

# React-Redux-Node-GraphQL Notes

## REACT

## CHEATSHEET

<https://github.com/LeCoupa/awesome-cheatsheets/blob/master/frontend/react.js>

## INSTALLATION

- <https://reactjs.org/docs/getting-started.html>
- Add React to a Website
  - We can add React to an existing HTML page
  - Can try adding a react component to the viewer
  - Might not be possible, since the divs from GWT are transpiled to JS and created at runtime

## MAIN CONCEPTS

- Getting Started
  - <https://reactjs.org/docs/getting-started.html>
- modern JS
  - <https://gist.github.com/gaearon/683e676101005de0add59e8bb345340c>
  - 1. can define variables with let and const statements. For the purposes of the React documentation, you can consider them equivalent to var.
    - var = global variable
    - let, const = block variable (in a method)  
variables declared with let can be reassigned, while variables declared with const cant
  - 2. use class to define JavaScript classes.
    - the value of this in JS classes depends on how it is called
  - 3. => "arrow functions". are like shorter versions of regular functions. i.e. `x => x * 2` is `function(x) { return x * 2; }`.
    - arrow functions don't have their own this value so they're handy when you want to preserve the this value from an outer method definition.
- JSX
  - <https://reactjs.org/docs/introducing-jsx.html>
  - Can embed {expressions} in html

- Can close tags immediately
  - `<img src={user.avatarUrl} />` instead of `<img></img>`
- `React.createElement` to create an html element
  - **the two below are the same:**

```
const element = (
  <h1 className="greeting">
    Hello, world!
  </h1>);
const element = React.createElement(
  'h1',
  {className: 'greeting'},
  'Hello, world!');
```

## ● Creating / Rendering elements

- React elements are created as objects, unlike browser DOM elements. "ReactDOM" updates the DOM to match the React elements.
- **to render a React element into a root DOM node, pass both to `ReactDOM.render()`:**
  - **these are the same**

```
<div id="root"></div>
const element = <h1>Hello, world</h1>;
ReactDOM.render(element, document.getElementById('root'));
```
- React elements are immutable
  - most React apps only call `ReactDOM.render()` once, which gets encapsulated into stateful components.
- React only updates the parts that are necessary on a state change - ReactDOM compares the element and its children to the previous one, only applies an update to the DOM if needed

## ● Components and Props

- **React Components** - pretty much UI widgets.
- split the UI into independent, reusable pieces, think about each piece in isolation
- components are conceptually JS functions that takes in "props" (properties) return a react element
- **"props" (properties)** - input arguments of components
  - a function accepts props, return a react element
 

```
function Welcome(props) { return <h1>Hello, {props.name}</h1>; }
const element = <Welcome name="Sara" />;
ReactDOM.render( element, document.getElementById('root'));
```
  - ES6 way also works! **A JS function can be converted to an ES6 class** - <https://reactjs.org/docs/state-and-lifecycle.html>

```
class Welcome extends React.Component { render() { return <h1>Hello,
  {this.props.name}</h1>; }}
```
  - props are read only, cannot change its value like a `+= 1`. NO MUTABILITY!  
instead, change an element by updating its "state"

## ● State and Lifecycle

- Only with ES6 class can we have state. State gets defined in the constructor of an ES6 class for a react component
- Add lifecycle methods for a class when they are created (mounted) and destroyed (unmounted)
  - i.e. `componentDidMount`, `componentWillUnmount`
- change state with `this.setState()`. NO MUTABILITY!
  - cant modify state directly. use `this.setState()`

- can update state asynchronously. use arrow function in this.setState()  
 this.setState(function(state, props) {  
   return {  
     counter: state.counter + props.increment  
   });});
- info for an updated state then goes on to update all other places in the component that depend on that state

## ● Event Handling

- events are in camelCase
- pass the {function} as the event handler in JSX
- call e.preventDefault(); to prevent default behavior for an action
  - e is a react SyntheticEvent
- Use **bind to "this"** to make sure that "this" is actually defined when the function is called. We do this bind in a class constructor
  - // This binding is necessary to make `this` work in the callback
  - this.handleClick = this.handleClick.bind(this);
- Can also do a bind in a public class field. See this example --  
<https://babeljs.io/docs/en/babel-plugin-proposal-class-properties>
- Can also do a bind to an arrow function.
- Can pass a parameter i.e. id into event handler
  - Thinking in react - <https://reactjs.org/docs/thinking-in-react.html>

## ● Conditional Rendering

- can do it in an if (boolean) call this class/function else call another class/function or return null/dont render
  - can also call the inline version of the conditional
- can create distinct components that encapsulate specific behavior
- only need to render some of them, depending on the state of the application.

## ● Lists and Keys

- Lists
  - Render multiple components into a list: Use JSX map function for lists to get it into a collection of elements, then render it to the DOM
  - In the example below, we loop through the `numbers` array using the JavaScript `map()` function. We return a `<li>` element for each item
  - function `NumberList(props) {`
  - const `numbers = props.numbers;`
  - const `listItems = numbers.map((number) =>`  
     `<li key={number.toString()}>`  
       `{number}`  
     `</li>`
  - `);`
  - return (`<ul>{listItems}</ul>`
  - `);}`
  - 
  - const `numbers = [1, 2, 3, 4, 5];`
  - 
  - ReactDOM.render(

- `<NumberList numbers={numbers} />`,
- `document.getElementById('root');`
- Keys
  - Keys are used to ID elements in an array to perform crud operations on them
  - if data already comes with an id, should use that as your key
- `const todoItems = todos.map((todo) =>`
  - `<li key={todo.id}>`
  - `{todo.text}`
  - `</li>`
  - `);`
- Call keys on the elements inside the map() call since we are adding an identifier for each list element as we are looping through it.

## Forms

- Controlled Components - the React component that renders a form also controls what happens in that form on the subsequent user input
- for `<input>`, `<textarea>`, and `<select>`
  - in the constructor
    - `this.state = {value: ""};`
  - create a handleChange function in the class
    - `handleChange(event) { this.setState({value: event.target.value}); }`
  - then the form upon render
    - `<input type="text" value={this.state.value} onChange={this.handleChange} />`

## Lifting Up State

- There should be a single “source of truth” for any data that changes in a React application. Usually, the state is first added to the component that needs it for rendering. Then, **if other components also need the state, you can lift it up to their closest common ancestor**. Instead of trying to sync the state between different components, you should rely on the top-down data flow.
- Lifting state involves writing more “boilerplate” code than two-way binding approaches, but as a benefit, it takes less work to find and isolate bugs. Since any state “lives” in some component and that component alone can change it, the surface area for bugs is greatly reduced. Additionally, you can implement any custom logic to reject or transform user input.
- If something can be derived from either props or state, it probably shouldn’t be in the state. For example, instead of storing both `celsiusValue` and `fahrenheitValue`, we store just the last edited temperature and its scale. The value of the other input can always be calculated from them in the `render()` method. This lets us clear or apply rounding to the other field without losing any precision in the user input.

## Use Composition, not Inheritance



- composition - i.e. a more “specific” component renders a more “generic” one and configures it with props
  - have all the classes extend `React.Component`, then use a tag `<SomeClass>` when it is being rendered to call `AnotherClass`
- i.e.
- class `UserNameForm` extends `React.Component` {
- `render() {`

- return (
- <div>
- <input type="text" />
- </div>
- );
- }}
- class **CreateUserName** extends React.Component {
- render() {
- return (
- <div>
- < **UserNameForm** />
- <button>Create</button>
- </div>
- )
- }}
- ReactDOM.render(
- (<div>
- <**CreateUserName** />
- </div>), document.getElementById('root'));

## Thinking in React

- Step 1: Break The UI Into A Component Hierarchy
  - draw boxes around every component (and subcomponent) in the mock and give them all names
- Step 2: Build A Static Version in React
  - render the UI but have no interactivity. dont add state at all yet
- Step 3: Identify The **Minimal** (but complete) Representation Of UI State
  - figure out what the props are, and what the states are
- Step 4: Identify Where Your State Should Live
  - identify which components mutates, or owns, the states.
  - **For each piece of state in your application:**
    - Identify every component that renders something based on that state.
    - Find a common owner component (a single component above all the components that need the state in the hierarchy).
    - Either the common owner or another component higher up in the hierarchy should own the state.
    - If you can't find a component where it makes sense to own the state, create a new component solely for holding the state and add it somewhere in the hierarchy above the common owner component.
  - build an app that renders correctly as a function of props and state flowing down the hierarchy (top down)
- Step 5: Add Inverse Data Flow
  - **support data flowing the other way:** the form components deep in the hierarchy need to update the state

# HOOKS

- <https://reactjs.org/docs/hooks-overview.html>
- React plans to slowly move away from classes towards hooks, since classes has a hard time understanding stateful logic between components when scaling
  - hooks - let you use state and other React features without writing a class.
- Hooks Rules
  - Only Call Hooks at the **Top Level** of a **React Function**
    - This makes sure that hooks are called in the same order each time function is called.
    - Don't call Hooks inside loops, conditions, or nested functions
  - Only Call Hooks from React Functions
    - DO NOT call Hooks from regular JS functions.
    - Instead, you can:
      -  Call Hooks from React function components.
      -  Call Hooks from custom Hooks
- Hooks API
  - <https://reactjs.org/docs/hooks-reference.html#useeffect>
  - Basic Hooks
    - useState
      - **this page shows the differences between setting state with a hook vs with a class - <https://reactjs.org/docs/hooks-state.html>**
    - useEffect
    - useContext
  - Additional Hooks
    - useReducer
    - useCallback
    - useMemo
    - useRef
    - useImperativeHandle
    - useEffect
    - useDebugValue

## ADVANCED GUIDE

- Refs and the DOM
  - We do use this in DBA Studio
  - Refs - use refs to access DOM nodes or React elements created in the render method
  - Typically in React, only need to use props (top down flow) for parent components to interact with their children. Modify child by re-rendering it with new props
  - However, several cases where we need an escape hatch to modify the instance of the React Element/DOM Node
    - Managing focus, text selection, or media playback.

- Triggering imperative animations.
  - Integrating with third-party DOM libraries.
- Not recommended to do this a lot in the code
- **RenderProps**
  - We do use this in DBA Studio
  - A component with a render prop takes a function that returns a React element and calls it instead of implementing its own render logic
  - i.e.
  - class MouseTracker extends React.Component {
  - render() {
  - return (
  - <div>
  - <h1>Move the mouse around!</h1>
  - <Mouse render={mouse => (
  - <Cat mouse={mouse} />
  - )}>
  - </div>
  - );
  - }
- **Code Splitting**
  - Don't know if we use this in DBA Studio
  - can “lazy-load” just the things that are currently needed by the user, which can dramatically improve the performance of your app
- **Context**
  - We don't use this in DBA Studio
  - React's version of a singleton pattern to share global parameters. Context provides a way to share values like these between components without having to explicitly pass a prop through every level of the tree.

## NODE JS

### NodeJS Cheatsheet

<https://github.com/LeCoupa/awesome-cheatsheets/blob/master/backend/node.js>

### NodeJS API

<https://nodejs.org/api/>

# MAIN CONCEPTS

- **Architecture**
  - single thread serving all the requests
  - event loop waits for events, uses event emitter to bind events to listeners
  - triggers a callback when an event is detected
  - requests don't block each other
  - asynchronous
    - Traditional serverside technology like apache is synchronous. need to finish serving a request before starting another one
- **used for**
  - REST APIs - accept get/post requests to get data from a database and serve it to a client
- **NPM - node package manager**
  - used to install node modules
  - nodes get installed into node\_modules folder
- **package.json - manifest file for Node**
  - tells npm how to install the packages
- **Files that use node in dbastudio are in the config and scripts folders**
  - public folder
    - index.html, default.css files
  - scripts folder
    - npm start -- start.js
    - npm build - build.js
    - npm test -- test.js
  - we don't use express
- **webpack - bundles scripts together**
- **webpack-dev-server - server for development**

# DEFINITIONS

- **Promise (asynchronous programming)**
  - [https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global\\_Objects/Promise](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Promise)
  - a value not necessarily known when the promise is created.
  - associate handlers with an asynchronous action's eventual success value or failure reason.
  - This lets asynchronous methods return values like regular methods. we return a promise to supply the value at some point in the future.
  - Promise is in one of these states:
    - pending: initial state, neither fulfilled nor rejected.
    - fulfilled: meaning that the operation completed successfully.



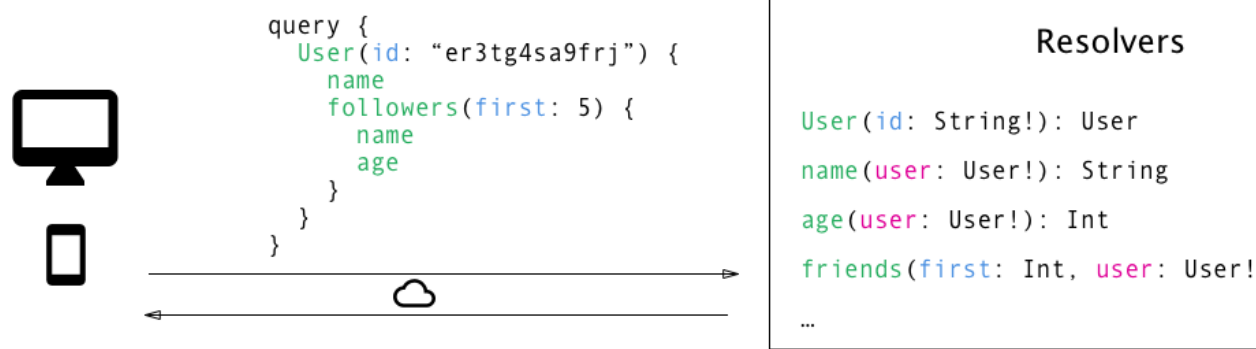
- rejected: meaning that the operation failed.
  - await - makes an async function pause until there is an answer from the [Promise](#). It can only be used inside an [async function](#).
- Callback (asynchronous programming)
  - A **callback** is a function which is called when another task is finished.
  - this is to prevent blocking and allows other code to run in the meantime. pretty much the staple of asynchronous
  - Using Callback , Node.js can process a large number of requests without waiting for any function to return the result which makes Node.js highly scalable
  - Express - framework for Node.js to create a web application
- Swagger
  - service we use to write API documentation for REST APIs
- polyfill -
  - a browser fallback, made in JavaScript, that allows functionality we expect to work in modern browsers to work in older browsers, e.g., to support canvas (an HTML5 feature) in older browsers

## GRAPHQL

<https://www.howtographql.com/>

- Replacement for REST calls for fetching data from an API
- REST
  - multiple endpoints to REST API
  - typically overfetches data, i.e make too many calls to the API with multiple endpoints and gets extraneous data more than you need
  - can also underfetch, i.e. an endpoint doesn't provide enough of the required information so client has to make extra calls to retrieve the bits of missing data
  - REST is generally inflexible because the data structures are fixed
  -
- GraphQL
  - one endpoint on GraphQL API
  - only send a single query to the GraphQL server with concrete data requirements. server then responds with a JSON object where these requirements are fulfilled.
    - this is useful since changes on the client can be made without any extra work on the server.
      - **query** - the client sends info to the server to express its data needs (i.e. fields data)
      - because clients can specify their exact data requirements to the backend, backend doesn't need to make adjustments when frontend requirements change
      - the structure of the data that's returned from the server is flexible and lets the client decide what data is actually needed.
  - Schema Definition Language (SDL) and Strong Type System -
    - <https://www.prisma.io/blog/graphql-sdl-schema-definition-language-6755bcb9ce51>
    - **schema** - specifies the capabilities of the API and defines how clients can request the data
      - schema is the contract between the client and the server to define how a client can access the server data

- Once the schema is defined, the teams working on frontend and backends can do their work separately since they both are aware of the definite structure of data being sent
- A schema is simply a collection of GraphQL types. However, when writing the schema for an API, there are 3 special root types:
  - **queries** - see above
  - **mutations** - make CRUD operations to info for fields stored in the backend
  - **subscriptions** - client establishes a real-time connection with the server for updates. i.e. when other clients update the server with info, the real-time update gets relayed to our client
- GraphQL is only as a specification. This means that GraphQL is no more than a long document that describes in detail the behaviour of a GraphQL server.
- **Resolvers** - where the magic of flexibility in GraphQL happens



- Each field in the query corresponds to a [resolver function](#). a resolver function fetches the data for its field.
  - GraphQL calls the required resolvers for a query to come in and fetch the specified data.
  - server returns packaged up data from the resolvers to return to the client
- Questions
    - Is GraphQL a Database Technology?
      - No. GraphQL is often confused with being a database technology. This is a misconception, GraphQL is a *query language* for APIs - not databases. In that sense it's database agnostic and can be used with any kind of database or even no database at all.

## REDUX

Look into react-plugin-workspace in KevinLe/icn repo

<https://github.ibm.com/fernando-gomez/icn/compare/react-plugin-workspace...Kevin-Le:react-plugin-workspace>

- Redux is a STATE CONTAINER for managing application state JS apps (i.e. React)
- Make state mutations predictable by imposing certain restrictions on how and when updates can happen. These restrictions are reflected in the three principles of Redux:
  - Single source of truth -
    - **store** - object that holds the application's state tree.
    - **state** of ENTIRE application is stored in an object tree within a single **store**.
    - only one single store in a Redux app, as the composition happens on the reducer level.

- This makes it easy to create universal apps, as the state from your server can be serialized and hydrated into the client with no extra coding effort.
- single state tree also makes it easier to debug or inspect an application
- State is read-only -
  - **action** - object representing an intention to change the state and is the only way to get data into the store.
  - ONLY way to change the state is to call an [action](#)
  - Any data, whether from UI events, network callbacks, or other sources such as WebSockets needs to eventually be dispatched as actions.
    - This ensures that neither the views nor the network callbacks will ever write directly to the state. Instead, they express an intent to transform the state.
  - Because all changes are centralized and happen one by one in a strict order, there are no subtle race conditions to watch out for.
  - As actions are just plain objects, they can be logged, serialized, stored, and later replayed for debugging or testing purposes.
- Changes are made with pure functions - with [reducers](#).
  - **reducers** - pure functions to specify how the state tree is transformed by actions
    - take the previous state and an action, and return the next state.
    - must return new state objects, instead of mutating the previous state

how to use each function - <https://redux.js.org/glossary>