Redux

- reactjs/components/App.js has the CONNECTOR that maps STATE and ACTIONS to this.PROPS >
    - access state and action through this.props >
    - button_click >
    - trigger's this.props.ACTION_NAME function > (type: INCREMENT_LIKE) (in actionCreators.js) >
    - all REDUCERS are DISPATCHED >
- switch/case in INDIVIDUAL REDUCERS >
    - return NEW state
- reactjs/components/App.js has the CONNECTOR that maps STATE and ACTIONS to this.PROPS
    - you can then use those actions to DISPATCH **ALL** REDUCERS… and in the INDIVIDUAL reducer files, you need a SWITCH statement, that checks the ACTION TYPE(in actionCreators.js)… then execute code
- The INDIVIDUAL files in the reducers folder are the functions that update the STATE.

Fundamental Concepts

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(). The action is the JS Object.

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.

Actions describe the fact that something happened, but don't specify how the application's state changes in response. This is the job of **reducers**.

In Redux, all the application state is stored as a **single object**. It's a good idea to think of its shape before writing any code. What's the minimal representation of your app's state as an object?

The **reducer** is a **pure function** that takes the previous state and an action, and returns the next state.
`(previousState, action) => newState`

It's very important that the reducer stays pure. ***Things you should never do inside a reducer***:

- Mutate its arguments;
- Perform side effects like API calls and routing transitions;
- Call non-pure functions, e.g. Date.now() or Math.random().

Given the same arguments, it should calculate the next state and return it. No surprises. No side effects. No API calls. No mutations. Just a calculation.

INITIAL STATE in REDUCERS

- The `combineReducers()` helper function(*reactjs/reducers/index.js file*) turns an object whose values are different reducing functions into a *single reducing function* you can **pass to** **`createStore()`**. (*which lives in your reactjs/store.js file*)
- all reducers are combined into a single reducer with `combineReducers()`…
- they are then passed into `createStore(reducers)`.
- the `const` values, **`rootReducer`** and **`defaultState`** **MUST** have matching keys…. BUT…in the `reactjs/components/App.js` file, you can use `mapStateToProps()` to create custom *keys* for your state data.. which in turn is passed to Components as props. Example Below:

`reactjs/store.js`

```
// import the root reducer
import rootReducer from './reducers/index'
import comments from './data/comments'
import posts from './data/posts'

// create an object for the default data
const defaultState = {
    posts,
    comments,
    testAction: {"html":"easy","php":"lame","python":"cool","java
script":"does everything"}
}

const store = createStore(rootReducer, defaultState)

export const history = syncHistoryWithStore(browserHistory, stor
e)

export default store
```

reactjs/reducers/index.js

```
import { combineReducers } from 'redux'
import { routerReducer } from 'react-router-redux'

import posts from './posts'
import comments from './comments'
import testAction from './testaction'

const rootReducer = combineReducers({
    posts,
    comments,
    testAction,
    routing: routerReducer

})

export default rootReducer
```

reactjs/components/App.js

```
import { bindActionCreators } from 'redux'
import { connect } from 'react-redux'
import * as actionCreators from '../actions/actionCreators'

import Main from './Main';

// These values, ie: `posts` and `comments` will
// become available to `this.props` in components.
function mapStateToProps(state){
    return {
        posts:state.posts,
        comments:state.comments,
        defaultStateExample:state.testAction
    }
}
// The functions that live in `actionCreators.js` file will
// become available to `this.props` in components.
// ie: increment, addComment, testAction, etc.
function mapDispatchToProps(dispatch) {
    return bindActionCreators(actionCreators, dispatch)
}

const App = connect(mapStateToProps, mapDispatchToProps)(Main)

export default App
```

…..

…..

REDUCER STATE IS EMPTY INITIALLY…

```
export default function posts(state = [], action) {
    switch (action.type) {
        case 'INCREMENT_LIKES':
            const i = action.index
            return [
                ...state.slice(0, i), // everything BEFORE what we are updating,
                {...state[i], likes: state[i].likes + 1}, // our functional stuff
                ...state.slice(i + 1)
            ]
            break;
        default:
            return state;
    }
}
```

- the state is held in a **Redux Store**
- the **store** contains all of the data for project in a single object.
- example setup of a basic `store.js` file

```
import { createStore, compose } from 'redux'
import { syncHistoryWithStore } from 'react-router-redux'
import { browserHistory } from 'react-router'


// import the root reducer
import rootReducer from './reducers/index'

import comments from './data/comments'
import posts from './data/posts'

// create an object for the default data
const defaultState = {
    posts,
    comments
}

const store = createStore(rootReducer, defaultState)

const history = syncHistoryWithStore(browserHistory)

export default store
```

- **Actions** - something that *happens* in the application, eg: someone adds a comment, removes an image, etc. You **Dispatch** an **Action**
- Whenever an **Action** happens it **Dispatches**
- the dispatcher contains two things.
    - The **type of action** that happened, ig: `DELETE_COMMENT`
    - a **Payload** of information, ie: *Which comment was deleted?*

- **Redux Dev Tools** will show you all information about an **Action** whenever it is **Dispatched** by the user.
- **Redux Dev Tools** will let you toggle actions, which basically let's traverse time and actions in the app.
- when you hit Reducers, how do you actually update the data and let React update the data?
- **ACTION CREATORS**
- created in `reactjs/client/actions/actionCreators.js`
    - Above there is only one `actionCreator` but some people will create an `actionCreator` for *each* file.

- These actions become available to `this.props` in your components (where applicable) like shown in the example below:

```
// in full example:
<img src="./like-button.png" className="like-button" onClick={this.props.increment.bind(null, i)} />
```

```
{this.props.increment.bind(null, i)}
```

… the action **increment** lives in your **actionCreators** file `reactjs/actions/actionCreatores.js` … it is connected to your `props` through the MAIN `App.js` file in the components folder with the command:

```
const App = connect(mapStateToProps, mapDispatchToProps)(Main)
```

… full file shown below…

- **SO**, when that action above: `this.props.increment` is fired off(dispatched), **ALL OF THE REDUCERS ARE TRIGGERED**… that means all of the files in the **reducers** folder. BUT they need to be added all to a single reducer, `index.js`
- inside the individual reducer files is where you handle conditional logic checking.
- ACTIONS EXAMPLE:

```
// defines "things" that happen in the application.
// - the `return` statements are the "actions"
// - the methods act as the "action creators"
// - these methods get "dispatched".
// - "reducers" are what goes back to update the "Redux Store", our single source of data.


// increment
// only send as little information as needed to complete the action,
// in this example all we need is the index to find which one to update.
export function increment(index) {
    return {
        type: 'INCREMENT_LIKES',
        index
    }
}

// add comment
export function addComment(postId, author, comment){
    return {
        type: 'ADD_COMMENT',
        postId,
        author,
        comment
    }
}

// remove comment
export function removeComment(postId, index) {
    type: 'REMOVE_COMMENT',
    postId
    index
}
```

- **REDUCERS** are what actually go back and update the `state` in React.
- The **reducer** does the actualt editing of the **state**
- create a **reducer** for each piece of state, eg: `comments` , `posts` , `users` ....
- ..eventually these get combined into a *single* reducer called the
- EVERY TIME an **action** is **dispatched**, **ALL** reducers will be run.... example:

```
$r.store.dispatch({type:'INCREMENT_LIKES', index:0})
// this will dispatch ALL reducers... look in the `rootReducer` const in reactjs/reducers/index.js
```

- `reducers/index.js` holds all of the independent reducers, and you need to be sure to place all independent reducers inside the `const` **`rootReducer`**
- **YOU MUST WRITE LOGIC INSIDE OF YOUR INDEPENDENT REDUCERS** to find if one the actions you are interested in is triggered, OTHERWISE JUST `return state`.
  ROOT REDUCER
- a **reducer** takes in 2 things:
  - 1) the action (info about what happened
  - 2) a copy of the current store(state)

- be aware that the browser state, ie: location of the URLs will also live in the state/store. Therefore, when we create our `rootReducer`, we need to pass in `routerReducer`: `routing:routerReducer` in **reducers/index.js**
  - **Redux's** `connect` will allow you to inject state data into whatever level you need it at, as opposed to ReactJS typically passing state down to components as props. You WILL STILL NEED TO PASS PROPS TO SUBCOMPONENTS, but the initial injection is there to pass it down to the main App component, instead of manually traversing multiple components to get the `state` down to `<Main />`.. this is all done through the `connect` component included from `react-redux`
- Your main Index.js file should include **reactjs/components/App.js** as an import.. and pass it to the main `Route` with `path="/"` nested in `Router` the file should look similar to the **Index.js** file below.
- **Main.js** will create a COPY of its DIRECT CHILD sub-components and pass `this.props` (ie: 's props') to them:
  piece of code in Main.js

```
// Pass any PARENT props down to CHILDREN

{ React.cloneElement(
    this.props.children, // this is the element(s) we want to clone.
    this.props // this is what we want to pass to the newly cloned elements.
) }
```

Index.js

```
import React from 'react'
import { render } from 'react-dom'
import css from './styles/style.styl'

import App from './components/App'
import Single  from './components/Single'
import PhotoGrid from './components/PhotoGrid'

import { Router, Route, IndexRoute, browserHistory } from 'react-router'
import { Provider } from 'react-redux'
import store, { history } from './store';

const router = (
    <Provider store={ store }>
        <Router history={history}>
            <Route path="/" component={App}> {/* match the root of the site. This is basically a "copy" of <Main /> Component */}
                <IndexRoute component={PhotoGrid}></IndexRoute> {/* This will be a CHILD of the <Main /> Component. Notice it's nested. */}
                <Route path="/view/:postId" component={Single}></Route> {/* This will be a CHILD of the <Main> Component. Notice it's nested. */}
            </Route>
        </Router>
    </Provider>
)


render(router, document.getElementById('root'))
```

App.js

```
import { bindActionCreators } from 'redux'
import { connect } from 'react-redux'
import * as actionCreators from '../actions/actionCreators'

import Main from './Main';

function mapStateToProps(state){
    return {
        posts:state.posts,
        comments:state.comments
    }
}

function mapDispatchToProps(dispatch) {
    return bindActionCreators(actionCreators, dispatch)
}

const App = connect(mapStateToProps, mapDispatchToProps)(Main)

export default App

```

the file above calls on `<Main />` which has some important details in it, such as the `React.cloneElement` which will CLONE all child elements and apply the entire state to the object..ie: the props from `<Main />`… and `<Main />` has the state props which is passed through by `connect` … Let's take a look at **Main.js**…

```
import React, { Component } from 'react'
import { Link } from 'react-router'


class Main extends Component {
    render(){
        return(
            <div>
                <h1>
                    <Link to="/">Reduxstagram</Link>
                </h1>
                {/* Pass any PARENT props down to CHILDREN */}
                {React.cloneElement(
                    this.props.children, // this is the elemen
t(s) we want to clone.
                    this.props // this is what we want to pass to
the newly cloned elements.
                )}
            </div>
        )
    }
}

export default Main
```

- A way to **pass a props object to a component**, is to use the *"ES6/JSX spread attribute"*. It will automatically map attributes and values for your component. Example: both functions will do the same thing…b

```
function App1() {
  return <Greeting firstName="Ben" lastName="Hector" />;
}

function App2() {
  const props = {firstName: 'Ben', lastName: 'Hector'};
  return <Greeting {...props} />;
}
```

REACTJS USES PURE FUNCTIONS

- https://tylermcginnis.com/building-user-interfaces-with-pure-functions-and-function-composition-in-react-js/
- example:

```
var ProfilePic = function (props) {
  return <img src={'https://photo.fb.com/' + props.username'} />
}
var ProfileLink = function (props) {
  return (
    <a href={'https://www.fb.com/' + props.username}>
      {props.username}
    </a>
  )
}
var Avatar = function (props) {
  return (
    <div>
      <ProfilePic username={props.username} />
      <ProfileLink username={props.username} />
    </div>
  )
}
```

AN IMPORTANT CONCEPT OF REDUX IS REDUCER COMPOSITION

- http://redux.js.org/docs/basics/Reducers.html#splitting-reducers
- a reducer should take a slice of the state and pass it off to other reducers.
- example:

```
function todoApp(state = initialState, action) {
  switch (action.type) {
    case SET_VISIBILITY_FILTER:
      return Object.assign({}, state, {
        visibilityFilter: action.filter
      })
    case ADD_TODO:
    case TOGGLE_TODO:
      return Object.assign({}, state, {
        todos: todos(state.todos, action) // only send the "todos" part of the state to the `todos()` method.
      })
    default:
      return state
  }
}
```

and we can then work with that slice of data in `todos()`…

```
function todos(state = [], action) {
  switch (action.type) {
    case ADD_TODO:
      return [
        ...state, // only contains state.todos;
        {
          text: action.text,
          completed: false
        }
      ]
    case TOGGLE_TODO:
      return state.map((todo, index) => {
        if (index === action.index) {
          return Object.assign({}, todo, {
            completed: !todo.completed
          })
        }
        return todo
      })
    default:
      return state
  }
}
```

….Read more at http://redux.js.org/docs/basics/Reducers.html.

HOT RELOADING REDUX REDUCERS WITH WEBPACK

- hot reload reducers
- open store.js
- accept hotreload - re-require the reducer - swaps it out inside the store

```
if(module.hot) {
    module.hot.accept('./reducers/', () => {
        const nextRootReducer = require('./reducers/index').default
        store.replaceReducer(nextRootReducer)
    })
}
```

REDUX DEV TOOLS

- need to update `store.js`

```
const enhancers = compose(
    window.devToolsExtension ? window.devToolsExtension() : f =>
f
)

const store = createStore(rootReducer, defaultState, enhancers)
```

API STUFF

- consider redux thunk or saga
- normalizer for deeply nested JSON

MAKING API REQUESTS

Usually, for any API request you'll want to dispatch at least three different kinds of actions:

An action informing the reducers that the request began.

The reducers may handle this action by toggling an isFetching flag in the state. This way the UI knows it's time to show a spinner.

An action informing the reducers that the request finished successfully.

The reducers may handle this action by merging the new data into the state they manage and resetting isFetching. The UI would hide the spinner, and display the fetched data.

An action informing the reducers that the request failed.

The reducers may handle this action by resetting isFetching. Additionally, some reducers may want to store the error message so the UI can display it.

You may use a dedicated `status` field in your actions:

```
{ type: 'FETCH_POSTS' }
{ type: 'FETCH_POSTS', status: 'error', error: 'Oops' }
{ type: 'FETCH_POSTS', status: 'success', response: { ... } }
```

Or you can define separate types for them:

```
{ type: 'FETCH_POSTS_REQUEST' }
{ type: 'FETCH_POSTS_FAILURE', error: 'Oops' }
{ type: 'FETCH_POSTS_SUCCESS', response: { ... } }
```

REDUX THUNK

- https://github.com/gaearon/redux-thunk
- Redux Thunk middleware allows you to write action creators that return a function instead of an action. The thunk can be used to delay the dispatch of an action, or to dispatch only if a certain condition is met.