React Notes

Useful Libraries / Resources

CSS library: semantic-ui.com
faker js - fake data library
Geolocation API (mozilla documentation)
lodash (_.)
axios (api calls)
react-router-dom (routing/navigation for dom based apps - use native for mobile)
console.developors.google.com (gapi documentation)
http://github.com/zalmoxisus/redux-devtools-extension - redux devtool debugger

- use `?debug_session=<some_string>` to save state of app / create checkpoints

redux-form (wizard form)
json placeholder - jsonplaceholder.typicode.com
json-server (json db)
rtmpserver - node-media-server - github.com/illuspas/Node-Media-Server - live streaming
video recording - obsproject.com
flash video player - npmjs.com/package/flv.js
draw.io - flowcharts/diagrams

Starting React App

npm install -g create-react-app
create-react-app <name of app>

npm commands
-g : global
--save : save to local project

babel - translates ESYYYY to ES5 (works on all browsers)

import - ES2015 Modules
vs.
require - CommonJS Modules

A react **Component** is either a:
- function - simple content
- class - everything else
  - pros

easier organization (subjective)
'state' system - easier to handle user input
understanding lifecycle events - easier app startup tasks

JSX
May have to use className inside of JSX instead of class to not confuse a js class with a jsx class

Objects are not valid as a React child - trying to show text inside of a javascript object, can't reference a javascript object inside of text:
- solution: objectText.text
{ } - referencing a js variable
you CAN use js objects in jsx but you can't display them in your app as text/image in the browser

Components:
Three Tenets of Components:
1. Nesting
2. Reusability
3. Configuration

Naive approach - try to make one generic component

making a Reusable and Configurable component:
- react components are made in ProperCase.js in src folder
- use React's 'props' system to make Component Configurable
- Props - pass data from a parent component to a child component and make it customizable (can only pass down from parent to child)
name of prop can be whatever you want, best practice to use the name of the function

export default Component; - make component accessible by outside files at end of file
to use a js Component inside of another - treat it as a JSX tag with <> rather than {}

Class-Based Components
with a functional callback have no good way to wait/re-render while gathering successful callback data, way around that is to use a class-based component to re-render itself with new information

Rule of Class-Components

1. javascript classes - ES2015, js inheritance works on prototypal inheritance, different from oop languages
2. must extend/subclass of a React.Component*
3. must define `render` method to return JSX

*2 - React expects class-based components have many methods borrowed from React.Component (`class App extends React.Component {`)

State in React Components

- must use class-based components
- props and states are different.
- js object that contains data for a single component
- update state to have component rerender
- state must be initialized when component is created* state *can only* be updated using the `setState` function

*(can be defined by constructor method or outside constructor method if componentDidMount does the data loading / lifecycle method)

Lifecyle Methods

main methods:

- componentDidMount - data loading (like constructor - best practices to not load data in constructor though, do it in cDM)
- componentDidUpdate - data loading every time component is updated
  - gets passed with previousProps
- componentWillUnmount - when you remove component from screen and data cleanup

other methods (used rarely):

- shouldComponentUpdate
- getDerivedStateFromProps
- getSnapshotBeforeUpdate

must use a class-based component to use lifecycle methods

render method called often, so don't initialize some request in the render method

- never want to directly assign state (except when you initialize the state in the constructor)
- return JSX and that's it

Most standards, config and helper functions at top and the functional component at bottom of component file

Event Handlers
Do not put parentheses at the end of a function when you pass a callback function to an event handler or it will be called every time it renders (i.e. this.onInputChange instead of this.onInputChange())
- onChange - will be called automatically when input changes
- onClick - when user clicks on element (supported by every element type)
- onSubmit - when user submits form

*must use these when using them in a standard jsx element

callback always run with a js 'event' object
- event.target.value property used for text input

name of callback doesn't matter (but best practice to call it onInputChange or handleInputChange for onChange)

Controlled vs Uncontrolled Component
uncontrolled - store value in the html input value / DOM
controlled - use the react component to store the input value

forms default to submitting input to server on enter (can prevent that with event.preventDefault())

"TypeError: Cannot read property 'state' of undefined"
solutions:
- this.x.bind(this) - older way of doing it
- turn function into an arrow function - needs babel (most common)
- use an arrow function in the jsx form to invoke the function

API Requests with React
Axios vs. Fetch
- Axios - 3rd party package, more code but easier
- Fetch - DIY

async keyword - allows for await

Ref's for DOM Access
when accessing DOM elements in React, use React Ref system (React.createRef)
gives access to single DOM element
pass Ref to JSC as a prop
key almost always = id

console only knows what's inside current DOM node after you expand the DOM

node, doesn't know anything else about it until then

Redux Notes
Action Creator : Person dropping off Form
V
Action : the Form
V
dispatch : Person receiving Form
V
Reducers : Insurance Departments
V
State : Compiled Department Data

avoid modifying existing data structures inside of reducers

To change the state of our app, call **Action Creator**
To produce an **Action** (how we want to change the data)
Which gets fed to the **Dispatch** function to make copies of the Action and
Forwards the Action to the **Reducers** (process the data and returns data)
To create a new **State** object where it waits until we need to update State again

Best practice: Name of reducers should be same name of state function keys in most cases

store.dispatch(actionCreator(action)) to modify state
state.getState() to get current state

React-Redux
root file inside of actions and reducers usually called index.js (best practice and functionally)

Named export allows export of many functions from a file

Provider takes care of interaction with the Store

mapStateToProps called with all of the state data from inside the store
- get data that's currently stored inside from redux store and pass that information into your component

configure connect by passing it this function (mapStateToProps)
will always get first argument of state and return an object that is going to show up as props inside of the component
use `ownProps` object in mapStateToProps to get props object inside component

- two params mapStateToProps gets often are `state` and `ownProps`
  - state - first argument, all the state out of our redux store
  - ownProps - 2nd argument, props object that shows up inside of our component

any time we want to update data with redux, must call Action Creator

passing `action creator` function to `connect` function will call `dispatch` function for us

Components responsible calling the Action Creators to get fetch the data
Action Creators responsible for making API requests

Sync Action Creators - instantly returns an action with data ready to go
Async action creator - takes some amount of time for it to get its data ready
- need to use middleware for this

Redux-Thunk
Middleware - plain js function called with every action that is dispatched
- can stop/modify an action
- most often used to deal with async action creators

with redux-thunk Action Creators can return an action object OR a function
- if returning an Action Object it must have a type and it can optionally have a payload

dispatch and getState() until function is an object
can manually dispatch an action once request is finished and update the data inside the store

define a function to return a function

Reducers
must return something (cannot return undefined)
produce 'state' (data) to be used inside of the app using on previous state and action object
must not reach out of its own function to decide what to return (reducers are pure)
- only return some manipulation of state and action object as input arguments

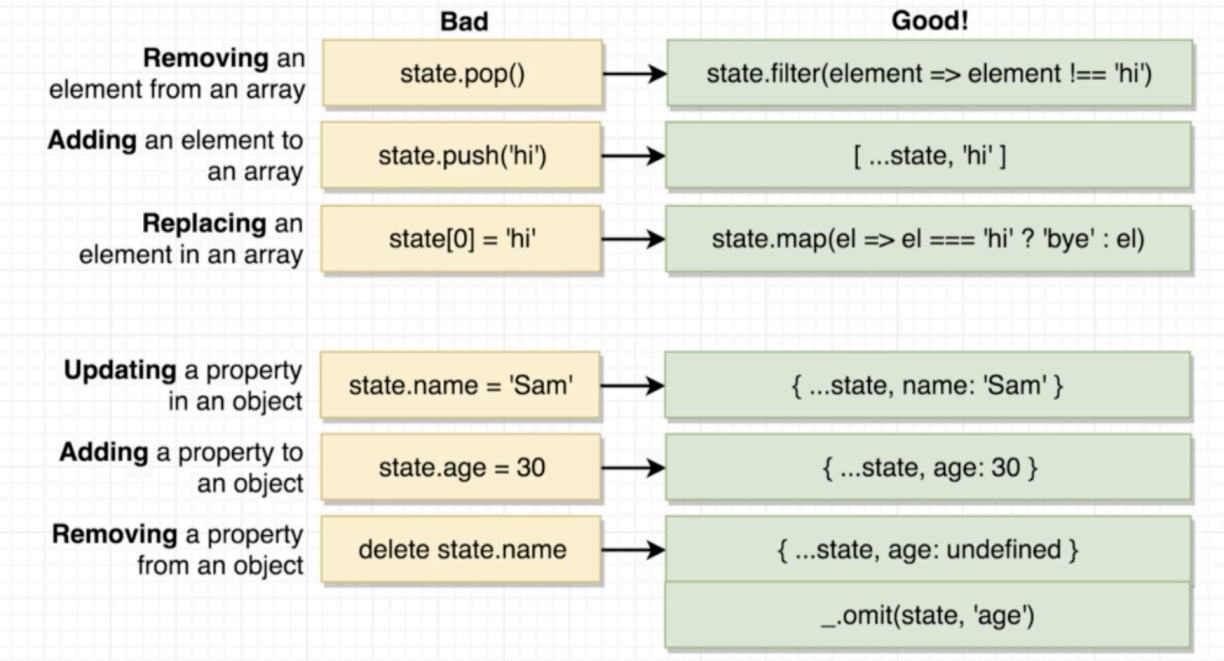
must not mutate its input state argument (BEWARE: misleading though)
- if you accidentally return the same value, then no update will be made to your application and vice-versa

if action is taken on site and notice nothing is happening, double check the type case strings in reducer and in actions index - use an actions/types.js file to keep

track of this instead of hardcoding into each file

JS Mutability + Comparison
strings and numbers immutable in js. arrays and objects mutable in js.
=== for numbers and strings check for value, instance for objects and arrays

State Manipulation In Reducers



removing a property from an object recommended to use _. (low-dash)

Overfetching Requests
Can use Memoization to not repeat calls on same fetch object - only runs on the first call
Can create a new Action Creator with the logic to uniquely fetch object

whenever you dispatch a function, redux-thunk will invoke it

try to keep Action Creators very granular - one major action per creator

await does not work with forEach syntax, would have to use a `.all` or something similar

React Router
extractedPath contains string in path, can use exact to make extractedPath have to match string in path
don't want to use anchor tags with react routers - browser changes pages and

loses data from previous page
- use Link from react router dom to replace anchor tags
- Link tags: anytime you want to navigate between pages, should use react-router-dom's Link component

any element that is not a child of a router cannot contain any React Router components

use <Switch> to look at the different routes and only show one for any given path. first path that is match will be shown and won't show any other route.

Three types of Routers
- BrowserRouter - uses the TLD or port as path (localhost:300/pagetwo)
  - most complicated to deploy usually
  - traditional server will respond with 404 if route is not defined, create-react-app server will return index.html if none defined - not typical behavior
  - have to make sure that server will do this when you deploy application using BrowserRouter
- HashRouter - uses everything after the # as path (localhost:300/#/pagetwo)
  - supposed to setup server to not look at anything after the hash - only used on client side
  - more flexible because it does not require more server setup
- MemoryRouter - doesnt use URL to track navigation (localhost:300/)

withRouter - reacttraining.com - withRouter gives us access to `history` object to pass on to our router

OAuth
access to user's profile
make actions on behalf of user

OAuth for Servers vs. JS Browser
- both result in token that a server can use to make requests on behalf of user
- Server *usually* used when we need user data when they are not logged into the app vs. Browser *usually* when they are logged in
- Server harder to set up because we need to store user info vs. Browser usually easy to setup with JS lib to automate flow
- only need client secret for Servers

Google OAuth js tag src='https://apis.google.com/js/api.js'
'gapi' in console to see object
reference gapi in app using window - window.gapi.load('client:auth2)
- window.gapi.auth2.getAuthInstance() to get auth instance

get is assigned to prototype of object for

callback functions setup as arrow functions

want Action Creators to change state of app (Auth state in redux store)
dispatch initial action when you finish initializing library to indicate whether user is signed in

Redux-Form
similar in nature to connect()
doesn't know a lot about what you're doing, will assume you're doing the right thing

validation - create function called validate outside of component class above export
- if validation is false: return an object where for each invalid field, use a key-value pair on the object with the { name of the field : error message}
- if errors object has keys with identical names to the name property of a Field, redux form will take error message and pass it to renderInput (component) function
- errors on `touched` (once a user clicks in and clicks out of an input
- use className="error" for form to show error messages, otherwise will display: none by default
  - if div with className="field" has "error" as well, field will turn red if error

initialValues - special redux form variable name where you can fill in initial values for specific property names of Field components in form
Avoid changes to "static" properties inside formValues object in onSubmit of an edit function. only pass in variable you want / need to change
- can use `_.pick` to specify which properties you want to change
- `_.pick(this.props.stream, 'prop1`', 'prop2', 'etc')`

REST-Based React Apps

- **Resource URIS**
  - Names and Verbs
    - use concrete names and not action verbs
  - URI Case
    - camelCase, snake_case, spinal-case
- **HTTP Methods**
  - GET - only retrieve data from server with URI
  - HEAD - same as ^ but transfers the status line and header section only
  - POST - send data to server

    - PUT - replaces *all* current reps (except id usually) of target resource with uploaded content
    - PATCH - replace *some* current reps of target resource with uploaded content
    - DELETE - removes all current reps of target resource given by URI
    - OPTIONS - describes communication options for the target resource
- **HTTP Headers**
    - General Header - both request and response
    - Client Request Header - only request
    - Server Response Header - only response
    - Entity Header - meta information or if no BODY is present
- **Query Parameters**
    - Paging - anticipate paging of resources early in design phase of API, paginate resources with default values when they are not provided by the client
    - Filtering - restricting number of queried resources by specifying some attributes and their expected values
    - Sorting - sort result of query on a collection of resources, contain name of attributes being sorted comma-separated
    - Searching - sub-resource of a collection, will have different format from the resources and collection itself
- **Status Codes**
    - 200 - OK - working
    - 201 - CREATED - new resource created
    - 204 - NO CONTENT - resource successfully deleted, no response body
    - 304 - NOT MODIFIED - cached data returned
    - 400 - BAD REQUEST - invalid request cannot be served, error should be in error payload
    - 401 - UNAUTHORIZED - requires user authentication
    - 403 - FORBIDDEN - server understood request but is refusing / access not allowed
    - 404 - NOT FOUND - no resource behind the URI
    - 500 - INTERNAL SERVER ERROR - stack trace should be logged and not returned as a response

can write out several early drafts of action creators if you follow RESTful conventions

Action	Method	Route	Response
List all records	GET	/streams	Array of records
Get one particular record	GET	/streams/:id	Single record
Create record	POST	/streams	Single record
Update ALL properties of a record	PUT	/streams/:id	Single record
Update SOME properties of a record	PATCH	/streams/:id	Single record
Delete a record	DELETE	/streams/:id	Nothing

when you dispatch an action of type delete the payload can be the id so you don't necessarily need to the .id property

`mapKeys` from lodash to concatenate objects to an object with a specific key in the object as the key for the object

Programmatic Navigation
BrowserRouter creates a `history` object
history object has ability to change and watch address bar
  – somewhat difficult to get history object from inside an action creator
instead, we can create the history object ourselves not BrowserRouter
  – use a non-BrowserRouter object (i.e. Plain Router)
history package installed automatically with react-router-dom ('history/createBrowserHistory')
use `.push(<path-to-send-user-to>)` to send user to a new page

URL-Based Selection
Selection Reducer - use a selectionReducer which will record what to change
vs.
URL-based selection - ID of object will be passed in the URL

":<variable>" can change path in Router to be any variable in App Component
match.params.<variable> will match the name of the variable^ in the Router path in App Component

if you refresh, it reloads app and store is empty (haven't fetched any data /

loaded up into app)

- ** with react-router, each component needs to be designed to work in isolation (fetch it's own data)
- needs to call the Action Creator to reach out to the API and fetch the data with the url
- every route needs to fetch its own data (use fetch function in componentDidMount())

React Portals

makes making modals easier
normally, all components are a child of div with id root, react portals allows to render modal as a direct child of a higher element (i.e. body)
if z-indexes are the same, whichever element comes later takes priority

import ReactDOM in modal (usually only import it in index.js)
if you click on an element that doesn't have the event define, it'll bubble up to the above it with the event defined
usually want to create a re-usable modal

React Fragments

allows you to return multiple elements without an enclosing without throwing off the styling of the div
can be shortened from `<React.Fragment>elements here</React.Fragment>` to `<>elements here</>`

The Context System with React

Props System - gets data from a parent component to a ***direct*** child component
Context System - get data from a parent component to ***any*** nested child component

can communicate data from highest to lowest level

2 ways to get information in and out of context:
default value OR in parent component, set up Provider object
V
context object
V
this.context / Consumer

export default React.createContext(<default context value>)
value passed into context object can be any js type

this.context
setup a `contextType` to link component to context object, then to reference info in the context object make use of `this.context` property
- must be called `contextType`

only elements wrapped in the provider will have access to the Context values
wrap element in `someContext.Provider` with tag `value=`
Provider is a react component of context

Provider "gotcha": Each time you render out an instance of the Provider, you're creating a new separate pipe of information

when using a Consumer, you don't need to provide contextType
always pass one "argument" (or child) as a function to the Consumer. that child will be called with whatever value is inside the pipe
- can use a helper method if that value is complex

use a Consumer any time you have multiple Context objects you're trying to get info from

Redux vs. Context
- distributes data to various components - BOTH
- centralizes data in a store - Redux?
- provides mechanism for changing data in the store - Redux?

To use Context in place of Redux:
- need to get data to any component in hierarchy ✓
- need to be able to separate view logic from business logic ✗
- need to split up business logic (not have a ton of code in a single file) ✗

Reasons to use Redux over Context:
- excellent documentation
- well-known design patterns
- tremendous amount of open source libs

Reasons to use Context over Redux:
- no need for an extra library

Challenges with Context over Redux:
- Hard to build a 'store' component with cross-cutting concerns

Hooks
can kinda introduce state and lifecycle methods to function-based components

- hooks make it easy to share logic between components
- write reusable logic (see useResources.js in hooks-simple project)

by default cannot use component state inside of a functional component

useState - allow functional component to use a component-level state
useEffect - allow a functional component to use lifecycle methods (componentDidMount, componentDidUpdate)
- (arrowFunction, array), when list is re-rendered, if the element inside the array are different, it will call the arrow function otherwise it will NOT be called
- allows you to "block updates"
- pass an empty array [] to only have useEffect rendered one time (aka componentDidMount only, no update needed)
  - if no second array is passed in, will be called every time
  - if second array is empty then empty again, it will **not** be called again (essentially identical to implementing componentDidMount)
  - if second array is of length 1 with same value in second call, it will **not** be called
  - if second array is of length 1 but diff values in each call, it will be called
  - if second array is of length 1 but two same value objects but not the same objects in each call, it will be called
  - if second array is > length 1 and same values, it will **not** be called
  - if second array is different length of 1st array, it will be called
- must return a cleanup function or nothing (cannot pass in an async function or a function that returns a promise), that function must be defined elsewhere then passed in to useEffect
  - or invoked immediately inside function (i.e. `(const func () => {})()`)

useContext - allow a functional component to use context system
useRef - allow a functional component to use the ref system
useReducer - allow a functional component to store data through a reducer
... and more ...

== [currentValue, setCurrentValue] = useState(initialValue)
== [this.state.resource, this.setState({resource: 'posts'}) = React function state={resource: 'posts'}