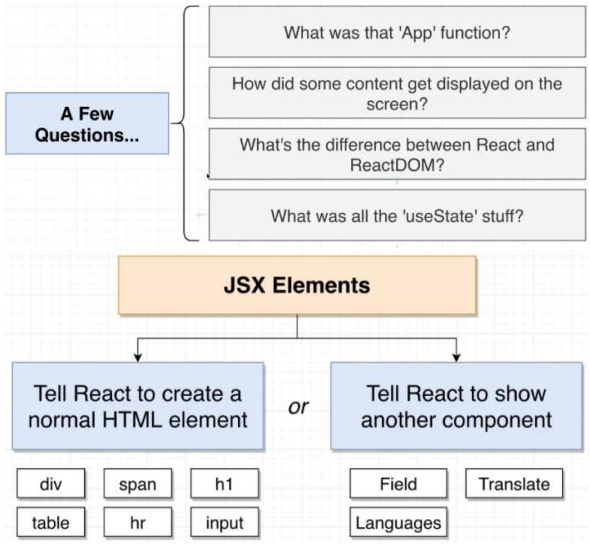
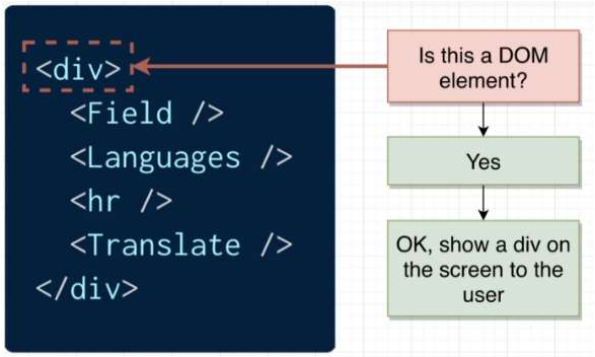
Introduction to React

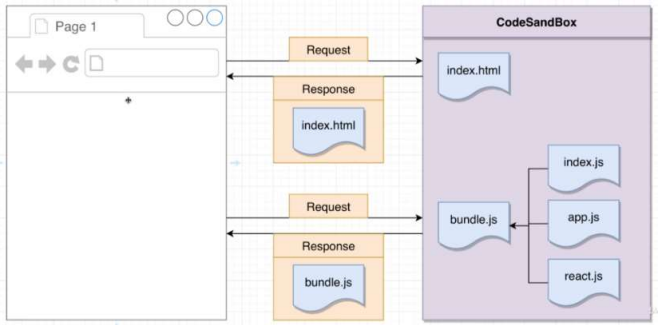


1) export default function App() {…}
   - The App() function is a **React Component** (two jobs: (1) produces JSX and (2) handles user events), which is a plain JS function that returns JSX (a set of instructions to tell React what content to show on screen)
   - We told React to use JSX to show things like a div, hr, or even other components.

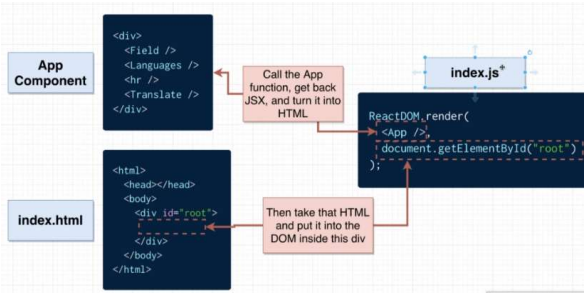


   - React will iterate over every element, ask if it is DOM element (if yes, show). If not, it is a component, it will call the component function and inspect all the JSX we get back from that component.

2) Screen display

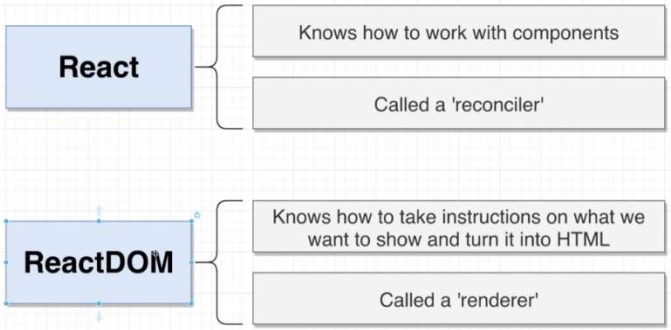


   - The mini browser makes a HTTP Request to the server. It will receive a Response with index.html. It then loads the HTML file and parses it.
   - One of the script tags in the HTML will tell it to make another Request to get a bundle.js file that is a combination of all the other JS files.



   - The first JS file to be parsed is index.js, which contains a ReactDOM.render() call.
   - It references the App() function, which means calling it and turning it into HTML.
   - The second reference is where we want to show our React project inside of index.html.

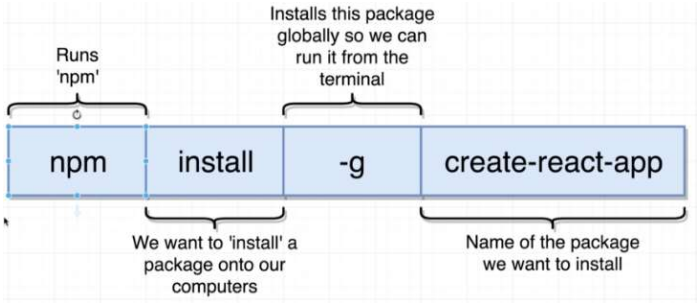
3) React VS ReactDOM



   - React is a **reconciler** which allows organisation of different components to work together, how to call a component function, how to get the JSX, and how to iterate through the elements to display as DOM or call another component function.
   - ReactDOM is the **renderer** which knows how to take a set of instructions on different elements and create them. It turns JSX into HTML in a DOM and presents it to user.

4) useState (only functional components)
   - It is a function for working with React's **"state" system**. This state system is all about <u>managing data</u> inside your application, specifically data that will change over time.
   - Use the state system to get React to update the content on the screen.

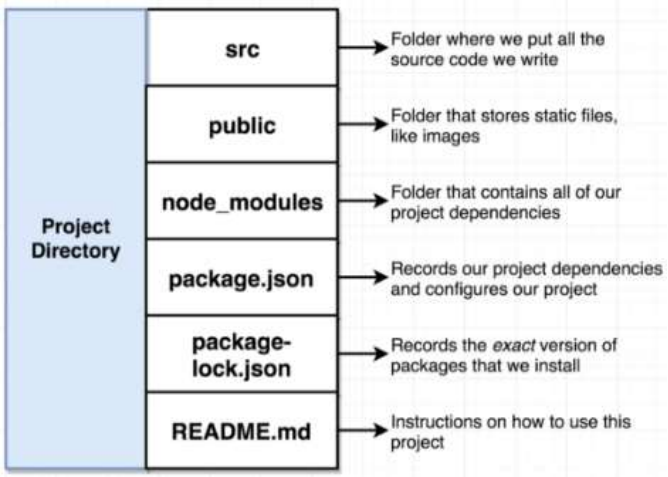
<u>Codes</u>



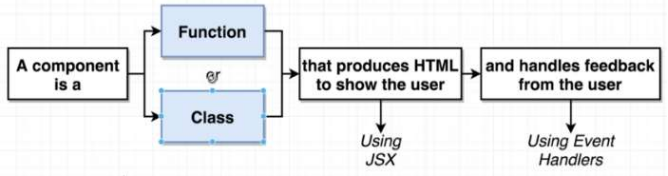
- **npm** (node package manager) has many subcommands (eg **install**). We want to install it globally (**-g**).
- **create-react-app** is a popular package used to generate React projects.
- **npx create-react-app my-app** is the alternative and recommended method.
- You'll realise that a React project relies on many dependencies, especially **Webpack, Babel, and Dev Server**. These are behind-the-scenes libraries that you don't have to setup.
  - Babel is merely a tool that backdates your latest code versions to an older version so un-updated browsers still work.

<u>React Structure</u>

- What is in the entire React project:



- What is a React Component?



  - It is either a function or class. Our example App() was class-based. A function-based would be like "const FunctionComponent = () => {…};".
  - Its two purposes are to produce the HTML to be displayed, and also to handle feedback.

Chapter 2: What is JSX?

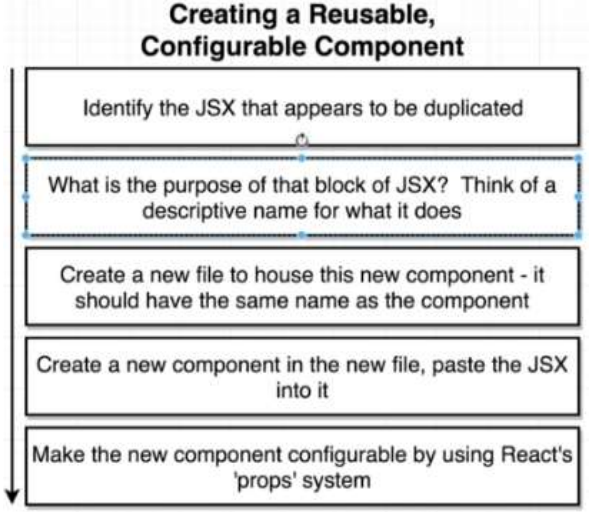
- The HTML-looking code inside App() is JSX, and is NOT html. **In fact, it IS a type of JavaScript.** It is just that browsers don't understand JSX so Babel converts it to normal JS.
- Try babeljs.io to see the conversion.
- Main purpose is to make it easier to code.

JSX VS HTML

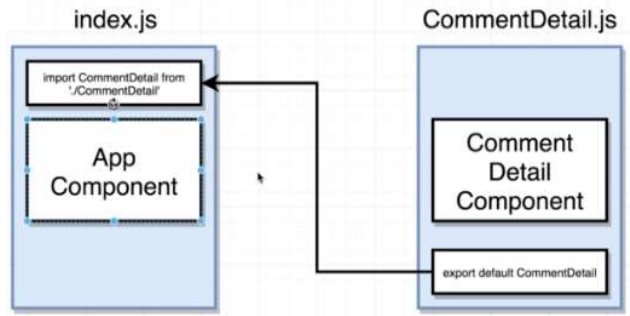
- Differences
  - Custom styling uses different syntax
  - Adding a class to an element uses different syntax
  - JSX can reference JS variables
- Eg HTML:
- Eg JSX: <div style={{ backgroundColor: 'red' }}>
- Eg HTML 2:
- Eg JSX 2: <div style={{ border: '1px solid red' }}>
- Eg JSX 3: <button style={{ backgroundColor: 'blue', color: 'white' }}>Submit</button>
  - The outer curly brace indicates you want to reference a JS variable inside JSX.
  - The inner curly brace indicates a JS object.
  - Hence, we are providing a JS object (sort of like a dict) where all the keys of the object reference a different property.
  - All property names if have dash: remove dash, then capitalise the second letter.
- There is no hard and fast rule about using "" vs '', but the community standard is usually "" for JSX, and '' for everything else (eg CSS styling).
- In HTML you use *class*, but in JSX you use *className*. This is to avoid collisions because in JSX the word *class* is used for class declaration instead of class attributes.
- JSX can take a JS variable and easily print it inside the JSX block. Remember that a single {} means it is referencing a JS variable (in contrast to a JS object which requires {{}}). So we can declare a variable or function, and then use / call it later by putting it inside the single {}.
  - {Variables} works with:
    - Strings, numbers
    - Lists (output will be concatenated with no spacing)
    - Functions
  - {Variables} do not work with JSX such as:
    - JS objects when shown as text. This is because React doesn't know how to show objects as text. ➔ BUT YOU CAN call a JS object and then its attribute if it is a string / number / list, ie object.text.

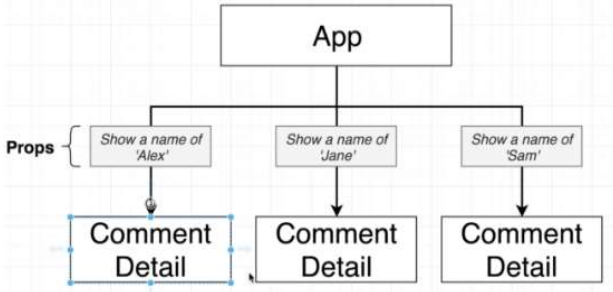
Chapter 3: Components

- The three big concepts to know are:
  - Component **Nesting**: a component can be shown inside of another
  - Component **Reusability**: make reusable components
  - Component **Configuration**: able to configure a component when it is created
- For CSS style: use **semantic-ui.com** for open-source CSS framework. Use the CDN easier.
- For fake images, use the **Faker library** *faker.image.image()* or the **Unsplash API** *https://source.unsplash.com/random*.
- Reusable:

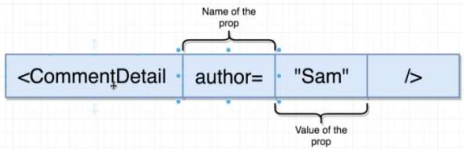


- Component Nesting:



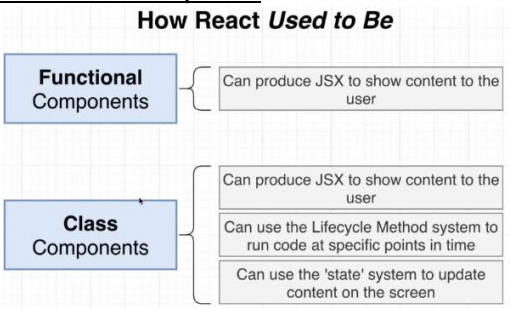


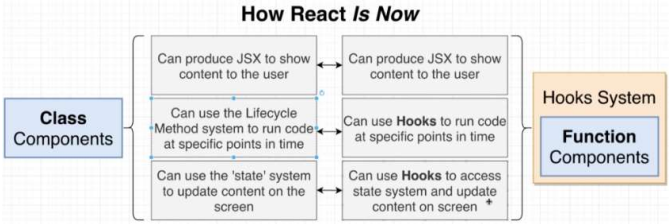
- The **props system** (props = properties) in React is a system for passing data from a parent component to a child / nested component. The entire goal is to communicate data from the parent down to a child to customise the child by customising how the component looks or how a user interacts with it.



- The name of the prop can be anything.
- In our JS, the *props* object is a JS object that contains all information that was passed from the parent.
- To put a component within another component, the outer component need to have an opening and closing tag, and the inner component is nested inside. The inner stuff can be plaintext or JSX as well.
- When you nest things inside of a component, eg <ApprovalCard><CommentDetail/></ApprovalCard>, you can use **props.children** to work with CommentDetail inside of the parent component's code.
- What if you forget to put in props? You can always set up default props by doing **component.defaultProps = { prop1: 'xxx' };** instead written inside that component's JS file.

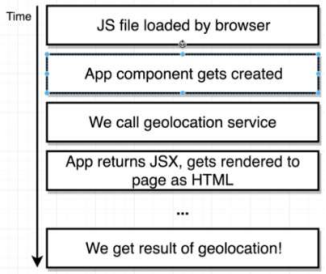
Chapter 4: Class-Based Components





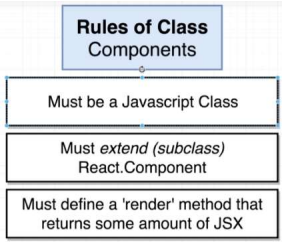
- Back then, functional components did not have access to the lifecycle method system or the state system which made it restrictive.
- But now with the Hooks system, functional components can execute code at specific points in time, which is similar to the lifecycle method system. Plus, it can make use of the state system.
- We will start with class components first because it is easier, can use 'state' to easily handle user input, and understand lifecycle events.

Timeline of Events



- The geolocation part takes several seconds, and the functional method doesn't really have a good way of waiting until the position to be calculated to subsequently be included in the success callback to be inserted into the JSX.

Refactoring Functional to Class

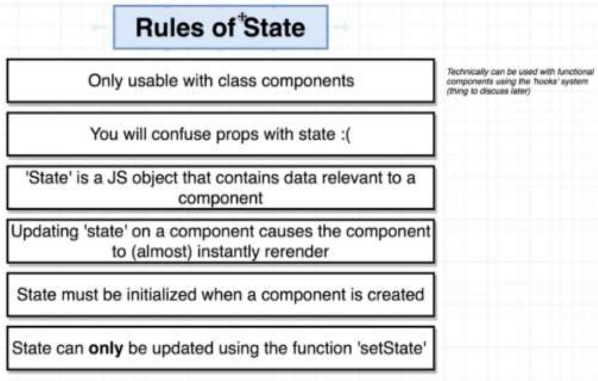


- Top reason is because React functional components cannot handle async (without Hooks/Redux), because a promise will be returned but React cannot render a Promise (it is not JSX so will have error). FCs will not rerender too – only rerender if the props they receive from the parent change.
- **Remember that class-based must define render()!**
- Some details:
  - Why must extend React.Component? Because render() is one of the new methods that we want to introduce, but we also expect there are other methods for the App class. These other methods are borrowed from the R.C class; we are borrowing functionality in our subclass.
  - Inside R.C there is the constructor() function which helps to initialise state when the app first renders. If you want to edit its behaviour and yet avoid completely overwriting it, you inherit the parent's constructor() with super():

```
constructor(props) {
    super(props);
    this.state = { lat: null, errorMsg: ''};
};
```

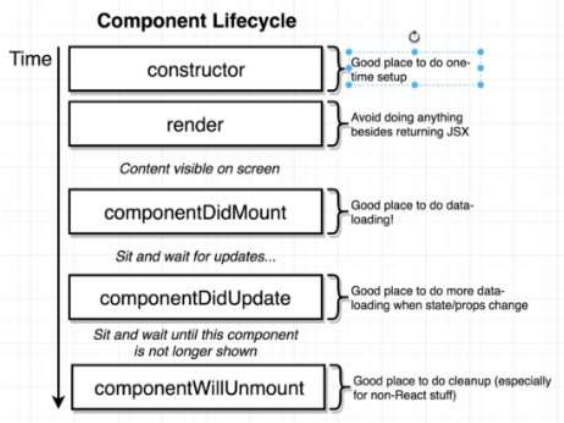
  - However, you may choose to completely replace the above step with componentDidMount() which is more simpler to use. (And by documentation, not supposed to do data loading inside constructor() even if possible, to centralise all the data loading inside cDM().)
  - You may also choose to do this as an identical alternative to using constructor():
    - state = { field1: null, field2: '' };
    - Use cDM() to this.setState({ field1: xxx }); to change state.

The State System



- Important points:
  - State is a JS object that contains data relevant to a component; it excludes irrelevant data. For eg, keep user's current latitude, discard unused info.
  - Updating state rerenders the component (good thing – make it automatically dynamic!). **It also rerenders any children that are showing as well.**
  - Only updated by '**setState'**.

Lifecycle Methods

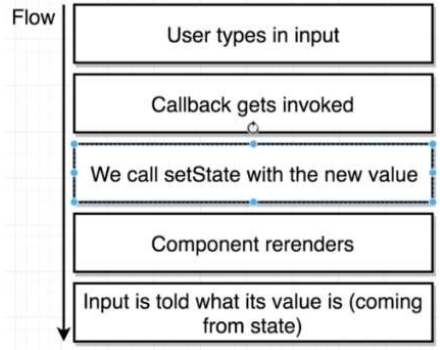


- A component **lifecycle method** is a function that we can optionally define inside our class-based components. If so, they will be called automatically by React at certain points during a component's lifecycle.
- Lifecycle = being created, show up in DOM, call setState to re-render, being removed.

- The above picture:
  - constructor() is optional to define, called when component is created.
  - render() not optional, but technically a lifecycle method. Called some time after constructor(). Content is then visible on screen.
  - Immediately after a component first renders on screen, componentDidMount() is called once. While both constructor() and cDM() allows data loading (there is no technical error in doing so), best practices is do it in cDM() so you always centralise all data loading stuff for clearer easily maintained code.
  - When setState is called that updates the component, render() is first called, then componentDidUpdate() is called. Can be called multiple times based on how much it is updated.
  - To stop showing component, componentWillUnmount() is called.
- There are 3 more lifecycle methods but very infrequently used: shouldComponentUpdate(), getDerivedStateFromProps(), and getSnapshotBeforeUpdate().

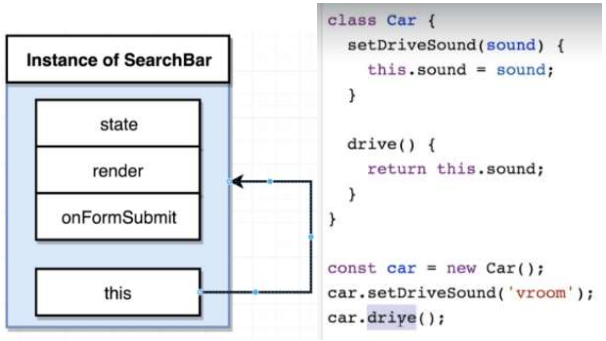
Chapter 5: User Forms and Events

- See project.
- Mainly using onSubmit, onClick, and onChange.
- Controlled VS Uncontrolled Elements



  - All this happens in the same component.
  - Basically in uncontrolled, whenever we type smth, the value will pass through React and go to the HTML side, and will be forgotten on the React side.
  - As developers, we don't like to store info inside HTML elements. Our React side should always have the info.
  - Hence to solve this, whenever user types something, it will be saved into state, and then use the value of state to replace the user's input. While on the surface no change, now our system memorises what user types in.

this



- this is problem mainly for CALLBACK FUNCTIONS, ie if you pass a function as an argument inside another function, that argument-function's *this* is undefined.
- this references the current instance. You use it to reference the properties (eg state, render, onFormSubmit), which will return you the instance's state/render/oFS object.
- When we do the code "this.sound", then what "this" is referring to is the object to the left of the dot when it is called. So: don't look at the method definition itself, but **look where we call the method, then look to the left of the dot**. Hence, "this" is referring to the instance of the car.
- So the error is because to the left there is no dot, aka undefined.
- this also failed because the function onFormSubmit is just a method definition, and it not tied to any "this". See here.
- **In other words, callback functions have to be bound or else the value will not be what you expect.**
- If you want to use the bind method with the constructor(). Also see this. Or this.
- Can also turn the callback function definition into an arrow function. In other words, wrap the callback as an arrow function.

Parent to Child / Child to Parent

- Parent to Child: use props.
- Child to Parent: pass a callback as a prop from the parent to child, and the child will call that callback.

API Requests

- Any time you have to do asynchronous operations, either (i) use promises, or (ii) use async-await syntax.
- There are two ways to do this: axios (a third party package) or fetch (browser function).
  - npm install --save axios
  - axios.create({
    - baseURL: 'http://….com',
    - *headers: { Authorization: 'Client-ID abcdef123'}, // for unsplash*
    - *params: { part: 'snippet', type: 'video', key: KEY } // for youtube*
    - })
  - .get('/path/page', { // address
    - params: { query: xxx } // customise request. These are the arguments behind the url, ie ?query=xxx.
    - });
- Async-await 1: async fn(term) { const x = await axios.get() }
- Async-await 2: fn = async (term) => { const x = await axios.get()}
- A fulfilled result of axios is an object that contains these:
  - data: {} // response provided by server
  - status: 200 // http status code
  - statusText: 'OK'
  - headers: {} // all the headers the server responded with
  - config: {} // the config provided to axios for the request
- Here is a list of axios commands:
  - axios.request(config)
  - axios.get(url[, config])
  - axios.delete(url[, config])
  - axios.head(url[, config])
  - axios.options(url[, config])
  - axios.post(url[, data[, config]])
  - axios.put(url[, data[, config]])
  - axios.patch(url[, data[, config]])

Images / Working with Lists / Keys

- Remember that **every element inside of a list of JSX elements needs to have a key property.** This is because when a new element is added to a list to be rendered, React will compare that list with what is on the DOM, and only add those new ones. To allow this to happen efficiently, it needs the keys to act as an ID for each element . See here.
- When using lists in React, highly recommended to define a key ~~on the root tag~~ that you are returning inside the map() statement~~, ie the most parent tag for each element~~.
- React has a **ref (reference) system** that gives access to a single DOM element that is rendered by a component. This allows us to get details of the DOM element such as its pixel height, without relying on document.qS() in React. See pics_trial's ImageCard.
- We create refs in the constructor() with React.createRef(), assign them to instance variables, then pass to a particular JSX element as props. Doing so will allow other functions in the component to work with ref. It will refer to that JSX element.
- Example:
  - class ABC extends R.C {
    constructor(props) {
    super(props);
    this.abcRef = React.createRef();
    }
    …

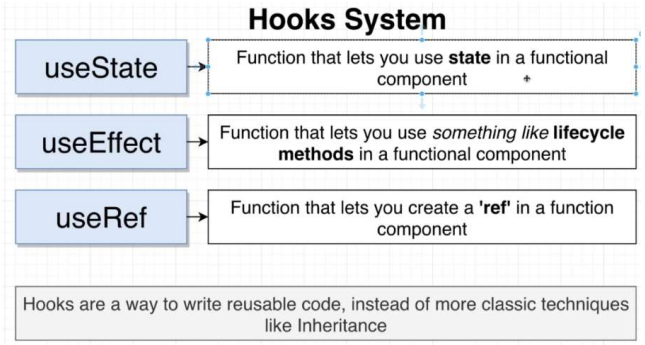
return
  - }
- In theory, can also assign to state but since it's not dynamic (will never change), we don't do so.
- **ADDITIONAL KNOWLEDGE:** If your result comes with HTML tags inside, how to handle?
  - Use replace() to replace all tags to blanks.
  - (dangerous method that React doesn't want you to know) In the HTML tag, add a prop called ***dangerouslySetInnerHTML****={{__html: result.x}}*. This will display those HTML tags. However, anytime you take a string from a third-party API, you might introduce a security hole in the form of **cross-site scripting XSS attacks**. This means accidentally rendering untrusted HTML to allow a hacker to execute malicious JS. When that JS runs, it replaces the entire screen with fake shit. Hence, only use if you trust.
- **ADDITIONAL KNOWLEDGE**: setTimeout(arrowfn, millisecs).
  - Returns an integer that is an identifier for the timeout.
  - Can clear the timer with clearTimeout(identifier).
  - (for functional components) Good to include it in useEffect's cleanup.

SUMMARY OF CLASS-BASED COMPONENTS

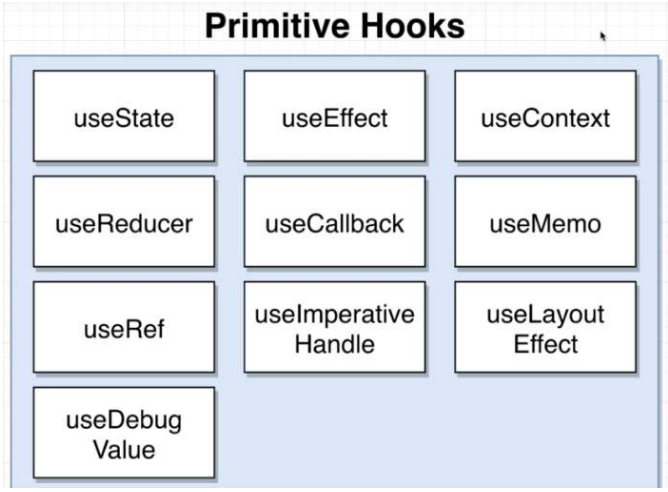
- Class-based components are easier to read, because you know which is the render() part, and the rest are just other parts. Function-based all parts look the same, hard to see immediately which is the "render()". *Not very strong case…*
- class App extends React.Component {…}; export default…;
- Need to initialise state with this.state = {…} OR state = {…}. Only change with this.setState{…}.
- Can initialise with constructor(props), with super(props). But actually better to do in cDM() to centralise all data loading inside it, instead of some here some there.
- Must have render() method, that runs the inside code every time the component renders/rerenders.
- Has built-in lifecycle methods: componentDidMount() for initialisation, componentDidUpdate() for rerendering, componentWillUnmount() to clean stuff when removing.
- When you rerender a component, its children all get rerendered too.

Chapter 6: Hooks System

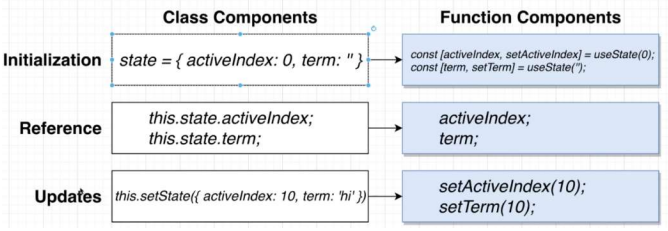
- Basically to give functional components more functionalities. Previously, doesn't use state or lifecycle methods, so Hooks will change that.



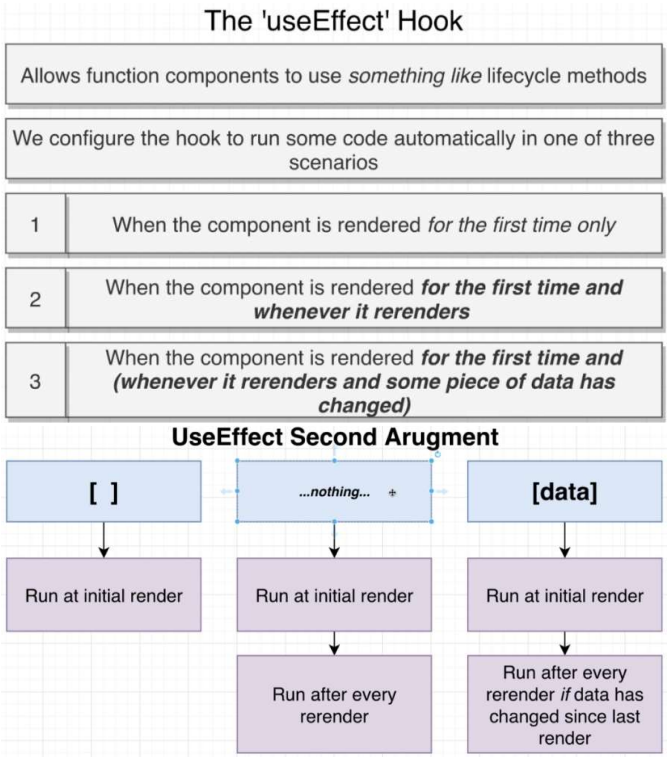
- These 3 functions for functional components:
  - useState lets you use **state**.
  - useEffect lets you use similar **lifecycle methods**.
  - useRef lets you create a '**ref**'.



- With Hooks, we get access to "Primitive Hooks" (not a real term). These 10 functions give more functionality to function components (FC). We will learn what each does, then learn how to use them to write our own custom hooks.



- **useState** gives you access to the state system inside a FC. When you call useState(), its LHS always returns two variables: first is the current state, second is a function that updates the first; similar to *this.state.count* and *this.setState* in a class.. The RHS inside the brackets () is the initial state. The names can be anything – if you are keeping track of a person's name then you can call it *name* and *setName* and RHS is " "; if counting numbers, then can be *counter*, *setCounter*, and 0. When a state is changed, the entire component rerenders. **Take note that even when rerendered, state will not be reset to default once setter is used even one time; the default initialisation value will fall away and not be used anymore.**



- **useEffect** allows us to write out some code that detects that our component is re-rendering and some piece of information has changed, and then run some additional code; aka it detects changes to cause a change.
  - **The main point will be using a uE() hook to update a state. This is very common!**
  - A clue that you'll need to use the uE() hook is that when you need to set up an event listener inside a component (because upon event, something changes). Attaching an event listener is usually done with uS() hook with the only first render argument.
  - Three second args:
    - 1) [ ] = Runs when component renders for first time only. ➔ only runs once.

- 2) nothing = Runs when component EITHER renders for first time OR after it rerenders. ➔ runs at initial render + after any other rerender.
- 3) [data] = When component EITHER renders for first time OR after (whenever it rerenders AND any of the data inside the array has changed).

- Format: useEffect(functionToRun, dependency_array);
- In practice, it is more common to see the first and third cases, and not really the second case.
- In the first arg, there is a function. This function **can only return one possible value, and that is another function. The goal of this returned function is to do cleanup.**
  - This means that you CANNOT put an async-await function inside here directly for your axios calls (because it returns a response, not a cleanup function). To go about this, you must build a function and then inside that function, call the async function. There are three methods:
    - ❖ Create an async-await function that is tagged to a variable. Then call that function_name().
    - ❖ Instead of naming it, wrap the definition in brackets, and then call it. Smth like **(**async () => {..}**)**();.
    - ❖ Normal promise method with .then().
  - When first render, the first arg function is invoked and we return that inner cleanup function (no call or run yet) to be stored for now.
  - In subsequent rerenders, we will call (actually run) the previously returned cleanup function, and call the overall function again. Store the return.
  - Cleanup function is also invoked when component is removed.
- The next few below are advanced useEffect details.
- Whenever you refer to a prop or piece of state inside of uE(), React's *ESLint* will want you to reference any different prop or piece of state inside the dependency array to solve **stale data references**. The **dependency array** is the second arg, the [data], that controls when uE() gets executed because whenever one of the elements in it changes, it causes uE() to execute. AKA you must put them inside the array, ie [data, prop1, prop2, state1, state2…].
  - Not doing so may lead to weird problems.
  - But including in ALSO leads to other problems! For eg, extra requests are issued because when one state changes, the whole thing rerenders, but those actions cause the other state to be changed as well, leading to another rerender! So there are two rerenders instead of when you need just one.
  - But generally better to follow warnings. **Include it!**
  - To solve this case, we introduce a **debounced term**. Debouncing = "wait until function hasn't been called in X time, and then run it". Basically it is making use of one state to effect another state.
    - ❖ Have two uE()s, one watches *term* that triggers a timer when it changes, while other watches *debouncedTerm* which activates when *term* changes. The axios request is placed inside the *debouncedTerm*'s uE().
    - ❖ Whenever *term* is (immediately) updated, it will trigger a timer that changes *debouncedTerm*. Any updates to *term* will kill the original timer and create a new timer.
    - ❖ Since *debouncedTerm* is updated, uE() will then run to make a new request.

- **useRef** allows us to get a direct reference to a DOM element.
  - We use it to allow us to close a dropdown menu component when we click outside of it. **Why is this challenging to implement?** This is because the Dropdown component can only listen to clicks INSIDE its component – it CANNOT listen to clicks outside its component, ie cannot listen to a click on the DOM body that is not part of it. To fix this, we must learn about **event bubbling**.
  - **Event bubbling** is when an event goes up the DOM tree. It checks each element as it goes up to see if they have an event handler, and if they do, that event handler gets activated. Event continues to bubble up.
    - When you click an element, the browser will create an **event object** that describes that click (eg position of mouse etc.), which the browser passes to React. React does a bit of processing on that event, and returns another event object to the onClick() event handler. This new event object then bubbles up to its parent and so on. At each parent, it will check if there is an onClick() event handler and invoke if there is any. This is called **event bubbling**. To be even more precise, it will NOT activate listeners that are not from the child to parent pathway.
    - Even if parent node has event handler but its children don't, as long as anything from parent all the way to its children get clicked, this bubbling still happens.
    - Almost all events bubble, but some do not.
    - Can stop bubbling up with stopPropagation(). But usually considered bad practice as it will create more bugs around other parts of your code.
    - There is a different version called **event capturing**. Basically a click on any element will cause an event to go from the most top all the way to that element (capturing), and then bubble up from there.
    - You might see unexpected behaviours but that is because **event listeners added manually (normal JS) will always run first, while those in React will run afterwards, and from most child to parent.** Hence, you must be aware on how the various event listeners in your app affect things.
    - To solve this issue, then we have two scenarios: (1) click on component = the manual body listener not activate, and (2) click on body = only body listener activates (which is already done the default). Hence, we need to know what element was clicked (event.target) and if it belonged to the Dropdown component.
    - The latter uses useRef() that gives us a direct reference to a DOM element. Do a "const ref = useRef();" and then assign ref as an attribute to the most parent tag. Refer to it via **ref.current**.
    - Ultimately: if (ref.current.contains(event.target) {}.
    - Example: *useEffect(() => { document.body.addEventListener('click', () => { setOpen(false); }, {capture: true}); }, []);*
    - Remember that if the component is not rendered, then you should remove the listener as well, as ref will be null.

Some new details

- React.Fragment is just a way of making "invisible" nodes. For example, just using may cause an additional border to be rendered when you don't want, so instead can use <R.F>. See this for a good use case.
  - Whenever we create an inner JSX inside of another JSX tag, that inner JSX element is provided to the outer one as a prop called *children*.
  - You can shorten <React.Fragment></React.Fragment> to just <></>. HOWEVER, some tools see that as invalid JSX, so safest to always just write <R.F>.
- **React Router** is the library for implementing **navigation**.
  - For eg, each component of your app will be localhost:3000/component1 etc.
  - However, it always has breaking changes. It is more important to learn the ideas and theory of navigation.
  - You can also build navigation stuff from scratch.
  - **Route mapping** is to route a certain URL to show a certain webpage. For eg, localhost:3000/ shows the Accordion component, while localhost:3000/list shows the Search component.
    - To get the web URL, use *window.location*. It returns an object that describes your URL. Focus on *pathname*. A pathname is everything after the *domain:port*.
    - In your App.js, you can then use a if-else on window.location.pathname to return the respective component. **However, this is not a very React way of solving this**. Rather, we can create a component that will display based on URL path. Its props are the pathname and the children. See the widgets app for Route and App.
    - When creating a navigation bar for your components, every time a user clicks on any tab, it will cause a **full page refresh** (aka call website to send all the CSS JS etc again). This is bad

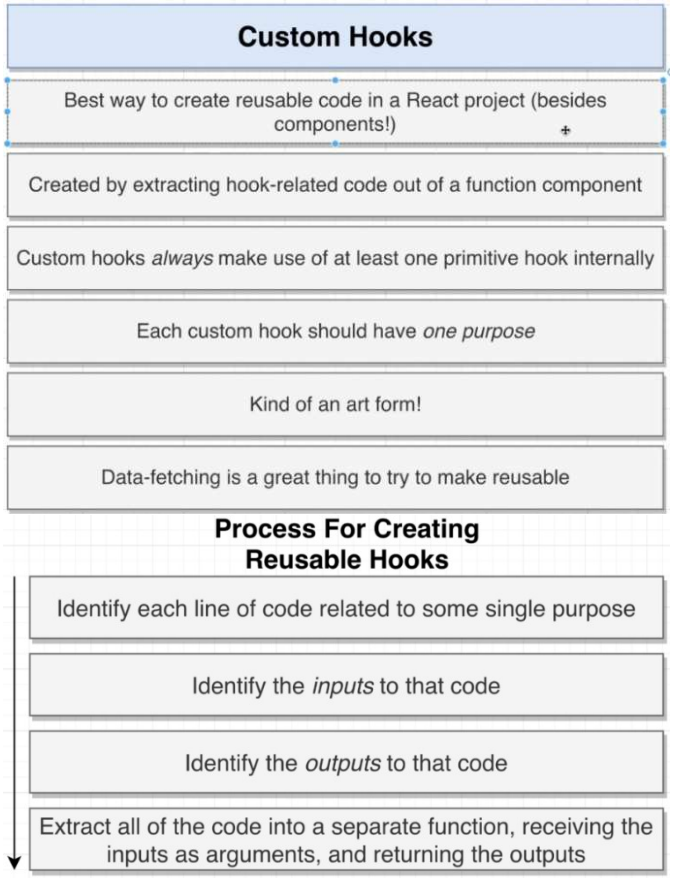
for network traffic. Instead, build it such that if URL changes, each Route can detect the URL has changed and update the state that is tracking the current pathname. Each Route then rerenders, showing and hiding components respectively. We do this by creating a Link component embedded inside the header. When each Link is clicked on, it sends a *navigation event* to all the Routes and compares pathname to see if they display or not.

- **window.history.pushState(obj, str, page)** to edit the URL in the URL bar; if not when you change components the URL will not change also, leading to bad user experience. *obj* is an empty object {}, *str* an empty string, and *page* is the pathname ie '/translate'.
- In the code under Link.js, you might see this "*const navEvent = new PopStateEvent('popstate'); window.dispatchEvent(navEvent);*". Just need to know that it communicates a **navigation event** to the Route components that the URL has changed. The Route components have a listener to popstate events and so will capture these navigation events. What is PopStateEvent? See here.

- If you are passing a prop like: x={(video) => setVideo(video)}, you can change it to x={setVideo}. Identical.

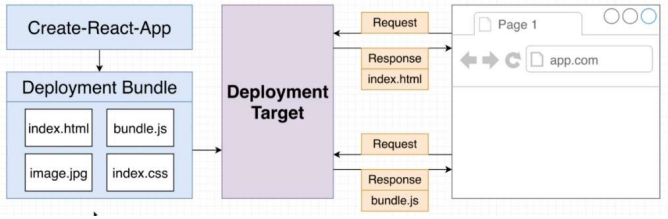
Custom Hooks

- When to make custom hooks? When you have two or more components that use roughly the same logic, it is time to create your own custom hooks. When you want to make JSX reusable, you use Components. But if you want to make functions reusable, you use custom hooks.
- Eg, we will be making video fetching a custom hook, so different components can fetch videos (and be further edited to behave in their own unique ways).
- Another way of thinking about it: in functional components the top part are the functions, while the bottom part is return JSX. Custom hooks are about converting the functions part / non-JSX part into reusable parts that other components can reuse.
- When returning something, the value can either be: (1) like the useState return, returns [x, setX, otherFn] as an array (more React-like), or more commonly (2) {x, setX, otherFn} as an object (more JS-like for the community). Both are fine.



- Note that the above steps are not foolproof; it is an art form, and you must always try and intuit.

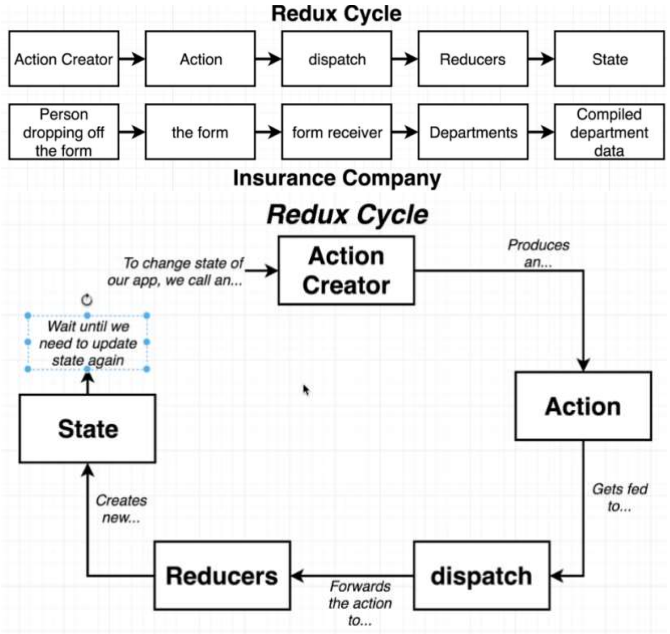
Chapter 7: Deployment



- The files we create are called the **deployment bundle**.
- When coding, we are running in **development mode**, which means we are running a **development server** when building the project. Now, we need to upload these files into the **deployment target**, which is the service to deploy the app.
- For React, we don't need a full virtual machine as we are NOT executing any code.
  - Only need a VM if you are running some kind of active server (a server that is actively running some code). For eg, may need a VM if you are running a NodeJS API in there: the VM will run this server process and the server will respond to incoming requests.
  - For the widgets app, we are merely hosting static files.
- We will use **Vercel**.
  - Deploy: vercel login ➔ vercel ➔ press Enter for all
  - Update: vercel –prod
  - May need to check security permissions: Set-ExecutionPolicy -Scope CurrentUser -ExecutionPolicy *RemoteSigned*
- Can also use **Netlify**. It works by saving your work in GitHub and then from GitHub you deploy.

Chapter 8: Redux

- **Redux** is a state management library. Unlike the previous, we will be extracting the states into this Redux library.
  - React's primary goal is about rendering content and handling user interaction, not about handling data. This is where Redux steps in to **handle data and make complex apps easier**.
  - Not necessary to make React apps.
  - Not explicitly designed for React – see it in other languages as well.



- The **Redux Cycle**:
  - **Action Creator**: function that returns a plain JS object called the Action. Purpose is just to create the Action.
  - **Action**: must have *type* (describes change we want to make to data; usually all caps and underscores for spaces) and optionally

payload (describe context of the change we want to make) properties. Hence its purpose is to describe some **change** we want to make to the data inside of our app, **ie its purpose is to change the state of the data inside the repository**. For eg when click on a button, it will change the state of the selectedSong to the one that was clicked, which said Action will go to (all) reducer that will make the relevant change.

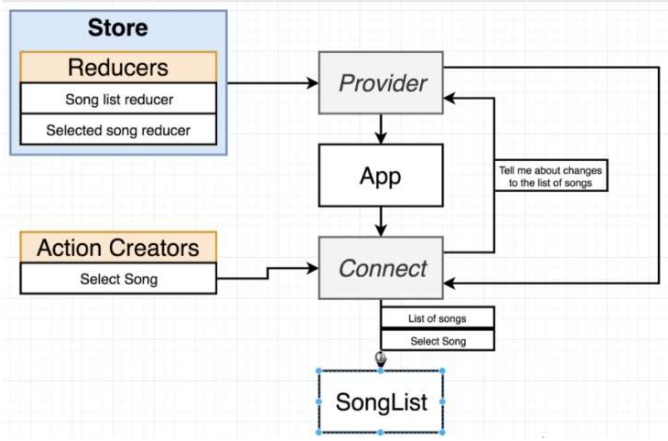
- **dispatch**: takes in an Action, make copies of it, and passes them to different places in our app. *Also passes the store's data that is relevant to each reducer.*
- **Reducers**: function that takes in an Action and some data from the store. It will then process the Action based on its *type*, do nothing, or make some change to the data, and return a brand new value / array / object to the 'central repository'. **All the states are managed here.**
  - Take note that 99% of the times we will NOT append to an existing array, but will create a brand new array with **[...oldList, action.payload]**.
  - To remove things, similarly we don't edit the original array, using **arr.filter(i => i !== 2)**.
  - If a reducer's sole purpose is to send data without changing anything (ie send a static list of songs), there need no Actions for it. It just will not take in any args.
- **State / Store:** the central repository of all data, from all types of reducers. This way, React doesn't have to go to each separate reducer and can just access the state.
  - const { **createStore, combineReducers** } = Redux;
  - const ourDepts = combineReducers({ dept1: dept1, dept2: dept2... });
  - const store = createStore(ourDepts);
  - By doing the above 3 lines, the **store** that you created represents our entire Redux app. It contains references to ALL of our different reducers, and to all of our states / data produced by those reducers as well.
  - cR() helps to wire all the separate reducers together.

- To start things off:
  - Define all your Actions, and Reducers.
  - Do the three lines to create State.
  - Create an Action instance. Optional.
  - Do **store.dispatch(action);** to feed that action to the "form receiver", who will then send a copy to all its reducers inside. Inside the brackets can be any of the other actions you want to do.
  - Also can just do store.dispatch(actionName('arg1', arg2));.
  - To view what is in store, use **store.getState()**. Do it like console.log(store.getState());.

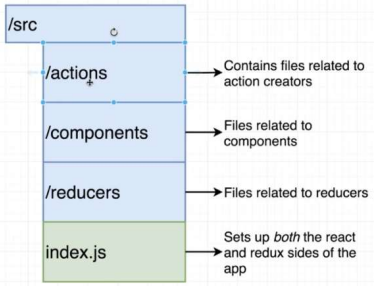
- You can only edit the data by calling dispatch() with any of the actions created by the Action Creator. You cannot modify data with any other methods. This makes it easier in the future because you know you can only use those functions to modify data; things are much more systematic with the few functions available if you wish to modify data.

<u>React-Redux</u>

- React and Redux are their own separate systems. To link them both, we use the **React-Redux**.
- There may be scenarios just to have state in the component itself, but generally keep it in Redux.



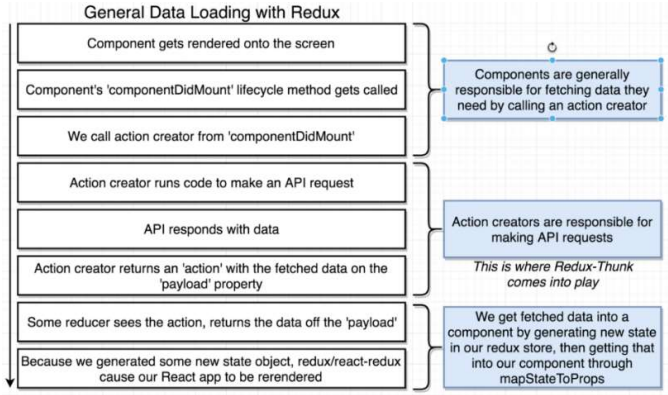
- React-Redux creates two new components: **Provider** and **Connect**. We will pass props into them to configure how they work.
  - The picture shows all the reducers being combined with cR() into a Store. This Store then is passed as a prop to the Provider, giving it permanent reference to the Store. Note that the Provider is rendered even before the App.
  - The Provider <u>provides info to all the different components inside the App</u>. To be more specific, the *Provider* makes the Redux *store* available to any nested components <u>wrapped</u> in the *connect()* function. (So yes the components like SongList are wrapped in Connect()). Applications will render a *Provider* at the top level that will wrap the App component inside it, ie the entire app's component tree is wrapped inside it. Generally, you cannot use a connected component unless it is nested inside of a *Provider.*
  - We also create an instance of Connect, which can communicate with the Provider despite the latter being the top of the hierarchy. It does so not via props, but via the **context system** (allows any parent component *Provider* to communicate directly with any child *Connect* even if there are other components in between like *App*). Connect is required to allow each component to talk to Provider and get the data in the store.
  - Hence, the Connect() not only is to get data out of the store, but also to get actions from action creators to communicate to that store. **AKA if you need to make changes to state or receive data, you will need connect().**
    - Connect's purpose is two-fold: to pass actions to the Provider to talk to the reducers and effect some change, <u>OR</u> to pass data as a prop directly into the SongList component, which then dispatch() will auto call it to effect those changes.
    - Connect also helps to connect the components with the AC. The component will call AC, and AC will be passed as a prop down into the component.
- The idea is: when we want to change the song, we will call the Action Creator that will dispatch an Action and tell the selectedSong Reducer to update its data and reflect the new song. Since there is a change of state, the Connect() will send the updated info as props down into the end components.
  - If a component does not change a data state, no need to wire up any action creators with it.
  - Upon a change in state, all components that are hooked up to Connect() will rerender.
- Your folder should look like this:



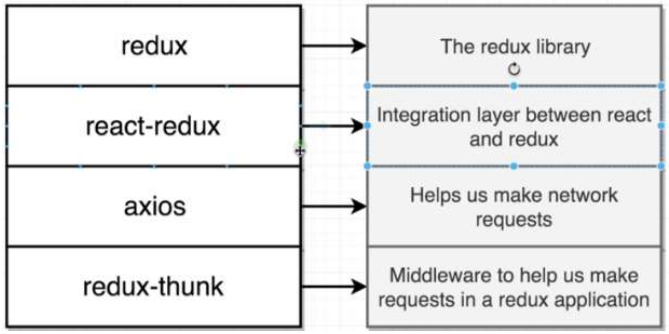
  - Why create new index.js inside folders like actions? It is because we can import them as '../actions' instead of '../actions/index'. Webpack will help to find the index.js if you do not specify the directory.
  - When you have **named exports** (eg export const fnName = ...), you can import them with curly braces {}. But if it is a **default export** (export default fnName;), then don't need braces.
- For imports:
  - Actions do not need any imports. Just define all ACs and As.
    - May want to import any base APIs.
    - May want to install Lodash.
  - Reducers (define all reducers and export a combined version):
    - import {combineReducers} from 'redux';
    - export default cR({ key1: r1, key2: r2... }); where key is the name of the output for each reducer.
  - Main App just import React and the various components.
  - src's main index.js:

- Import React, ReactDOM, and App as per normal.
- import {Provider} from 'react-redux';
- import {createStore} from 'redux';
- import reducers from './reducers'; # your reducers!
- ReactDOM.render(
  <**Provider store={createStore(reducers)}**>
  <App/>
  </**Provider**>,
  document.querySelector('#root')
-);
- **AKA** have a Provider and send a store into it as props. This store was made from the combined reducers. Use it to wrap the App.
- For **Thunk**, see that chapter.

- Components:
  - import React from 'react';
  - import {connect} from 'react-redux';
  - Import the ACs you need.
  - **const mapStateToProps = (state, ownProps) => { return { songs: state.songs }; };**. This is for retrieving (all) the data inside store, and process it, such that it will show up as props inside our component. This means that if you type "this.props" inside the component code, it will be the return output from this function (it also includes the dispatch() function as well). Its name can be anything.
  - **export default connect(mapStateToProps, { key1: ac1 }) (ComponentName);**. Why are there two parentheses?
  - This is because the first function returns another function. The mSTP argument in the first function helps to allow Connect to talk to Provider. If you don't need any input state to use as props, put "null".
  - The {…} object contains all the ACs you want to call, and will pass it as prop into your component to use (call by its key name). **Why need to import, then call here, rather than directly inside the component?** Because Redux is not magic and does not automatically detect ACs being called – calling the AC directly will not tell Redux to send it to all the reducers etc. That only happens when dispatch() is used to send the As to the Rs, and that happens only when sent to connect(). In summary, connect() does a special operation on the functions / ACs inside the object, call the ACs, take the returned As, and then call dispatch() on them.
  - For the second function, we call it with the second set of parentheses with the component as the argument.
- Route:
  - Create all Reducers in their own JS file and export a combined version of them.
  - In src's index.js, create a store from the combined reducers. Pass it as a prop down Provider. Provider will wrap App at the highest level so it can talk to all the components within it.
  - Create all ACs that will return an A (has type and payload).
  - Import the (2) AC you want in a (3) Component.js. Within this component, also create a (1) mapStateToProps that will retrieve all the store data. Make mSTP to return a simple object on the data you actually need. Export with connect() both (1) and (2) and (3). Whatever is inside connect() will be returned as props.
  - Continuing, when AC is triggered, it will return an A to all Reducers. Reducers will only update a value if type matches. Store data is updated and returned back as props. All components that are connected to connect() will rerenders if anything in store is updated.



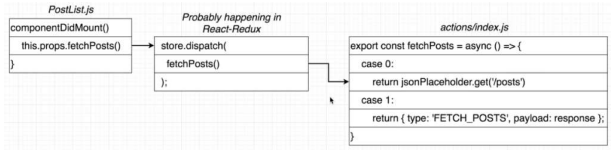
- The above picture is the standard method (always use this) to use when you want to load data into Redux app from an API:
  - Use a class-based component so you have access to cDM(). Each component is responsible for fetching the initial data they need to get. Use that cDM() to call an AC.
  - AC will get and receive data from API. Update store data.
  - Use mSTP() to get data from store into component (via connect()).



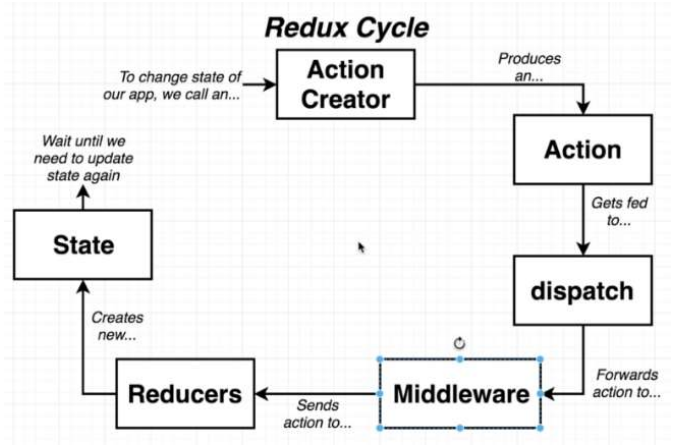
- Dependencies:
  - npm install --save redux react-redux axios redux-thunk
  - react-redux is because react and redux are separate things and need this interface layer to talk between each.
  - Redux Thunk is a middleware that are essentially function to change the behaviour of our Redux store, by adding new features. Specifically for Thunk, it is a middleware that helps us make network requests from the Redux side of the app. There are other alternatives but Redux Thunk is the most popular.

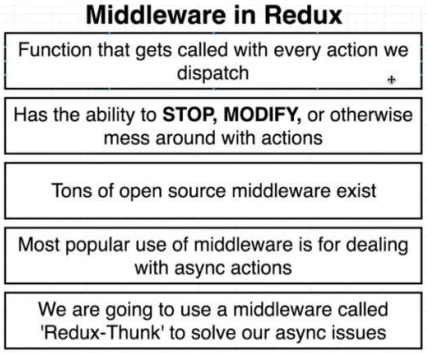
Redux Thunk

- ★ Why need Redux Thunk? Because if you do normal way with axios, you will get an error where "Actions must be plain objects".
  - This means that our Action Creators inside Redux are supposed to return plain JS objects that have a *type* property and optionally *payload*. But at present, we are not returning an Action from our Action Creator! This is because while the returned item LOOKS LIKE an object, but when translated to ES2015 it is not **due to that async-await syntax**! For proof, copy and paste into babeljs.io to see.

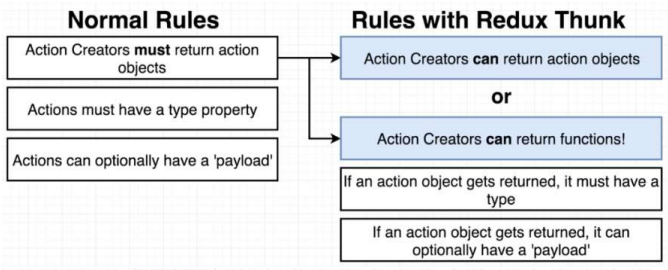


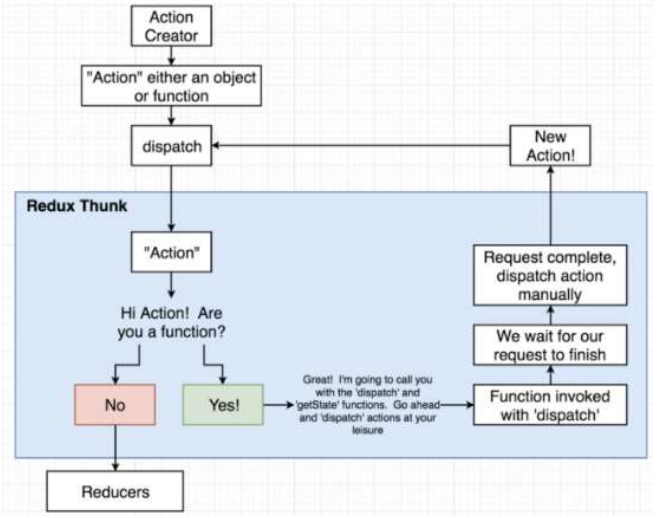
  - From picture, Babel will turn the code with a section of case 0 and case 1. When the AC is dispatched and called for the very first time, it will be case 0, and returns a Request object, which is not a simple JS object; it is not an Action; it has many other methods in it and probably no *type* property.
  - Then, one might ask, why cannot just remove the async-await syntax and just get the Promise and return it in the payload? (Note that without the async-await syntax, we are returned a Promise but it has no data inside yet, like an empty shell. Only after the request is finished after some time, will the Promise give us access to that data). Doing this would remove the error message and we are indeed returning a normal JS object. **So why cannot?** This is because by the time our actions get to a reducer (from AC -> A -> dispatch -> R which is extremely instantaneous like fractions of a second), our data is still fetching (takes a few seconds). The reducers will take the empty promise and do nothing with it since there is no data. There is nothing we can do to make reducers wait for the fetching to complete. Our component will render before the fetch is complete (and will not rerender upon completion!). The store is still not updated. Hence, this method fails.
  - If you want to learn more: see here and also the next video.
  - Just know that use Thunk for async actions.





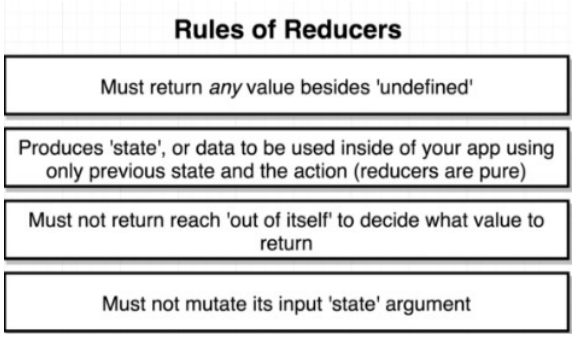
- There are two types of action creators: **synchronous AC** (instantly returns an action with data ready to go) and **asynchronous AC** (takes some time to get data). To allow async ACs to work, you need **middleware**.
  - We call our ACs to return an Action that will be fed to dispatch(), but rather than sending that A off directly to Rs, it will be sent to all the different middlewares (can be one or multiple) inside the app.
  - A **middleware** is a plain JS function that is going to be called with every single A that you dispatch. Inside that function, a middleware can stop journey to the Rs / modify the A / change the A. An example is a middleware that console.log() every dispatched action.
  - There are many open-source middleware, but you can also write your own. Redux Thunk is a popular one to solve async issues.



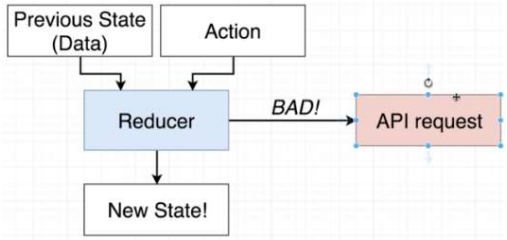


- Thunk has a lot of uses – handling async requests is just one.
- Redux Thunk is an all-purpose middleware that with React, either returns an object (with type and maybe payload), or returns a function. If it returns the latter, it will automatically run that function for you. That is all it does.
  - Hence underwhelming because all it does is return a function from an AC, and call that function.
- So how does this help us? Look at second picture of how Redux works internally:
  - Assume this is our AC (example is incomplete – for learning):
    - export const fetchPosts = () => {
      return function() {
      const promise = jsonPlaceholder.get('/posts');
      return { type: 'FETCH_POSTS', payload: promise };
      }};
  - AC will create either an object or function and pass to dispatch(), which will then pass it to Thunk (middleware – see Redux Cycle pic). If it was an object, pass to reducer and end.
  - But if it returns a function, Thunk will invoke that function with input arguments dispatch() [to update states] and getState() [to get all states in the store]. Like this:
    - function(**dispatch, getState**) {
    - const promise = jsonPlaceholder.get('/posts');
    - return { type: 'FETCH_POSTS', payload: promise };
    - }};
  - Thunk then invokes that function with those two arguments.
  - This means the async function is being run and we will wait for that request to finish. When the request is completed, then we can some time in future **manually** dispatch that action. When we dispatch an action, we get a new action (as a JS object) which will go through the whole process again (but this time directly to R) until it goes to the R and hence the store.
- Can see source code (only like 16 lines!) at gh.com/reduxjs/redux-thunk.
- Imports:
  - src's index.js:
    - import {createStore, applyMiddleware} from 'redux';
    - import thunk from 'redux-thunk';
    - aMw is for connecting Thunk to Redux.
    - <Provider store={createStore(reducers, **applyMiddleware(thunk)**)}>
- Other changes:
  - For ACs that return a function, its inner function doesn't need to return an action. Instead, we can dispatch the action right in there. This relates to the part about manually dispatching it so.
  - fetchPosts() example:
    - export const fetchPosts = () => async dispatch => {
      const response = jsonPlaceholder.get('/posts');
      dispatch({type: 'FETCH_POSTS', payload: response});
      };
  - Why got two arrow functions? The first one is the outer function, the second one is the inner function. Due to ES2015, and as outer function is just returning one expression that we are immediately returning, can do so. Still same as:
    - export const fetchPosts = () => {
      return async dispatch => {
      const response = jsonPlaceholder.get('/posts');
      dispatch({type: 'FETCH_POSTS', payload: response});
      };
      };
    - Confused? It is the same as onClick={() => console.log(123)};.
  - Why suddenly can use async-await? That is because we don't really care what the inner function returns – Redux or Thunk will not even look at it. So despite the inner function returning an async-await with the case 0 things, it will never be used. It is only what is returned from the outer function (ie is it an object or a function?) that Redux cares about. After which, Thunk runs that inner function. The manual dispatch then brings the simple JS object to all the Rs for processing.
  - Why no getState as arg? If no need to use, don't need.

Role of Reducers



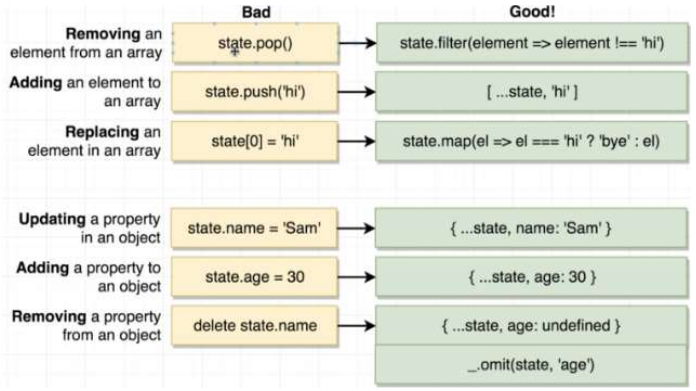
- Some notes:
  - Rs MUST always return something, if not got error.
  - At start, Redux will run all Rs once.
  - When Rs first boot up, it will receive two arguments – undefined and some A. Assume this R:
    - const abcReducer = (selectedSong = null, action) => {
      if (action.type === 'SONG_SELECTED') {
      return action.payload; }
      return selectedSong; };
    - Then Redux will run abcReducer(undefined, {type: 'xblahx'}). (We don't care what *type* that A has, it is just a random A.)
    - The portion "selectedSong = null" is ES2015 for "if selectedSong === undefined, selectedSong = null".
    - Hence when the first time Redux runs the R with undefined argument, we default its value with null. This only happens once because after that, the value will not be undefined anymore.
    - After all the above, it returns the value to be stored as a state.
  - In subsequent runs, it would not be "undefined" that is called as the first arg, but rather whatever the state-value that was returned previously. AKA abcReducer(prevReturnedValue, A). Note how then there is a cyclical action where R's first argument will always be the previously returned state-value. The only real change is the Action object as second arg, with different types and payloads.
  - Rs not supposed to reach out of itself. For an anti-example:



    - Cannot call API, read some file in hard drive, solicit some user input, or reach into DOM and get smth.
    - An R should only look at the previous state value (the data) and the Action, to see what to return. It should only look at some computation done on the state and the A.
    - Hence it is **pure** because it only returns values that use only its input arguments.
  - The last rule is possibly misleading and maybe wrong.
    - "**Mutate**" in JS means to change the contents. For eg, pushing an element into an array, removing one, or changing one to another.
    - Basically it is saying we should not change the old state that was taken as input arg.
    - Why is rule wrong? Because actually you CAN mutate it and there will be no errors. It is just easier to tell beginners not to do so, because there is an edge case where it DOES lead to an error. In the Redux source code for combinedReducers():

```
let hasChanged = false
const nextState = {}
for (let i = 0; i < finalReducerKeys.length; i++) {
  const key = finalReducerKeys[i]
  const reducer = finalReducers[key]
  const previousStateForKey = state[key]
  const nextStateForKey = reducer(previousStateForKey, action)
  if (typeof nextStateForKey === 'undefined') {
    const errorMessage = getUndefinedStateErrorMessage(key, action)
    throw new Error(errorMessage)
  }
  nextState[key] = nextStateForKey
  hasChanged = hasChanged || nextStateForKey !== previousStateForKey
}
return hasChanged ? nextState : state
```

    - Basically **previousStateForKey** is the state data from the store while **nextStateForKey** is the return value of the R. At line 174, **hasChanged** will compare those two if they are the exact same array in the same memory address. So if those return the exact same array in memory, hasChanged will be false; if different memory then hasChanged is true. Hence, if any R changed a state, hasChanged will be true. Ultimately, if hasChanged is true it returns the new entire state object (the store) that has been changed by all the Rs; if false then it returns the old store. **HENCE**, if it returns the old store, then Redux will not notify that any data has changed, so no rerender; if returned new store, all apps get rerendered.
    - IN SUMMARY, we avoid mutating old state because if you return that old state, your components will not update.
    - Best is just don't mutate state. Use below. Note that _. uses the **lodash library**, a popular way to work with objects and arrays.
      - ❖ _.omit(state, thingtodelete) does not execute on the original state object, but rather creates a new object without the thingtodelete k-v pair.



- Can also use mapStateToProps as the place where you do calculations. For eg find a single user. It is the place where you want to do some precalculations on props or Redux states.
  - This is better because it increases reusability.
  - Assume the component is kept in a file, while the mSTP / connect parts is kept in another file. Then, one can use that component on its own if he doesn't need it any states.
  - If you need to reference the component's props (which you might need to get the userId that was sent to it as props), use the second argument **ownProps**. It is a replica of the props that the component is given:
    - const mapStateToProps = (state, **ownProps**) => {
      return {users: state.users.find(user => user.id === ownProps.userId)};
      };

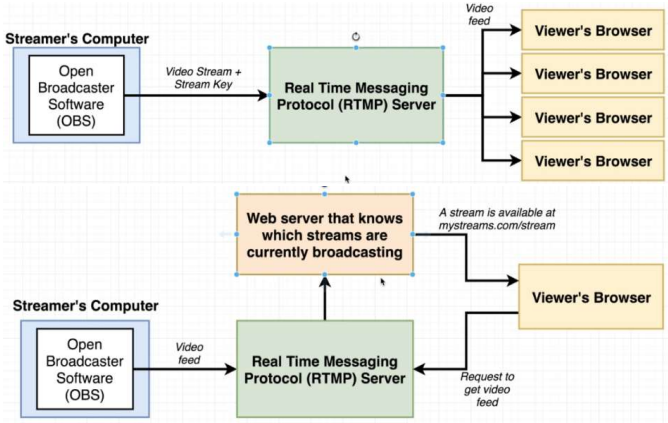
Memoizing Functions

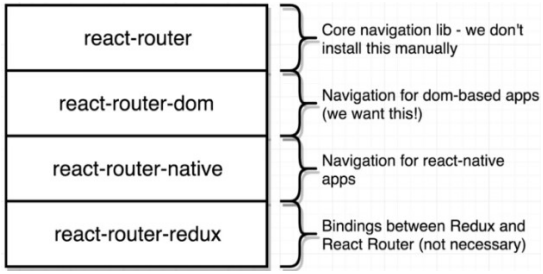
- You may face a problem of making too many requests for a simple thing. There are two methods to solve this.
- First is using Lodash (need to be at website for Lodash to be on Window Scope).
  - Imagine you have a function:
    - function getUser(id) { fetch(id); return 'Hi!' };
  - Then you memoize it:
    - const memoizedGetUser = **_.memoize**(getUser)
    - The returned function will be wrapped up with additional code. It will have the same behaviour as the old getUser function, but the only difference is you can only call memoizedGetUser() **one** time with any **unique** set of arguments. After calling that one time you can still call it, but it will not be invoked, and instead return whatever that was returned previously.
  - npm install --save lodash
    - import _ from 'lodash';
  - Now let's test it out. Assume the function is this:

- export const fetchUser = function(id) {
  return async function(dispatch) {
  const response = await jsonPlaceholder(`/users/${id}`);
  dispatch({type: 'FETCH_USER', payload: response.data});
  }
  };
- Do you then memoize the outer or inner function?
  - If you do the outer, then the returned value is the inner function. By returning a function, Thunk will help invoke that inner function, which will call the API every single time. So memoizing here is of no use.
  - However, if you memoize the inner function, it also won't solve the issue. This is because every time the AC fetchUser is called, it will return a new inner function in memory to memoize; we are memoizing a new version of the function every time we call the AC, so all the requests will still be run.
  - How about memoize both? Seems to work actually!
  - export const fetchUser = _.memoize(function(id) {
    return _.memoize(async function(dispatch) {
    const response = await jsonPlaceholder.get('/users/'+ id);
    dispatch({type: "FETCH_USER", payload:response.data});
    });});
- To solve the issue, can define a function outside the AC that will make the request and dispatch our A. Memoize outside the AC so that it only gets memoized once.
  - Functions with a _ at the front means private function.
  - export const fetchUser = (id) => dispatch => {
    _fetchUser(id, dispatch);
    };
  - const _fetchUser = _.memoize(async (id, dispatch) => {
    const response = await jsonPlaceholder(`/users/${id}`);
    dispatch({type: 'FETCH_USER', payload: response.data});
    });
- The drawback of memoize is that if the user data has changed on the API, it will not be updated as there will be no more get requests to it. You can solve it by creating a new AC that is not memoized to call the API.
- memoize is actually a very small piece of code! See here.

- Second method is to wrap the sub-ACs with a larger AC.
  - Larger AC will call the sub-AC that fetches the list of posts. Then from there find all unique userIds. Iterate over them and call sub-AC that fetches for each userId.
  - If there are async-awaits in there, you must async-await the larger function as well.
  - You must manually dispatch all the sub-ACs (those w/ async-awaits). Hence when you dispatch(fetchPosts()):
    - export const fetchPosts = () => async dispatch => {
      const response = await jsonPlaceholder.get('/posts');
      dispatch({type: 'FETCH_POSTS', payload: response.data});};
    - You are dispatching the brown and red part (the return value).
    - With Thunk when you dispatch a function, it will invoke it and pass dispatch() as the first argument. Hence it will get the posts and dispatch the A, which will update Rs.
    - Hence, whenever you call an AC inside of another AC, need to dispatch the result of the sub-AC.
  - Assume fPAU() is this right now, then:
    - export const fetchPostsAndUsers = () => async dispatch => {
      await dispatch(fetchPosts()); };
    - Same as:
      export const fetchPostsAndUsers = () =>
      async (dispatch1) => {
      await dispatch1(async (dispatch2) => {
      const x = API; dispatch2({…}) }); };
    - 1) Call AC fetchPostsAndUsers().
    - 2) The result of main AC (blue part onwards) will be dispatched automatically.
    - 3) Thunk receives it. Check if blue part is a function. It is, so runs the function (runs green).
    - 4) Runs the green dispatch1. Dispatch the code within green to Thunk, ie the brown-red parts (which is the sub-AC fetchPosts()).
    - 5) Thunk receives it. Check if brown-red is a function. It is, so runs the function (runs red).
    - 6) Call API. Store response. Manually dispatch result, which is an A, a normal JS object. Update stores. Red ends. Brown ends.
    - 7) Green ends. Blue ends.
    - **SUMMARY:** Every time see dispatch(), send to Thunk and check if object or function. If latter, run it.
    - **Why need dispatch1() when got dispatch2()?** Without dispatch1, you will only call brown just to receive a function definition of red (red will not be run).
  - Use getState() to get the updated store data. Then use Lodash's map function.
    - **_.map**(getState().posts, 'userId); // goes to all posts and just pull off the userId portion. Returns array of those.
    - **_.uniq**(the_above); // returns array of all unique values.
    - **_.chain(**getState().posts).map('userId').uniq().forEach(id => …)**.value()**; // chain a bunch of functions. Note that the results of each function will be passed to the next as a compulsory argument, so inside the () is more for the second and so on arguments. Need to use .value() to actually run it.
  - Save those unique values in an array, and use forEach() to iterate through them and call their respective API.
    - No need to "await" for the dispatch() of individual ids. This is because we have no other code below.
    - Async-await syntax do not work in forEach() anyway.
    - But assume you need, you can work with Promises. Something like "await Promise.all(userIds.map(id => dispatch(fetchUser(id)))).then()…".

Chapter 9: React Router



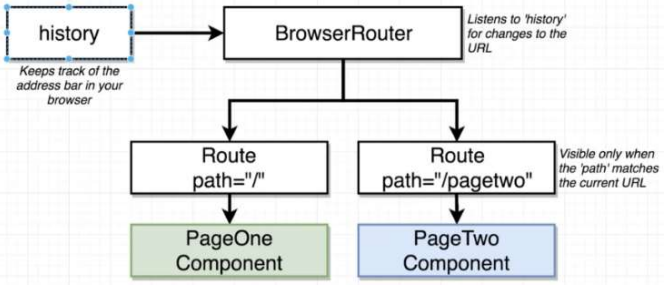
- Components:
  - OBS streams your desktop, possibly to some outside server. Not limited to one OBS – can have multiple OBS (multiple streamers) sending to that one RTMP server.
  - RTMP is a specialised server that receives an incoming video feed and broadcasts to many viewers' browsers. Its sole purpose is to stream videos. Need that separate API server that stores a list of all the streams / channels that are available inside our application.
  - Path: User will from their browser talk to the web server to find available streams. He clicks one, which makes a request to the RTMP server to get live server feed, which then displays on screen.
- **React Router** helps in navigation.
  - 

    - R-R is core library. Included in others.
    - R-R-D is for web apps.
    - R-R-N is for mobile apps.

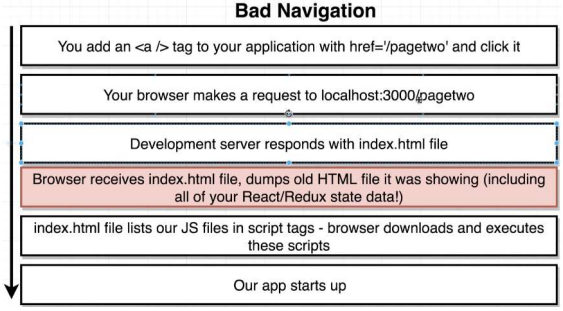
- R-R-R allows computability btw Redux and RR. No one usually uses this; even RR's authors say don't use.
- npm install --save react-router-dom
- import {**BrowserRouter**, **Route, Link**} from 'react-router-dom'; // for main App
- A quick example:

```
const App = () => {
    return (
      <div>
        <BrowserRouter>
          <div>
            <Route path="/" exact component={PageOne}/>
            <Route path="/pagetwo" component={PageTwo}/>
          </div>
        </BrowserRouter>
      </div>
    );
  };
```

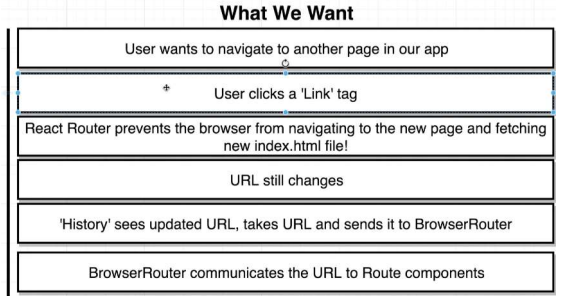
- BR and R are components.



- Some notes:
  - RR only cares about the address after domain:port, ie localhost:3000**/pagetwo** or twitter.com**/abc**.
  - When app is created and loaded in browser, we created a **BrowserRouter** instance, which the BR component internally creates an object of its own called **history**. This *history* object keeps track of the address in the address bar; it will look at the current address bar and extracts just the **path** (the part after domain:port). It passes this path to the BR, which the BR then communicates the *history* object down as a prop to the components. The components will then decide to be visible or not. Any change in address the *history* object will inform BR which will inform the components.
  - ACTUALLY, the *history* object not only tracks the address bar, it can change it too. This is important for programmatic navigation. See later chapter.
- The path-matching process is not foolproof and has some gotchas.
  - In <Route path="/" **exact** component={abc}/>, it means show *abc* only if URL path matches the prop path.
  - Different Routes can be matched by the same URL.
    - The ***exact*** keyword then tells if to match exactly for that path or not. If not, because the path-matching process uses sort of like a "contains" method, then even component with '/page' will show up when current URL is '/page/5'. **All components with path prop being a partial of the current URL gets shown.**
    - For better control, you can use *exact* everytime. Providing *exact* only actually means "exact={true}". This rule is same for other props without an assignment.
- Navigation with RR and traditional HTML <a>:
  - Do not use the traditional HTML way of linking documents, eg Click.

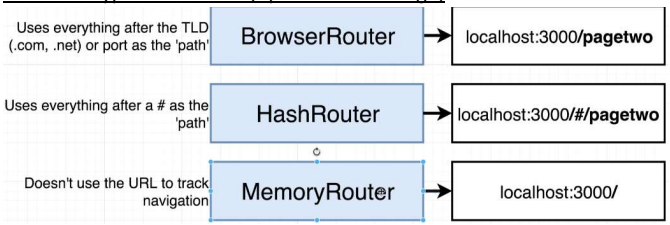


  - As per second box, making a request receives a new index.html file, even if it's just navigation. As per fourth box, it will dump the old one including all the JS files and data. In summary, traditional HTML means getting a brand new HTML and dumping all variables and memory.

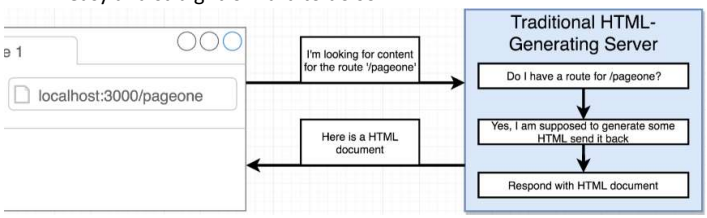


  - To fix this, replace the <a> tags with **<Link>**.
    - Instead of *href*, use *to* instead.
    - Underlying it, you're still showing an anchor <a> tag. The key is that with Link, RR prevents the browser from navigating away and fetching any new index.html files. The URL still changes though, which will be extracted by *history*, and sent to BR, then sent to the Route components to compare.
    - This methodology of not making an additional request for a separate HTML document when we click on a link is called **single page app (SPA)**. It means just loading up a single HTML document that allows users to navigate around, but they are still making use of the same HTML document and showing/hiding elements based on URL.
    - **<Link to="/path" className="xxx">Click Here</Link>**

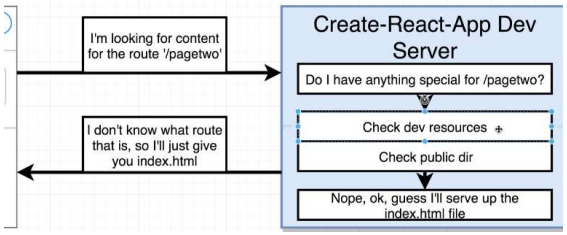
Different Types of Routers (optional knowledge)



- The only difference is they have different methods in evaluating the URL for choosing which components to display.
- BR: Uses everything after the **top level domain TLD** (.com / .net) or port (:3000).
- HR: RR automatically puts a little # character into the URL, and will reflect the current path after that #.
  - import {**HashRouter**, Route, Link} from 'react-router-dom';
  - Replace all BR with HR.
  - When you go to localhost:3000, it will become localhost:3000/#/ instead.
  - Any time you navigate to another page, it will only edit the part after the #/.
- MR: Ignores URL.
  - When you navigate, the URL stays the same.
- **Why the difference?** It is because it comes down to deployment.
  - BR is going to be very easy because a lot of deployment services expect that you will use something like BR, so they will make it easy and straightforward to do so.



  - For a traditional HTML webpage (not React), the above picture happens. You ping the server, which will ask if it has the route it needs to return. If yes, return the HTML page. If not, return 404.
  - Doing with React, we are actually launching via the React Development Server RDS that is running on our computer (your npm start) and we make requests to localhost:3000.



- When you pass a prop of page='/pagetwo' in Route, a traditional server would have returned 404 because it wouldn't know what to do with it. But RDS would do the above process in picture. It would check **dev resources** (click browser Network and hover over the items for the address) or the **public dir** (in VSCode the *public* folder accessible via localhost:3000/manifest.json). But it won't be able to find '/pagetwo' there, **so RDS serves the index.html file**, which contains a line to serve the bundle.js file which contains all the app code. The app then loads and RRouter loads up too, the history object created by RRouter inspects the current URL, sees that we are at URL/pagetwo, the history obj then tells BrowserRouter that we are at /pagetwo, then the BR tells the Route components also, so they will render as appropriate.
- Hence the key difference is that your server is configured like the RDS, so that it can work with RRouter (cannot find page = return index.html). Traditional servers will not work (cannot find page = 404 error).

- **How about HR?** Similar concept, just configure backend server to not take a look at anything after the #. By doing so, the server doesn't care what is after the #, and so will only look at localhost:3000 and serve that index.html only. Hence, it is more flexible because does not require any special configuration to the backend server; can just have single html file and a single route, and always request to that route. When app loads up, RRouter only looks at after # to decide what to show on screen.
- Real world projects (comment on Udemy):
  - MemoryRouter uses no visible path. Useful in cases where direct navigation via the address bar doesn't make sense. Consider for example a game. Sure, the game has a start page, maybe some settings, a highscore list, and various stages going through the game. But navigating directly to e.g. /game/lvl/5 is (probably) not something you want (depending on the game of course).
  - HashRouter has a visible path, but uses the hash thing. Practically for the user this is no different from the BrowserRouter, but using the hash also means you as a developer, when you deploy, can deploy anywhere with no configuration of the web server. It will Just Work™.
  - BrowserRouter also has a visible path, but to properly work, you will need to do special configuration on the web server to make sure that your index.html file is served, even when the web server is asked for something else (like /pagetwo, from the example, which doesn't actually exist on the web server). **So why use it?** Based on what I have seen BrowserRouter is most common to work with if you have access to the server to configure it. Most dev's hate the # used in HashRouter so regardless of ease of use they avoid it. I believe it's the only thing that works with GitHub pages though, so there is a very specific but important use case. >> I'm a developer. And yes, I hate hash routing. Ugly for sure, but also because the hash already has a different function in browsers already. HashRouting is basically a hack, a misuse of functionality meant for something else.

- **BR and HR both are returned index.html. What's diff?**

The big difference is that BrowserRouter uses the history API

A `<Router>` that uses the HTML5 history API (`pushState` , `replaceState` and the `popstate` event) to keep your UI in sync with the URL

eg.

```
<BrowserRouter basename="/calendar"/>
<Link to="/today"/> // renders <a href="/calendar/today">
```

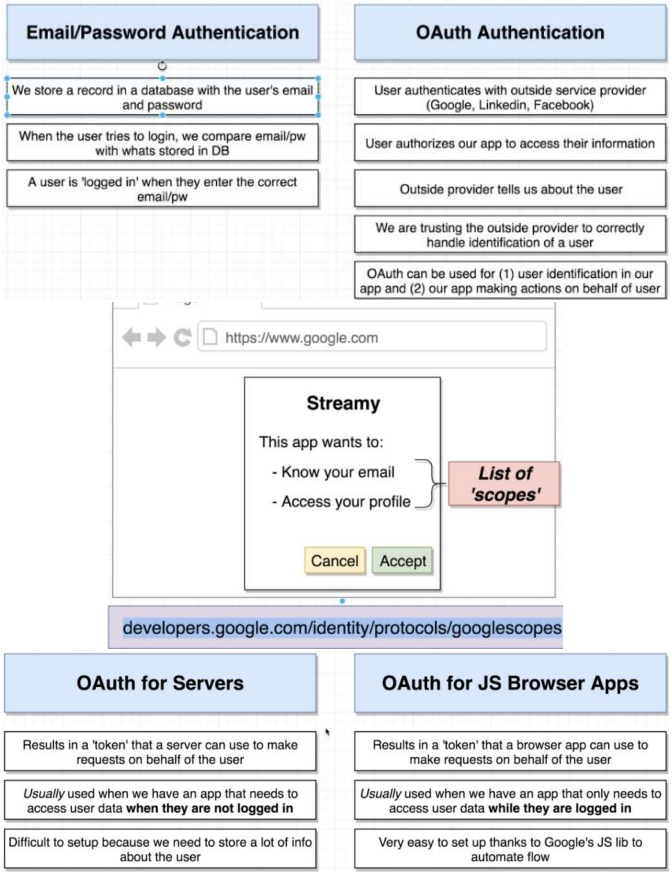
and HashRouter uses the #

A `<Router>` that uses the hash portion of the URL (i.e. `window.location.hash`) to keep your UI in sync with the URL.

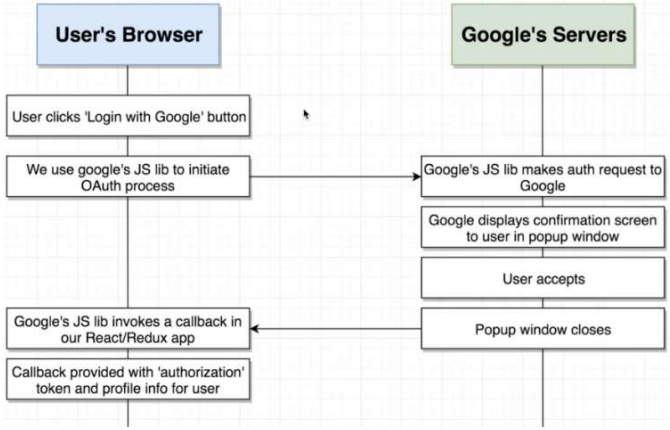
eg.

```
<HashRouter basename="/calendar"/>
<Link to="/today"/> // renders <a href="#/calendar/today">
```

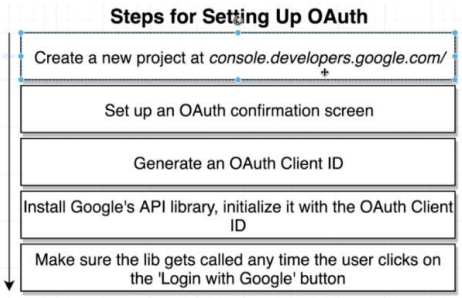
Chapter 10: Authentication (Google OAuth 2.0)



- We will be using **OAuth for JS Browser Apps**. It is for enabling authentication of users who are making use of a JS app running inside a browser.
  - OAuth for Servers is used when building an app that needs to access users' data when they are not actively logged in to your app or actively making use of it. For eg, an app that is going to attempt to access the user's email every 10 mins and delete emails that are flagged as spam. You will need to access their email account every 10 mins regardless if they were logged in or not.
  - Since we are just dealing with user when they are actively logging into our app and using it, use the second one.
  - The token allows us to take some action on behalf of user, without their direct involvement.



- Process Flow:
  - (UB) User clicks our login button to log into Google, this button has a click event handler that will call Google's JS library and initiate the OAuth process.
  - (GS) JS library makes auth request to Google. Sends a pop up window to confirm. User confirms.
  - (UB) The JS library invokes a callback in our React-Redux app. Callback is invoked with some authorisation token and profile info of user. This proves the user is him.
  - (GS – not shown in picture) If user logs out, another callback is invoked. So must wire up to listen for that as well.



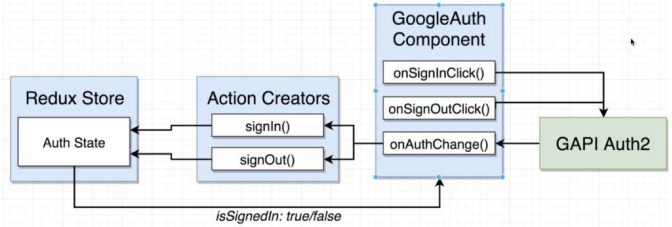
- Steps:
  - OAuth Consent Screen > External > CREATE > App name
  - Credentials > CREATE > OAuth client id > Web application > Allow only http://localhost:3000. Copy the Client ID. The secret id is only for OAuth for Servers.
  - Google never offer it via npm. Instead, go index.html and add this **<script src="https://apis.google.com/js/api.js"></script>**. If you type "**gapi**" in console, should return smth.
  - Create a GoogleAuth component and put it inside the other component where you would like that button to appear.
  - Since tons of websites use the gapi library, Google wants to keep it small, so when you first load it is very bareboned. It has a **load** function, which helps to load up some internal library by making a follow up request to Google to add extra functionalities.
  - Hence: "**gapi.load('client:auth2')**".
  - Then initialise it: "gapi.client.init({ clientId: 'ur_client_id' })".
  - Login: gapi.auth2.getAuthInstance().signIn()
  - Logout: gapi.auth2.getAuthInstance().signOut()
- Actual code:

```
state = {isSignedIn: null};
componentDidMount() {
    // the second arg is because the process will take some time, so the callback is the fn to run after it is complete
    // scope is ask user what different parts of his profile he'll be providing to us
    window.gapi.load('client:auth2',
      () => {
        window.gapi.client.init({
          clientId: '621755719957-fpah4f6q6t0kbal30t3i8frdk7ce1eq6.apps.googleusercontent.com',
          scope: 'email'
        }).then(() => {
          this.auth = window.gapi.auth2.getAuthInstance(); // get the auth instance to do things with it.
          this.setState({isSignedIn: this.auth.isSignedIn.get()}); // because cDM() only executes after first render, so need setState() to rerender and update screen.
        });
      }
    );
};
```

- In JS, since there are no classes, they use **prototypes**. If you see gapi.auth2.getAuthInstance().isSignedIn, you realise that there is no get() function – it is stored in the attribute __proto__ instead.
  - The above cDM() code actually only runs once at start. The setState() will cause a rerender just once (a total of two renders). The UI you see will then <u>never</u> change anymore even if you sign in or out. To fix this, we need to do the below.
  - Add a listen(), which is a method we can pass a callback function to. Doing so will invoke the function any time the user's authentication status changes. This will allow us to update the text inside the Header any time the user signs in or out without needing to refresh the page (will auto rerender). I have a feeling it adds a listener to be more precise.

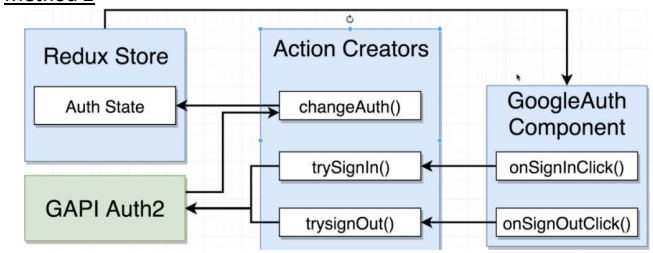
<u>Redux and Google Auth</u>

<u>Method 1</u>



- Not a proper full Redux architecture. Done so as to put all the GoogleAuth in one component for learning purposes.
  - Every time when onAuthChange() is called (person successfully logins or logout), call AC signIn() or signOut().
  - Redux will dispatch the A into the Rs. Update store (which houses the Auth State).
  - Why use Redux? Because there will be many components that work differently if user is signed in or out. A central store will be more efficient.
  - Must send the store as a prop to GoogleAuth as a single source of truth on whether user is signed in or not, to display what kind of button.

<u>Method 2</u>



- Follows Redux more closely, but has authentication logic all over the place.
  - In Redux, only the ACs manage the state. Yet in Method 1 we have the GA Component ultimately doing it by managing the GAPI Auth2 (green box) which is kind of a state.
  - Method 1's attempt to sign in or out happens only at onSignInOutClick() at the component level, but Method 2's attempt to sign in-out only happens at the AC level. This follows Redux rules more closely.

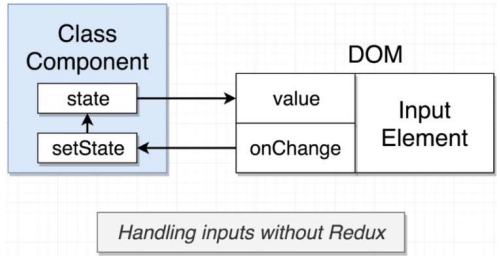
<u>Chapter 11: Redux Devtools Extension</u>

- An extension that you can put in Chrome or Firefox to tell you what is in your store. This makes debugging easier.
  - https://github.com/zalmoxisus/redux-devtools-extension
  - Import *applyMiddleware* and *compose* from 'redux'.
  - const composeEnhancers = window.__REDUX_DEVTOOLS_EXTENSION_COMPOSE__ || compose;
  - const store = createStore(reducers, composeEnhancers(applyMiddleware(thunk)));
- There is even a debugger mode:
  - localhost:3000**?debug_session=<some_string>**
  - The string can be anything. You can even type feawfilbwefbw.
  - This tells Redux Dev Tools think you are starting a debug session, so it will save all your data inside of your Redux Store and is persistent across all refreshes (by default when we refresh the store's data will fall away as well).

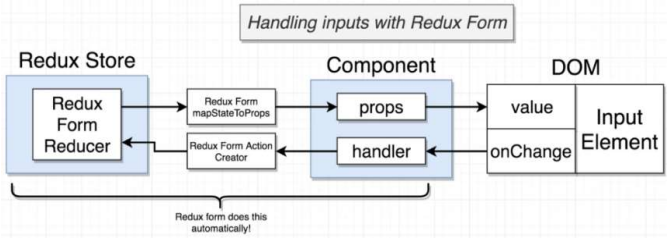
<u>Chapter 12: Redux Form</u>

NOTE: Redux Form is deprecated. Use React Final Form instead!

<u>Original Method (what we did previously)</u>



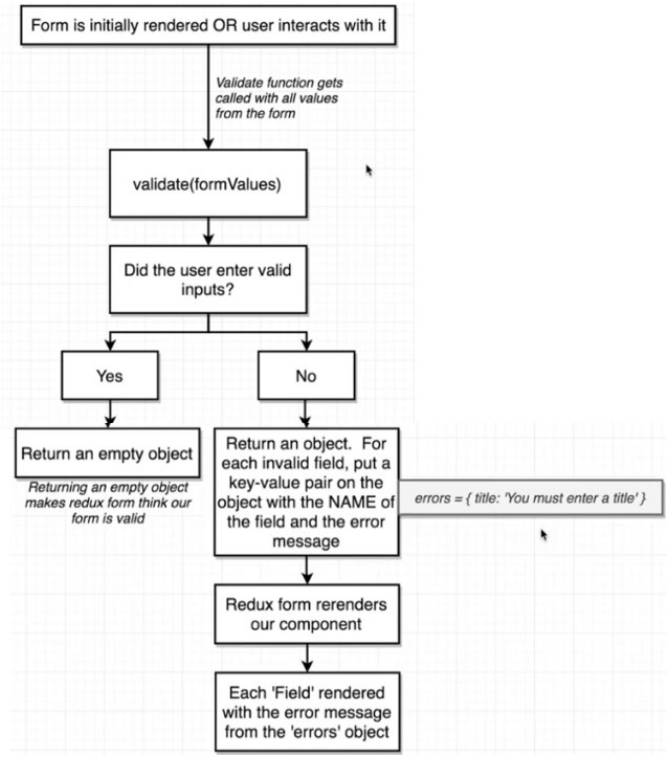
<u>New Method</u>



- final-form.org
- In Redux, we want all data to be in store instead:
  - No more data in the components.

- Any time a user changes an element, use a callback handler to call an AC to change data.
- R on the left will hold all the state of our different forms inside our app. All form data exists in the Redux Store, maintained by a R.
- Send data back as props.

- The difference here is that Redux Forms handles all the above automatically.
  - Don't have to write the Rs as they are written for us. We just need to wire them to our store.
  - No need to write mSTP(). Will be handled.
  - No need to write ACs. Will be handled.
  - Only thing to do is to ensure the props feed the DOM, and any onChange calls the handler.
- Steps:
  - RForm already created Rs for us. So no need to create. But what we need to do is to wire it to our store.
  - In your reducers index.js:
    - **import {reducer as formReducer} from 'redux-form';**
    - export default combineReducers({ …, **form: formReducer** }); // **MUST** assign to the key **"form"**.
  - In your component:
    - import {Field, reduxForm} from 'redux-form';
      - Field is a component. Something like input field.
      - reduxForm() is a function that sort of has the same functionality as connect(). It will be the one to automatically call the Redux Form AC and mSTP() to your component.
      - rF() will receive a single object as argument to configure. The key is "form" and the value can be any named string *(what does the value do?)*.
    - export default reduxForm({ form: 'streamCreate' })(ComponentName);
    - Doing the above will give your component many props to use.
  - When you want to use <Field>:
    - **<Field name="title" component={this.renderInput} label="Enter Name"/>**
    - You may experience errors. This is because Field does not know how to actually render input elements. It is merely a component to handle the RForm system.
    - To fix that, need to give it a *component* prop. The value is either a component, or a function to call. It must return some element to show on screen.
    - It will receive a **formProps** argument which contains all the props that were passed with the <Field>, and also a *meta* property on *error* and *touched*.
    - See streams1's StreamCreate.js.
    - The renderInput() function is then:
    - renderInput(formProps) {
      return <input onChange = {formProps.input.onChange} value = {formProps.input.value}/>; };
    - Or can shorten with: **return <input {...formProps.input}/>**. This will pass all the .input properties as props.
    - In summary, the key is that RForm does not know how to show input elements. Its main goal is to handle all the ACs and Rs automatically, to cut down on the repetitive tasks when doing state management with Redux.
    - If you need to customise your Field, just add props to it. They will be automatically passed down into the component, because Field otherwise would not know what to do so this is its default behaviour.
- For **form submission**:
  - <form **onSubmit = {this.props.handleSubmit(this.onSubmit)}**>
  - The handleSubmit() is RForm's callback function to handle submits. It changes how things are done.
  - Internally when form is submitted, the handleSubmit() will handle the event object for us and automatically call preventDefault().
  - You pass our own onSubmit() function into it. Unlike previously, it will not receive an event object, but rather it will receive all the values that were input into the form. **This is where you customise the journey of the data** and send it to APIs.
- For **form validation**:
  - **Validation occurs every single time user interacts with the form**, even if it just clicking on a box. It will call a function called **validate(*formValues*)** that takes in an argument of all form values, which we must define validate(). *formValues* is the exact same object taken as input when we do our custom onSubmit().
  - Validation: <Field>'s name=x property VS validate()'s error.x property; if they have the same names, compare accordingly and pass error message as prop to the Field's component prop if fail validation. It will show up in renderInput() because it's the component as **formProps.meta.error.**
  - If all validation passed, return an empty object, which indicates to RForm that form is valid.
  - If not, return an object with k-v pair of the wrong field.
  - Define validate() and wire it up as a k-v pair in reduxForm().
    - const **validate** = (formValues) => {
    - const errors = {};
    - if (!formValues.title) { errors.title = 'You must enter a title.' };
    - if (!formValues.desc) { errors.desc = 'You must enter a desc' };
    - return errors;
    - };
    -
    - export default reduxForm({
    - form: 'streamCreate',
    - **validate: validate**
    - })(StreamCreate);
  - Validation occurs right at first render. But we don't want the errors to show right at the start. One method is to show error only after use has clicked out of the field. Look at **formProps.meta.touched**, which is true only if field has ever gained and then lost focus (user selected then unselected it).
  - While doing the above, you may face an error where the error message doesn't display. This is because of Semantic UI causing its CSS display:none as it wants to hide error messages. To fix this, the <form> tag should have a className="error" so that error messages will be shown.



12.1 React FINAL Form

As an update to Final Form.

- npm install --save final-form react-final-form
- For Reducers, no need anymore "formReducer". It is now:
  - import { combineReducers } from "redux";
  - import authReducer from "./authReducer";
  - import streamReducer from "./streamReducer";
  -
  - export default combineReducers({
  - auth: authReducer,

- streams: streamReducer,
- });

- For your form component:
  - Instead of import {Field, *reduxForm*} from 'redux-form', now you import {Field, Form} from 'react-final-form'.
  - The validate() is now inside the component.
  - The render() behaves differently. It will now become:

```
render() {
    return (
      <Form
        initialValues={this.props.initialValues}
        onSubmit={this.onSubmit}
        validate={this.validate}
        render={({handleSubmit}) => (
          <form onSubmit={handleSubmit} className="ui form error">
            <Field name="title" …"/>
            <Field name="description" …/>
            <button className="ui button primary">Submit</button>
          </form>
        )}
      />
    );
  };
```

Chapter 13: REST-Based React Apps

- To simulate the API server in our project, we will use the **JSON Server** (npmjs.com/package/json-server). It strictly follows **RESTful conventions**, which is a standardised system of defining different routes on an API that works with a given type of records.
  - As per the below picture, it refers to a standardised way of Routes and request Methods used to commit or operate all those different Actions.
  - For eg, if you want a list of all records, make a GET request to the route /streams on our API server. If our server follows RESTful conventions, then whenever it receives a get request to /streams, it returns an array of all the different data it holds.
  - As an anti-example, you CAN choose not to follow the conventions, and have anyhow-named requests to do something. But it is harder to work with across many engineers, so best is to follow RESTful rules.
  - Other programmes like Express do not follow RESTful conventions very strictly.

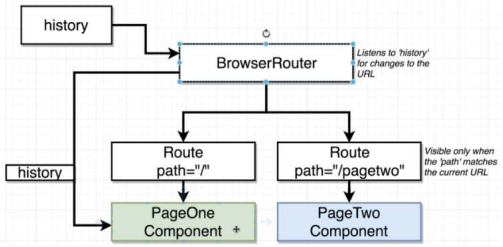
Action	Method	Route	Response
List all records	GET	/streams	Array of records
Get one particular record	GET	/streams/:id	Single record
Create record	POST	/streams	Single record
Update ALL properties of a record	PUT	/streams/:id	Single record
Update SOME properties of a record	PATCH	/streams/:id	Single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

Steps

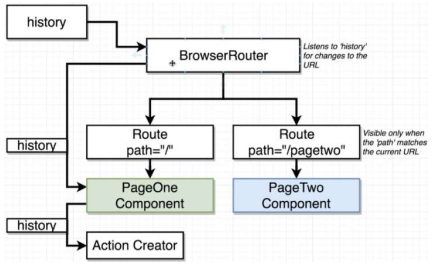
- Create a new folder **mkdir api**, cd to it, then type **npm init** in it (press Enter for every question). It will create a **package.json** file to allow installation of JSON Server. Inside that api folder:
  - npm install --save json-server
  - Create a **db.json** and treat it like a database. Any new entries will be stored here.
  - In **package.json**, delete the *test* line in *scripts*. Replace with "start": "json-server -p 3001 -w db.json". This means start the json-server on port 3001, and watch db.json for any changes.
  - To CRUD, talk to the Route based on the picture above. We will use ACs to send the request via axios (with RThunk!).
  - For axios, baseURL is localhost:3001.
  - Create an AC that uses the outer-inner function structure that we see with RThunk. So it will be an axios.create({…}).post('/route', formValues);.

13.1 Navigation of Users

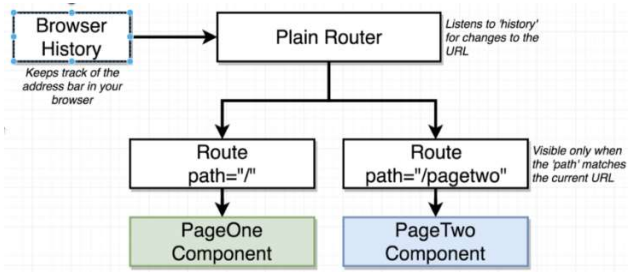
- There are two types of navigations: **intentional navigation** (when user clicks on a Link component) and **programmatic navigation** (we run code to force users to navigate to a different page).
- When doing the latter, we should only navigate user away when we get an API response with success (or error).



- As mentioned previously, the BR will internally create the *history* object and pass it down as a prop to the components. The components can then easily trigger some navigation event.



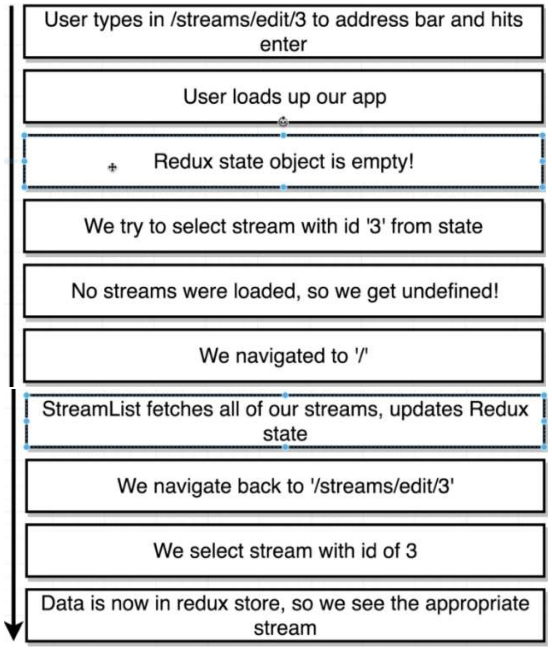
- But what if your trigger is happening inside a non-React component, such as an Action Creator? One method would be to any time our component calls the AC, the component should pass the *history* object into the AC. But this is not ideal too because we must make all our components call the AC with the history object (?).



- Another solution is to manually create the *history* object instead. It will be created in a dedicated file inside of our project. Any time you want to access it, can import it and use. RRouter will have no control over it.
  - Note that when we create the history object, it should be the corresponding type to whatever Router you have. For eg, the BrowserRouter created a browserhistory object, which only looks at everything after the domain:port part of the address. This is different from the other Router types.
  - Instead of BR, we create a plain Router. While you can pass down history objects as props down Router, BR creates its own, it will not use custom history objects. Hence, create your own Router.
- Steps:
  - Create a history.js inside src.
  - import {createBrowserHistory} from 'history';
    - This *history* package was installed with React Router DOM. Hence in actuality, the *history* object is a dependency for RRDom.
    - createBrowserHistory() allows you to create a new history object.
  - export default createBrowserHistory();
  - import Router from 'r-r-d' and history.
  - **<Router history={history}> … </Router>**
  - **history.push('/path');**

- That is all. It will work similarly the same as before, just that now you have more control over the history object.

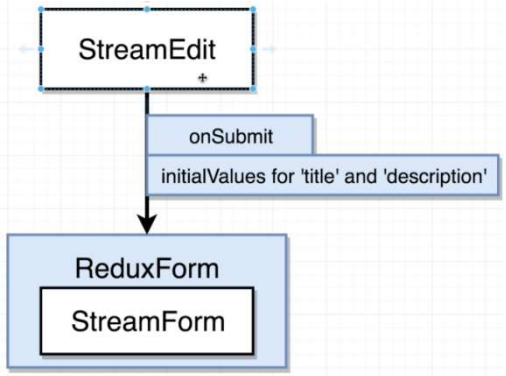
URL-Based Selection

- Navigation also includes **URL-based selection**. This is only possible because of React Router DOM. This is when you put the ID of the stream in the URL, eg localhost:3000/streams/edit/**:id**. The **colon:id** part of the URL is a variable. Using RRD, we can look at the URL and just pull off that portion of the URL, eg for lh:3000/s/edit/17, we can pull that 17 off as a prop into the component, which then knows to show just only data for 17.
  - For Route: path = "/stream/edit/:id".
  - For Link: to = {`/streams/edit/${stream.id}`}.
- If you console.log(props), you will realise you have many props. This is because Route is helping to render the component. All are useful props, but we will focus on **match** for now.
  - The *id* is found on props.match.params.id.
  - You CAN have multiple wildcard params. So a path with /s/edit/:anything/:smthelse or s/edit/hello/yay will see params.anything: "hello" and params.smthelse: "yay".
- Sometimes you might face *streams: undefined*. Why?
  - 

  - AKA: The streams-fetching logic only happens in the root Route (ie lh:3000/ only).
- Hence **when using React Router, it is very important to obey this rule – each components needs to be designed to work in isolation (fetch its own data!).**

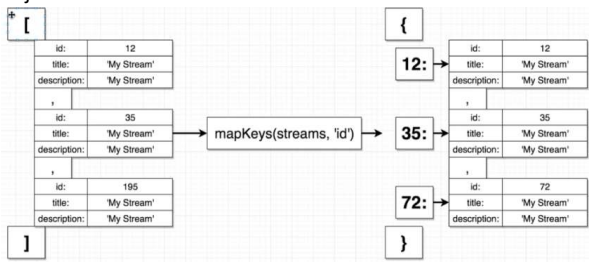
- There are some issues with React Router. Suppose you have this:
  - <Route path="/streams/new" exact component={StreamCreate} /> <Route path="/streams/:id" exact component={StreamShow} />
  - Because ":id" is a variable, then at lh:3000/streams/new, you will show BOTH StreamCreate and StreamShow! Because as a wildcard variable, it shows on any URL of structure lh/streams/X.
  - To solve this, we import from R-R-D the **Switch** component. We place it to wrap all the Routes. What it does is Switch looks at all the different Routes and will only show the first child that matches the path.

13.2 Props with Redux Form



- When we pass down props from a component into a ReduxForm-wrapped component, we are actually passing into the RF, which then pass down into StreamForm.
- There are special prop names for you to use – one is **initialValues**. You MUST pass down a prop from StreamEdit called initialValues into StreamForm. Must be that name. Then StreamForm can use those values as its initial values.
- <StreamForm initialValues={{title: 'EDIT ME', description: 'CHANGE ME TOO'}} />
  - Notice the {{}}. Double. Because sending in an object.
  - The properties *title* and *description* will be matched to the Fields with the same *name*. This is how Redux Form works.
  - Or just do this: <StreamForm initialValues={this.props.stream}/> because it already is an object with those properties. **BUT** doing this is improper because your this.props.stream contains things like id and userId too which you do NOT want to change.
  - One idea is to create an object with variables, ie initialValues = {{title: ${…} …}}. But there is a shorter way with Lodash.
  - With Lodash: initialValues = { **_.pick(this.props.stream, 'title', 'description')** }. The _.pick() allows you to just pull those properties out and returns a new object.

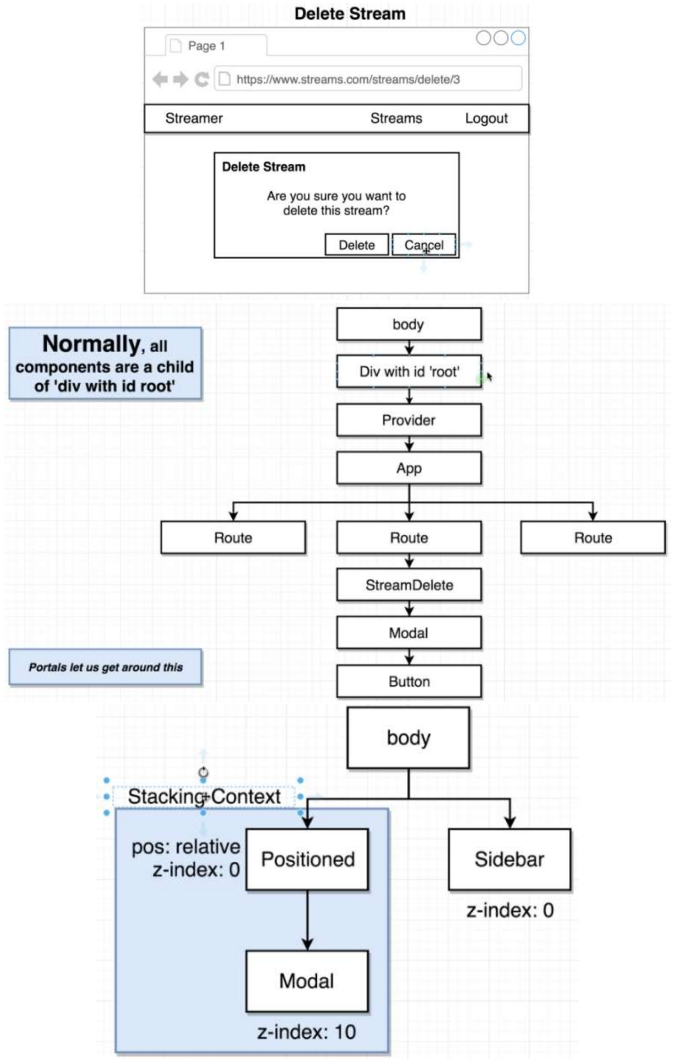
13.3 Other Advanced JS

- **Key interpolation** is a ES2015 syntax that allows you to create
  - Example of a reducer:
    - const xreducer = (state={}, action) => {
      - … case EDIT_STREAM:
        - ❖ return { …state, **[action.payload.id]: action.payload** };
    - Remember that reducers need to return a new object or else Redux will think no change. Hence, the spread operator.
    - The arguments after it means to create a new k-v pair, with the format of **[k]: v**. We are NOT creating an array for the key; just take it that it is just syntax. The *key* is an expression, so we need the [] because we don't know ahead of time what *key* represents, eg it can be 1 / 2 / 3 / 4 / 5…
- More with Lodash's **mapKeys**.
  - _.mapKeys(arr, 'key') is a function that converts an array of objects arr to return an object of objects (a dictionary of objects). To choose the keys for the objects in the new object, declare what *key* is by choosing one of the attributes of the original inner objects.
  - 

  - This was used to help us turn the array of entries into an object:
    case FETCH_STREAMS:
    return {...state, **...**_.mapKeys(action.payload, 'id') };
    - Return a new object that copies whatever *state* was.
    - Red part takes the array *action.payload* and converts it into an object with *id* as the key.
    - The green … tells you to merge the two objects. Because right now, …state is an object, but …mapKeys() also returns an object. The second … then tells you to extract all the inner objects inside mapKeys() and add them into *state*.
  - To convert an object back into array, can use the built-in **Object.values(obj)**.
- Remember the difference between PUT and PATCH. Note that even using PUT, it will not remove the *id*. But note some backend servers manage the conflict even if use PUT. Just for you to be aware of.

Chapter 14: React Portal

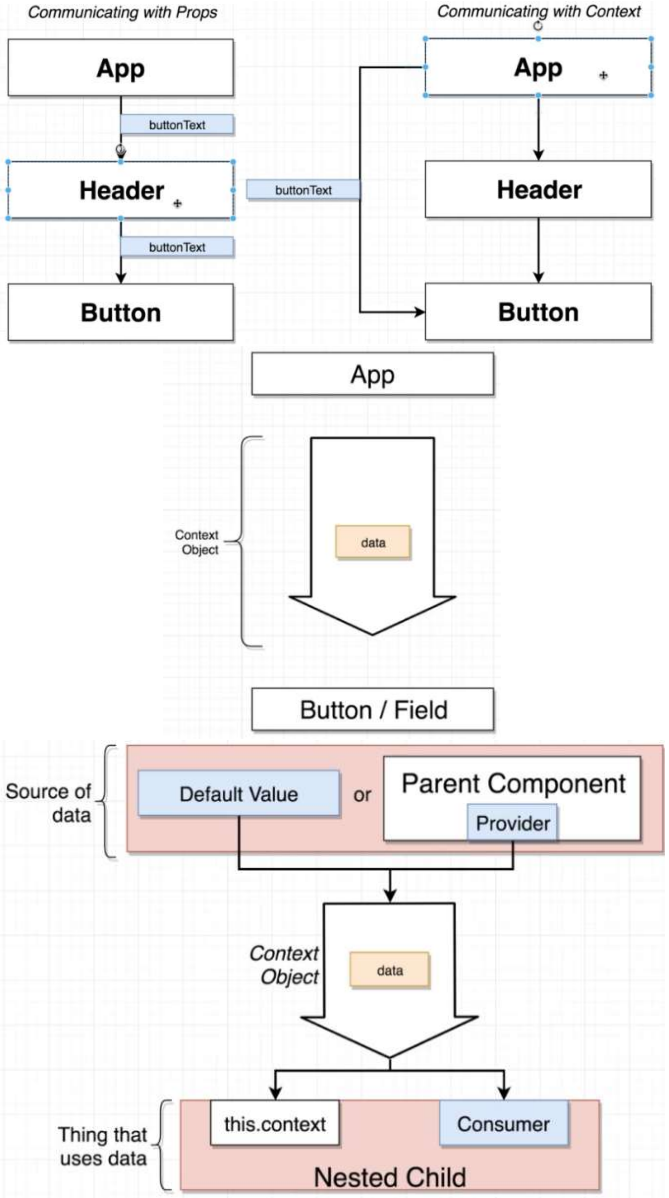
- Sometimes when we want to delete something, we want to confirm with the user first with a little pop up called a **modal window**. A user cannot do anything else until they interact finish with that modal window.



- Because Modal is so down below the chain, and yet because of how it works (slightly blur background but still show it at foreground and middle), and yet all its parents have weird stylings, it will be hard to display it while working along all the various component's stylings.
  - **Stacking context** will cause the comparison to change: it will be Sidebar vs the root of the stacking context ie Positioned. Sidebar is placed later in the code so it appears. There are many ways why SC happens, but position: relative is one.
  - Using **React Portals**, we don't have to stick to this hierarchy of "everything a child of div-root", but we can rather than having Modal render as a direct child of StreamDelete, render it as a child of another element up the hierarchy such as *body*. After that, we can set **z-index** to as high as possible to ensure it has priority.
  - To reiterate, a Portal allows us to render some element not as a direct child, but instead render under some other element in our HTML structure, usually the *body*. Doing this allows to get around the stacking context issues.
  - Remember, you use Portal when you want to render some React component into some HTML that was not created by your React application, ie from a third-party like backend etc.
- Steps:
  - Inside index.html, add a sibling of after where id='root'. This is because we want to attach Modal to inside this block; if it were attached to the #root block, it would replace all our contents instead.
  - Modal.js:
    - import React from 'react';
    - import ReactDOM from 'react-dom';
    -
    - const Modal = props => {
    - // first arg = JSX to display, second arg to anchor
    - return ReactDOM.createPortal(
    -
    -
    - texthere
    -
    -
    - , document.querySelector("#modal"));
    - };
    -
    - export default Modal;
  - When user clicks on the background, the Modal should disappear. One method of doing this is when click, we use the *history* object to send user back to root page. Done by making the most parent element with onClick = {() => history.push('/')}. Then to prevent event propagation, on one of the parent elements use onClick={e=> e.stopPropagation()}. So those clicks in between these two elements with onClicks will leave the page.
  - Remember **event propagation**, where if you click on a child element and that element does not handle an onClick event, it would bubble up until it gets caught by a parent element's onClick listener.
  - But also remember to make your Modal reusable!
  - See stream_trials1's Modal.js and StreamDelete.js for info!

Chapter 15: Context System

- While the Props System is about getting data from parent to a direct child, the Context System is about getting data from a parent to any nested child component.
- It is NOT a replacement for Redux. It cannot do things like make async-await API calls with Thunk, nor make a central repository accessible to all components etc. It is merely just passing props down but skipping a few in between components.

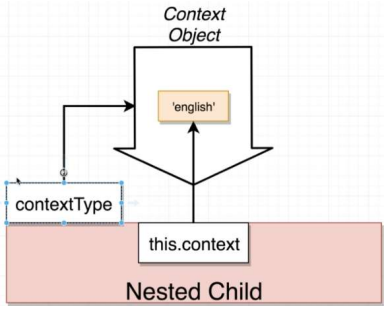


- What happens is we will use a **context object** to help transport props. The key is knowing the 2 ways to get data into the object, and 2 ways of getting data out of the object.

- **INTO**:
  - Set up a **default value** when our CO is created.
  - Inside our parent, create a **Provider** component which pushes info into the CO. Not the same Provider in Redux, just share same name.
- **OUT**:
  - Use **this.context**.
  - Inside the target child, create a **Consumer** component.
- There are scenarios where you may choose to use each method.

- You may want to create the CO as a separate JS file and only those components that need to use it will import it (as opposed to putting into the main folder). All you need to do (yes so simple!):
  - import React from "react";
  - export default React.createContext();
  - Creation is simple 2 lines. It is using it which is most important!

Method 1 – Default Value and this.context

- This method is very limited because your defaultValue is always constant and cannot be changed. See the Field.js as example.
- In your CO file: export default React.createContext(**'defaultValue'**);
  - defaultValue can be anything. A string, integer, array, object etc.
- In target component:
  - import LanguageContext from '…';
  - 

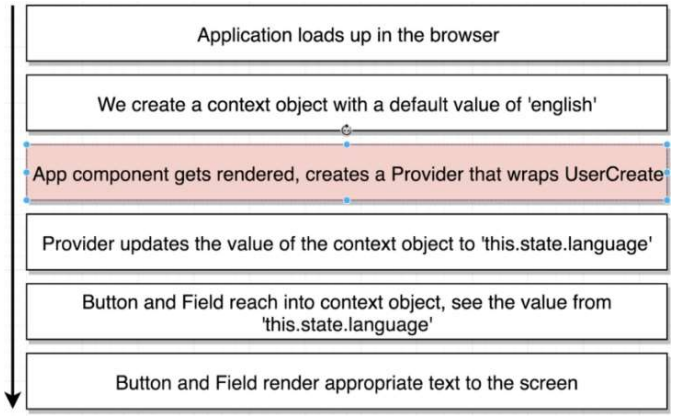
  - Set up a **context type** to link the CO and your component:
    - static contextType = LanguageContext;
    - Must be named contextType! Special name.
    - *static* adds a property to our class (Button) itself, as opposed to an instance. It affects ALL instances of the class / object.
  - Use **this.context** to get the info.

Method 2 – Provider and Consumer

- Figure out which component is the source of info. For eg, the App component was the place that hosted {state} and to choose if English or Dutch, so it is the component that wields the source of truth. We create the **Provider** here. Import the CO here.
- Figure out the component to send the data to. Wrap it with the Provider, eg:
  - <LanguageContext.Provider value={this.state.language}>
    <UserCreate/>
    </LanguageContext.Provider>
  - MUST have the **value** prop and named so like that. This is the value you want to put inside the CO. When it changes, it changes the info it sends to CO as well.
    - It can be any data type like strings or arrays etc.
- Go to the exact target child (aka the Button etc.).
  - No static contextType stuff.
  - Find the spot where you want to extract the info. Wrap it up with <LanguageContext.Consumer></LC.C>.
  - In between them, it will always be a function that we want to pass to it. Technically, it is a child that we are sending it. It will always be a function. This function is going to be automatically called by Consumer with the current value inside the pipe, and that value will be the first arg.
  - Basically we are providing a function as a child to the Consumer component. It will take that child function and auto invoke for us.
  - Example:

```
render() {
    console.log(this.context);
    return (
      <button className="ui button primary">
        <LanguageContext.Consumer>
          {(value) => this.renderSubmit(value)}
        </LanguageContext.Consumer>
      </button>
    );
}
```

- All these work because when you console.log() the CO, you realise it has the Provider and Consumer properties inside it.



- But there is a big gotcha – the red part. Every time you reload, you re-render out of an instance of Provider, you are also creating a brand new pipe to convey info from parent to child (as opposed to using the old one). Every time you use Provider, you create a new channel of info flowing down into a separate set of components.
- Hence must use Consumer.

Method 1 VS Method 2

- Consumer is better when you want to get info out of multiple different COs inside of a single component. Method 1 is more for accessing a single CO.
- Each data needs its own unique CO.
- For multiple COs, no difference on which Provider wraps who first.