Week 5 Workshop

React Course

Agenda

Activity	Time
Get Prepared: Log in to Nucamp Learning Portal • Slack • Screenshare	10 minutes
Check-In	10 minutes
Weekly Recap: Redux Actions & Thunk, Json-server, Fetch, Animations	50 minutes
Task 1:	50 minutes
BREAK	15 minutes
Tasks 2 & 3	90 minutes
Check-Out	15 minutes

Check-In

- How was this week? Any particular challenges or accomplishments?
- Did you understand the Exercises and were you able to complete them?
- You must complete all Exercises before beginning the Workshop Assignment. We will review the Exercises together next, along with this week's concepts.

Week 5 Recap – Overview

Some of the New Concepts You Learned This Week

• Combining Reducers • Redux Actions • Redux Thunk • Middleware	Json-server REST & HTTP Methods Fetch & Promises Animations

Next slides will review these concepts,
along with some of the Exercises you did this week.

Redux Action Types – Why?

- We have been keeping action types in a separate file as constants, e.g.:

```
export const ADD_COMMENT = 'ADD_COMMENT';
export const DISHES_LOADING = 'DISHES_LOADING';
export const DISHES_FAILED = 'DISHES_FAILED';
export const ADD_DISHES = 'ADD_DISHES';
export const ADD_COMMENTS = 'ADD_COMMENTS';
export const COMMENTS_FAILED = 'COMMENTS_FAILED';
export const PROMOS_LOADING = 'PROMOS_LOADING';
```

- Discuss as a class: *Why* have we been doing this? It could seem like we're just repeating ourselves.

Redux Action Types – Why?

- Some answers, straight from the Redux documentation:
  - It helps keep the naming consistent because all action types are gathered in a single place.
  - Sometimes you want to see all existing actions before working on a new feature. It may be that the action you need was already added by somebody on the team, but you didn't know.
  - The list of action types that were added, removed, and changed in a Pull Request helps everyone on the team keep track of scope and implementation of new features.
  - If you make a typo when importing an action constant, you will get *undefined*. Redux will immediately throw an error when dispatching such an action, and you'll find the mistake sooner.
- Note: Like Redux itself, this may be more useful in larger projects. In very small projects with few actions, this may not be necessary, and Redux will not force you to do it. However, for the reasons noted above, it is a good idea and a very common convention.

Redux Action Types - Importing

- You have seen this syntax for importing action types for use in another file:
  - `import * as ActionTypes from './ActionTypes';`
- What is `import * as ____` doing? It makes all your exported constants available to you in that file as ____.<name of const>. E.g.
  - `ActionTypes.DISHES_LOADING`
- If you had written instead:
  - `import * as Foo from './ActionTypes';`
- Then you would have to use: `Foo.DISHES_LOADING`
- (but don't use Foo.)

Redux Action Creators

- *Action:* a JavaScript object with an action type and often a payload of data sent to the Redux Store.
- *Action creator:* a function that creates and returns an action, e.g.:

```
export const addCampsites = campsites => ({
    type: ActionTypes.ADD_CAMPSITES,
    payload: campsites
});
```

  - Remember, an arrow function that has only one expression in its function body, without curly braces surrounding the function body, will return that expression automatically even though there is no return statement.
- When you want to dispatch this type of action, you call the action creator function inside dispatch, e.g. `dispatch(addCampsites(campsites))`
- Using action creators is a convention as well. Redux does not force you to do this. You could instead create your action object inside your dispatch, such as: `dispatch({ type: ActionTypes.ADD_CAMPSITES, payload: campsites})`
- Using an action creator ensures that if you need to change how your action is created, for example if you need to introduce some Redux Thunk middleware, then you only need to change it in one place instead of chasing it down all over your code.

Redux Reducers

- Actions describe *what happened* (or what you want to happen), such as ADD_COMMENT and a payload that holds the comment you want to add.
- Reducers do the work of actually making things happen.
- Using `dispatch()` sends the action to the store, where it is handled by the root reducer defined in `createStore()`
- Reducers are *pure functions* with this signature:
  - `(previousState, action) => newState`

  They take the dispatched pre-existing state and an action, then create and return the new state (without mutating the old state) to the store.

And then what?

- You may have heard React being called "declarative".
- This means that in React, you do not tell the DOM exactly what to do - delete this node, add that node, etc.
- Instead, you set the virtual DOM the way you want it – you declare what you want, and the real DOM *reacts* without you touching it directly, thanks to React magic (a process called *reconciliation*).
- With Redux added to React, this declaration happens via the action and reducer pattern ultimately modifying the application state inside the Redux Store.
- The Store then saves this new state as the new virtual DOM state, and via the React-Redux bindings, your React app will update as necessary.

Redux Reducers

(currentState, action) => newState

```
export const Promotions = (state = { isLoading: true,
                                     errMess: null,
                                     promotions: [] }, action) => {
    switch (action.type) {
        case ActionTypes.ADD_PROMOTIONS:
            return {...state, isLoading: false, errMess: null, promotions: action.payload};

        case ActionTypes.PROMOTIONS_LOADING:
            return {...state, isLoading: true, errMess: null, promotions: []}

        case ActionTypes.PROMOTIONS_FAILED:
            return {...state, isLoading: false, errMess: action.payload};

        default:
            return state;
    }
};
```

- The reducer function takes two arguments: the current state and the action that has been dispatched.
- Most often, it will use a switch statement to cycle through potential action types, create a new state object with the change requested by the action, then return it.
- Discuss in class: What does the `state =` syntax do in the parameter list of this reducer?

Redux Middleware

- Redux Middleware lets you insert code that's executed after an action is dispatched and before it reaches the reducer, via using `applyMiddleware()` in your call to `createStore()`.
- `applyMiddleware(...middleware)` - you can chain middleware functions as arguments to applyMiddleware here so that actions can go through multiple middleware in turn (such as *thunk* and *logger*)
- This is the preferred method for a third-party extension point, where you can intercept and view/modify/stop the action before it reaches the reducer, or dispatch other actions

Redux Thunk

- *Thunk* is a general programming concept where you place a function inside another function to be executed later.
- Redux Thunk is a type of middleware that allows you to write action creators that return a function instead of an object.
- A common example: If you want to send a call to a server then wait for it to respond before dispatching another action, Thunk helps you do that.

Client-Server Communication

- HTTP – hypertext transmission protocol
  - Get, Put, Post, Delete HTTP methods (there are others – Head, Options..)
  - HTTP response status codes e.g. 200, 404, 303 ...
- URL – uniform resource locator, unique address of files on servers that are on the internet
- JSON – JavaScript Object Notation
  - Strings containing information in similar format as a JavaScript object literal with a few differences, mainly that the property name must be in double quotes: { "id": 1, "wings": 2 }
  - one way of encoding data transmitted between server & client that is extremely easy to plug into JavaScript and to extract data from as objects
  - There are other ways to encode data for transmission, e.g. XML

The Cloud

- Modern web apps often hosted on 'cloud' backend
- The 'cloud' is a cluster of server computers, if one fails your server is moved to another, high-traffic and crucial websites are safer using cloud services
- Developers/companies used to (and some still do) host their own websites on their own servers they would manage, or they would use web shared hosting providers.
- Now there are many cloud services that manage servers for you – Amazon Web Services, Azure, Heroku, Digital Ocean, etc.

REST: Representational State Transfer

- Architectural style for developing Web services (resources available on the Web)
- REST is *not* a library, *not* a framework, *not* a language, just a standardized approach to handling network data transfer
- Two common approaches: REST and SOAP (REST is newer)
  - SOAP: Simple Object Access Protocol, uses XML for data encoding instead of JSON, other differences – is an actual protocol specification unlike REST
  - REST: Can use either XML or JSON (or even plain text or HTML), most often JSON
  - REST becoming increasingly popular

REST (cont)

- 6 principles of RESTful web service/API
  1. Client/server separation
  2. Statelessness
  3. Uniform Interface
  4. Cacheable
  5. Layered System
  6. Code on Demand (optional)

REST (cont)

- REST uses HTTP methods Get, Post, Put, Delete, roughly analogous to Read Create Update Delete in CRUD terms
- CRUD operations: Create Read Update Delete
  - CRUD operations are a general computer/database programming paradigm of the four basic operations you can do with data.
- Resource and Representation -
  - Any piece of data available from the server is a "resource"
  - When a resource is requested, the server will encode it into the appropriate format for transfer, such as JSON, and this is the "representation" that is sent over the network.
  - The resource is not sent, the representation is what's transferred.

Promises

- A part of ES6 JavaScript
- Allows us to run code asynchronously – not all on the same timeline
- We can send out requests but not have to wait for the responses to continue executing our app, deal with responses later when they return
- Creating a promise: `new Promise( function(resolve, reject) { ... } );`
- Example:
  ```
  const myFirstPromise = new Promise((resolve, reject) => { … });
  ```
- The function being passed into the Promise constructor is called the *executor function*.
  It takes two other functions (resolve, reject) as arguments.
  - `resolve:` called when promise task succeeds, return value holds task results
  - `reject:` called when task fails, returns reason for failure – typically an error object.

Promises

- The .then() method is used on a promise to handle what happens when a promise resolves or rejects
- The first argument of the .then() method is a callback function that handles the resolve case, the second (optional) argument handles the reject case
- The return value of a .then() method is returned as another promise, so you can chain them together to create a promise chain
- The .catch() method can be chained to a .then() method to catch any rejected promises that are not handled
- The throw statement can be used at any point to throw to the next available catch block

Fetch

- Modern API replacement for XMLHttpRequest
- Alternatives: Axios, Superagent
- Supports promise-based approach – `fetch()` returns a promise
- Client makes fetch requests, server makes fetch response.
- Fetch requests must provide a URL, may also provide a request method (default is GET), headers, body, more
- Fetch responses will provide a status code (404, 200, etc) and a response.ok property (true for 200-299 status code), may also include headers, body, more

Workshop Assignment

- It's time to start the workshop assignment!
- Sit near your workshop partner.
  - Your instructor may assign partners, or have you choose.
- Work closely with each other.
  - 10-minute rule does *not* apply to talking to your partner, you should consult each other throughout.
- Follow the workshop instructions very closely
  - both the video and written instructions.
- Talk to your instructor if any of the instructions are unclear to you.

Check-Out

- Turn in your assignment files:
  - *AboutComponent.js, ActionCreators.js, ActionTypes.js, ContactComponent.js, HomeComponent.js, partners.js, MainComponent.js*
- Wrap up – Retrospective
  - What went well
  - What to improve
  - Action items suggestions
- Work on your Portfolio Project more if there is time left
- WE MADE IT! Congratulations everyone! This was not an easy course and you have learned a lot in just five weeks.
- Next course: REACT NATIVE **mobile app development!**
- Enjoy your 1-week break everyone :)