrementBy(this.state.decBy)}>
  decrement by
</button>
```

```
const mapDispatchToProps = (dispatch) => {
  return {
    increment: () => dispatch(increment()),
    decrement: () => dispatch(decrement()),
    incrementBy: (val) => dispatch(incrementBy(val)),
    decrementBy: (val) => dispatch(decrementBy(val)),
  }
};
```

# Use Object.assign

- MDN Docs Object.assign

Use `Object.assign` to preserve all the other properties on the state, and to apply all the properties of state to a new object so it doesn't overwrite the old state.

```
let newState = {};
switch(action.type) {
  case INCREMENT:
    return Object.assign(newState, state, {data: state.data + 1});
}
```

# .connect() / mapState / mapDispatch

```
class Dashboard extends React.Component {
  // component stuff
}

// pick properties of state to be avail in .props
const mapStateToProps = state => ({
  data: state.data,
  appName: state.appName,
});

// create functions that will dispatch actions
const mapDispatchToProps = (dispatch) => {
  return {
    increment: () => dispatch(increment()),
    decrement: () => dispatch(decrement()),
    incrementBy: (val) => dispatch(incrementBy(val)),
    decrementBy: (val) => dispatch(decrementBy(val)),
  }
};

export default connect(mapStateToProps, mapDispatchToProps)(Dashboard);
```