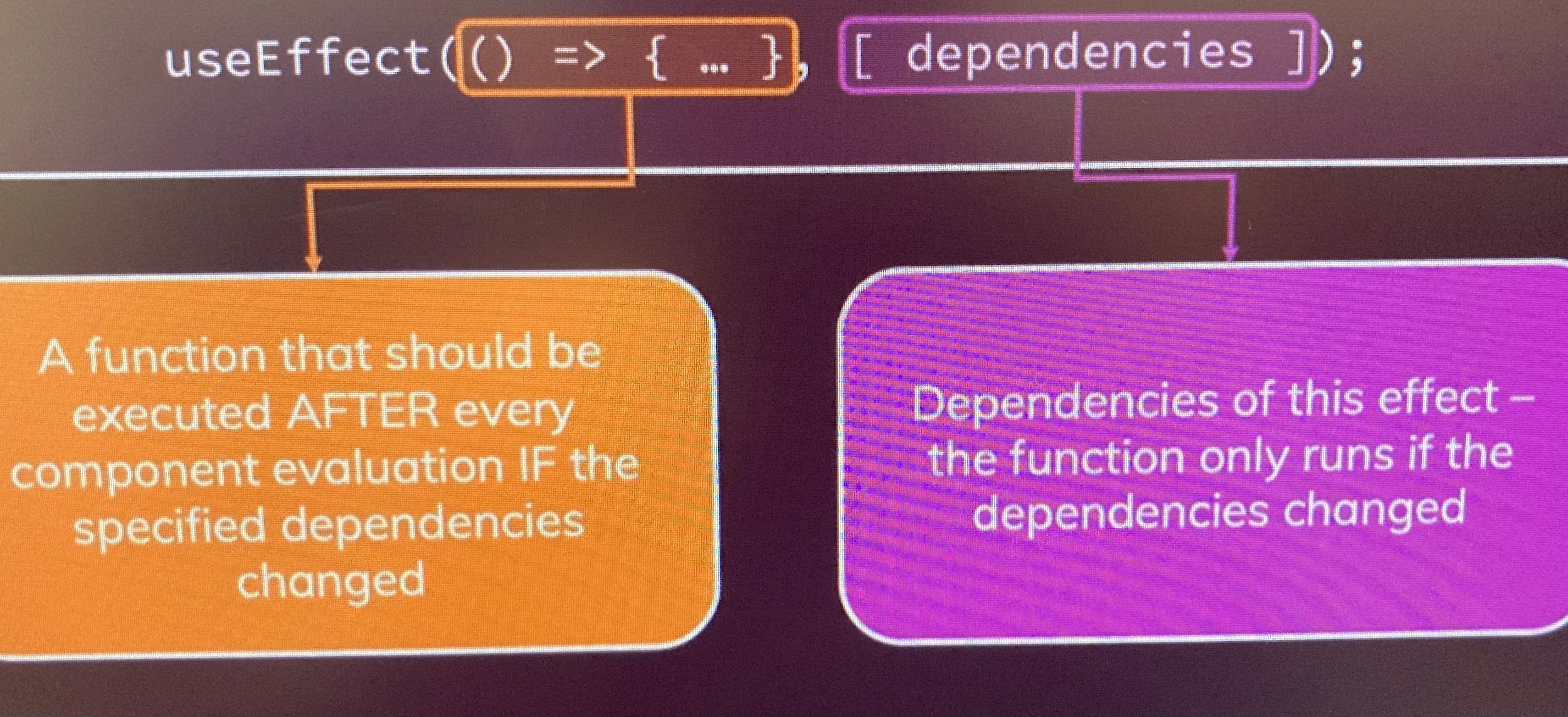
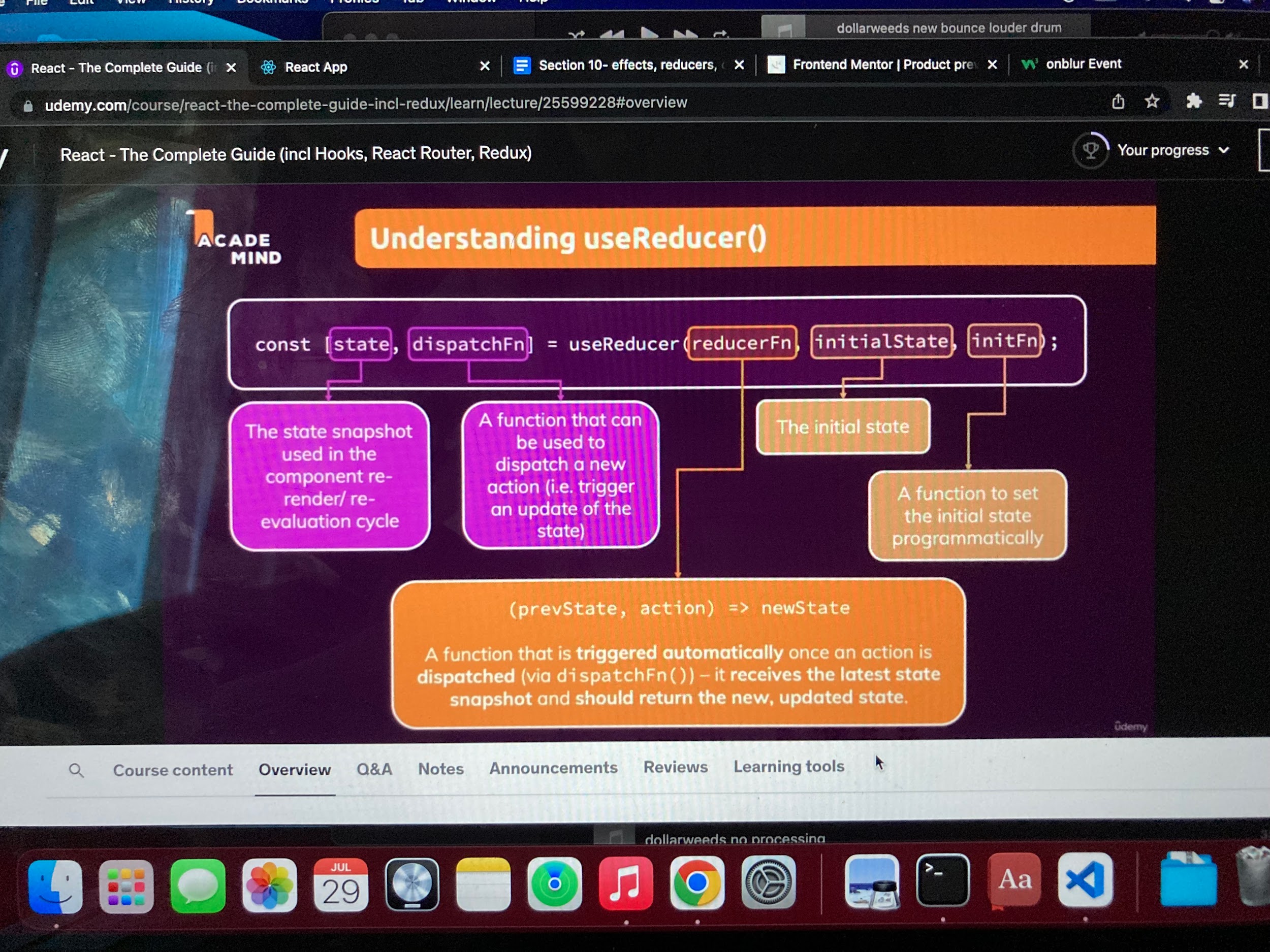
Section 10- effects, reducers, context

* Advanced features
  + Working with side effects
  + Managing more complex state with reducers
  + Managing app-wide or component-wide state with context
* **What is an Effect (side effect)**
  + Components in react app, app as a whole, react library itself have one job– render UI, react to user input, rerender when needed.
    - Evaluate and render JSX, manage state and props to make sure components have needed data and to reflect user input correctly, react to user events and inputs, re-evaluate components and their JSX code upon state and prop changes and manipulate real dom as needed
    - All baked into features covered so far and future features etc..
    - In typical react component– component is just a function executed from top to bottom, everything deals with bringing something to the screen and/or dealing with user input
  + **Side Efffects**
    - Everything else that might be happening in your application
      * Ex. sending http requests or storing something in browser storage(local storage)
      * Timing or intervals too
      * Sending http requests to backend servers
    - These tasks are not directly related to bringing something onto the screen^
    - React doesn’t care about these tasks as it is primarily focused on bringing things to the screen
    - As a result these are tasks that happen outside of the normal component evaluation and render cycle - especially as they migh block/delay rendering ex. (http requests that can fail)
      * If we did send http request as we normally handel components, if in response you could trigger infinite loops, bugs, or too many requests…
    - Better way of handling these-
      * **useEffect() hook**
        + Another built in hook (function you can run inside component function)
      * Called with 2 args
* 
* 1st
  + Function
    - runs after every component eval. If specified dependencies change which is the 2nd argument you pass
      * When dependency changes the function reruns
* 2nd
  + Specified dependencies
    - Array full of dependencies
      * Whenever dependencies change the first function re runs
* Basically
  + In the function you can put any side effect code that will only execute when the dependencies specified by you change and not when the component rerenders
  + Everytime you reload a page the web, browser, react, scripts restart… this is not react specific..
  + Nice to take data and store it somewhere so it persists reload
    - Also when app starts we check if the data was persisted, if so we load user in automatically so they don’t have to reenter details
    - Thats where we could use useEffect()
* **Storing data**
  + Ex. store state change in browser storage
    - Common cookies or local storage
      * Browser storage independent of react
    - localStorage is a global object available in browser
    - localStorage.setItem(‘isLoggedIn’, ‘1’)
    - Access localstorage in dev tools, application and yo ucan see changes you set
    - Useful for storing login info, but useEffect not needed as it runs in a button handler function
    - Ex. where useEffect is needed is refreshing page to avoid loop
      * Use it in useEffect and it is evaluated by react after every component reevaluation, this runs and if state is update, components run again, but useeffect only runs again if dependencies change.
      * Data fetching is not directly related to UI, and this is where useEffect is very useful
    - With no dependency it will only run once you load the page as the only change is the initial change.. If you want it to update with components more then you establish dependencies
    - Dependency is simple, all you do is include whatever functions you need to be reevaluated and add a pointer to them, except for state updating functions
    - useEffect(() => {
    - setFormIsValid(enteredEmail.includes('@') && enteredPassword.trim().length > 6)
    - }, [enteredEmail, enteredPassword]);
    - This is checking if the form is valid to submit so you don’t have to check multiple times in the email and password handler.. It only runs if the enteredEmail and enteredPassword dependencies change
    - useEffect main job is to handle side effects.. Often http requests, but also if we listen to keystrokes and save data as we do in email, but also want to trigger something else in response…
    - Use effect convenient way to enact code in response so “something” happening.
  + Using an effect for cleanup work
* **Debouncing**
  + Triggers another function component execution could cause problems in a more complex application with http requests for form validation etc.. in this current example it is triggering a change with every key stroke.. Def. not ideal with a request if you send a request for every change. Etc.. can test if a user typed something and wait for a pause instead to validate
  + This is called debouncing..
  + This is a problem if you send http requests as it will send tons of them and can overload and even get you banned from the http server..
  + There is a workaround that allows you to delay using setTimeOut
  + By default the setTimer will only delay the individual keystrokes but they still get individually recognized and sent, but they may just wait a few seconds before doing the same thing…
  + You have to return a “cleanup function” to prevent this
    - Runs before side effect function execution, but does not run before the first run of the function… if you refresh the page it won’t run, but if you trigger the function handler by inputting if that is the case, it will run
    - Now you can take advantage of the fact that the setTimeOut actually returns an identifier for the set timer then run the built in clearTimeOut function within cleanup function
    - Save the setTimeOut in a const then run that as the value in the clearTimeOut function
    - This will clear the timer that was set before this ran… and the timer runs from the beginning.. And remember that the cleanup won’t run when the browser first opens
* **useEffect recap**
  + Very important so important to truly understand.. And the tricky part is knowing what it does and when it does things
  + By default with no dependency or empty [] the useEffect() runs after every component cycle.. Not before, or during but after and that includes the initial time it is rendered when the page is loaded.
  + If you add the [] then it will run only the first time by default
  + If you specify a dependency like result of passwordChangeState etc.. it will run each time that function is run and also the initial component cycle when the page loads
  + Cleanup functions can be added because they run before hte state change function each cycle, but not before the initial component cycle
  + Note for useEffect with [] it runs when it initially renders only, but if there is a cleanup function, it won’t run at all unless the component is removed
* **useReducer()**
  + State management for more complex state
  + Multiple states that belong together, change together or are related
    - If you have tons of state hooks working with teh same things then you can end up with buggy code
  + Can be a replacement for useState if you need more powerful state management
    - Only necessary in certain scenarios as it is more complicated to set up
  + One concern cna be when you update state based off of previous snapshot… you are supposed to use a function.. But only works if you are updating state based on prev snapshot of the same state
  + Usereducer is good option when you have multiple related states.. Like capturing input value and checking if htat input value is valid and/or if you have state updates that depend on other state.. Using state could cause problems if the states don’t update in a compatible way and you could end up with issues
  + Often times if you have multiple states or states that interact, you could combine all into one state.. Ex. like an email state, but with all this in mind useReducer is a good option as well
  + How does it work
    - Similar to useState returns two values in an array… first is the state snapshot used in the rerender/reevaluation cycle, 2nd is function that will dispatch a new action rather than just setting a new state value.
    - Only diff is the function will look a bit diff.
  + Runs into 3 arguments
    - First is reducer function
      * Function that is triggered automatically once action is dispatched(via
      * dispatchFn.. It gets the latest state snapshot and should return the new updated state
    - 2nd
      * Set the initial state
    - 3rd
      * Optional initial function… sets the initial state programmatically if you have a complicated situation with http requests etc…
* 
  + - The reducer function receives two arguments, state and action… action pulls info from where you run the dispatch function… state contains most recent state snapshot
    - When you run the dispatch function in a handler, you create an action… this could be anything, but typically an object that represents values you need… then you can test in the reducer function if it is equal to the action and set to the new value
* **When to use useState vs useReducer**
  + No rule, but you will generally just know
  + If usestate is too cumbersome/bugs/unintended behaviors etc..
  + If you have multiple related states that get updated together indicator u need reducer
  + useState is primary tool and typically what you start off with
    - Best if updates are easy
* **React Context**
  + Alternative to props for passing data/functions that avoids forwarding through components that don’t need the info/function
    - Less commonly usd than props but still used regularly
  + React has Component Wide (behind the scenes) State Storage
  + In bigger apps, passing data from one place to another can sometimes be inconvenient if they aren’t directly related… you would have to pass among many other components that don’t need the info just to pass it along. This isn’t bad, but can be a pain
  + React context allows us to trigger action in component wide state storage then pass it directly to the interested component without building a large prop train
  + Setup new component
  + Import react then use syntax
    - React.createContext()
    - Thru this you pass in default context.. Context in this scenario is app or component wide state.. You can define that state.. Typically object where you manage state of something specific
    - It returns an object that contains components then you can export that object
    - Then you have to do two things with this when you import
  + Provide it
    - Tells react here is my context..
    - All components that are wrapped by it will have access
    - Where you wrap it depends on where you need it
    - Note- when you wrap it is not actually a component so you have to tweak it to make it work in jsx by accessing a property on it called .Property that contains a component
      * <AuthContext.Provider>
  + Consume it
    - Hook into it/listen to it
    - It has initial value you set but you have to listen to it
    - Two options
* React hook
  + Better way
  + useContext hook built into react
  + Call useContext and add pointer to the desired context
  + It returns context value
* Consumer
  + Less common
  + Wrap child component with AuthContext.Consumer
  + Takes a child that should be a function
  + Add function in parent that points to default and changes it
  + Use to call in child
* **Making context dynamic**
  + Allows us to pass not only data to other components, but also functions can be forwarded to components that aren’t directly related
  + Props vs context
    - Usage depends on the situation
      * If you are passind directly to a child then props is the most convenient
      * If you are forwarding or have the need to pass it easily to different components across the app,, context is better
      * One thing to consider is using context on an element like a button would bind that context to that button so props is better if you don’t have a specific thing you always want to happen
* Fine tuning context
  + Good idea to add empty dummy functions to the context script itself for auto completion. VSC will recognize it by the default if you use it elsewhere
    - Not necessary for it to work, but good practice
* **Complete context and state management component**
  + Sometimes good idea to create separate context management component, if you have a lot you are doing in App… you can manage state and everything from somewhere else then pull in app using useContext
  + Basically just take all state and context data and separate it to clean up the app component then wrap the entire app component in index.js in it then use useContext to grab in app
  + Then wherever needed as opposed to props, you load useContext then import the actual context file, then save the useContext(ContextFileExport) to a const of your choice then use the const to pull needed data ex..
* **Context Limitations**
  + Certain situations where it wouldn’t be ideal like for components that are reused.. Like buttons..
  + Great for state management across components or across entire app
    - Still some limitations
      * React context is not optimized for high frequency changes
    - For our example where we were concerned with authentication which doesn’t happen that often it is good, but if state changes happen constantly like multiple per second… react context might not be optimized for that
    - There are other options for app wide or component wide state that changes often where context would be extremely useful but that will come later
* **Learning the Rules of Hooks**
  + Set of rules that should be followed when using hooks
  + Hooks are the functions that start with use
    - useState, useEffect, useReducer, useContext and many more we haven’t got to yet
  + 1st
    - Only call react hooks in react functions
      * In react component function
        + What we’ve done so far
      * In custom hooks (covered later)
  + 2nd
    - Only call react hooks at the top level of your functions
      * Don’t call in nested functions
      * Don’t call in block statements
  + 3rd (extra rule)
    - This one is an addon for useEffect()
      * Always add everything you refer to that is taken from within the function as a dependency unless theres an exception
        + This doesnt include api things baked into the browser etc.. like console.log() or setTimeOut
        + Only data exposed by the surrounding function like state or props data
        + Exception is state updating function like setFormIsValid()... guaranteed by react to never change… no need to add as dependency, but you could add if you want
  + **Forward refs**
    - Allows us to interact with reusable components imperatively
      * Not by passing state that changes something in the component, but by calling a function inside
        + Not done often, but sometimes helpful
    - **useImperativeHandle hook**
      * Allows us to use functions from a component imperatively, or not through typical props/state management, or controlling the state through the parent component
      * Directly calling or manipulation something in the component programmatically
        + Rarely done, but sometimes useful
      * Run it and pass two components
        + 1st

Ref

Note— typically you only need to pass props in the main component function as an argument, but can also pass ref and we would do that here then also run it as the first argument in the useImperativeHandle

For this to work, you have to export the component function in a special way by wrapping it with React.forwardRef()

Makes the main component a component that is capable of being bound to a ref

After this- the component can take a ref prop and is controllable with refs… but only what you expose through the component

* + - * + 2nd

Function that returns an object with all the data you need from outside

* + Recap
    - For focusing inputs this can be very useful
    - With useImperativeHandle and forwardRef you can expose functionalities from a react component to its parent component to then use your component in the parent component through refs and trigger certain functionalities
    - Doesn’t just work for functions, but could also expose the value
      * Typically best to avoid^^ but good to be aware of and may come in handy for focusing and possibly other cases like scrolling