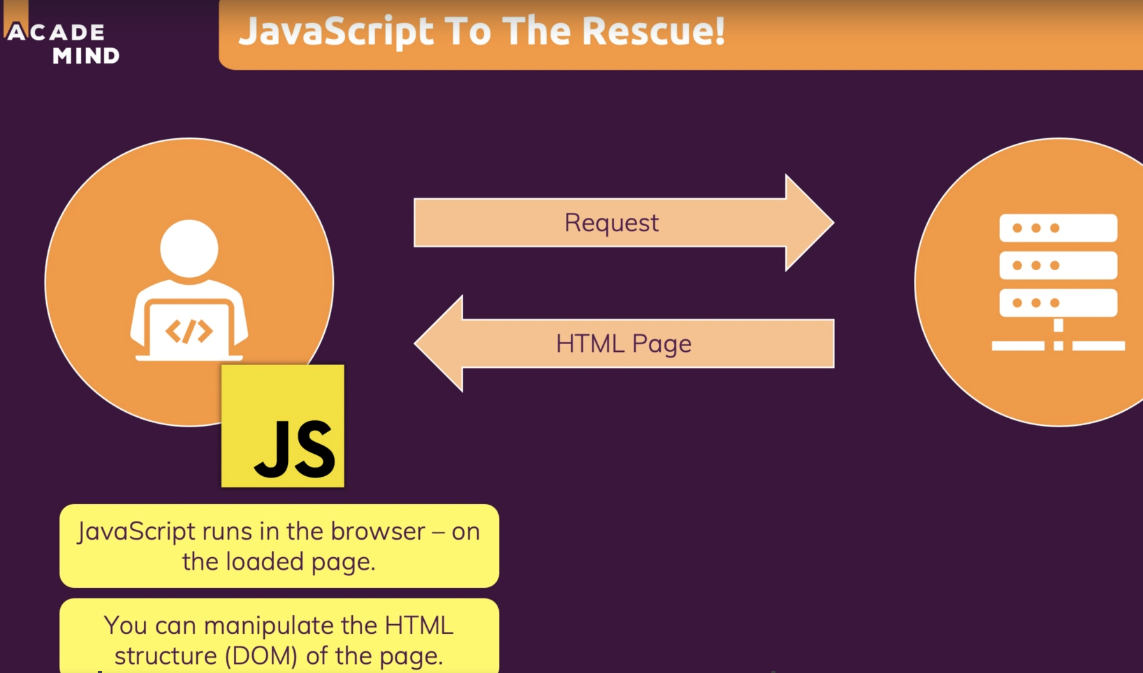
**Getting Started**

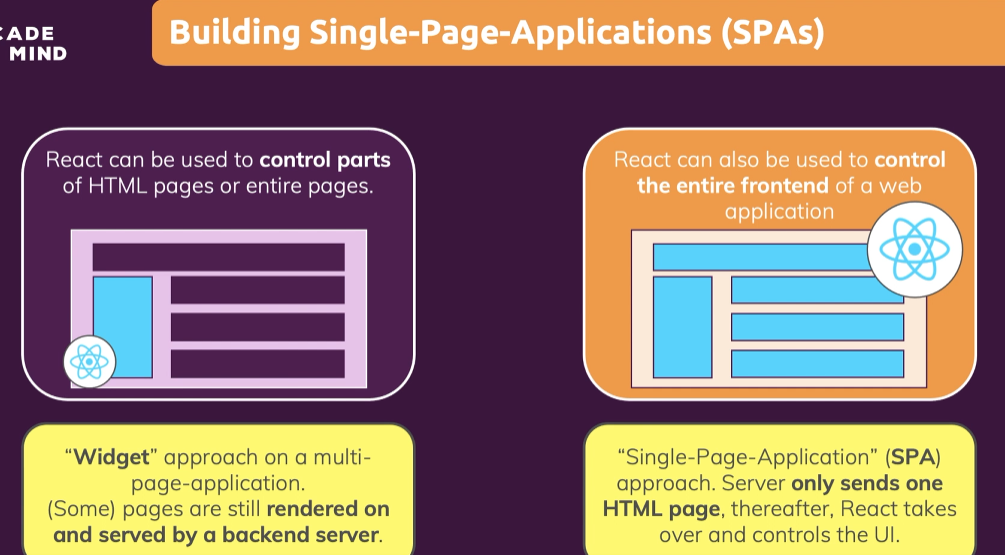
Compared to mobile apps, traditional websites could/can feel a bit slow or clunky because of the traditional request response architecture.

JavaScript runs in the browser, working on an already loaded page. 

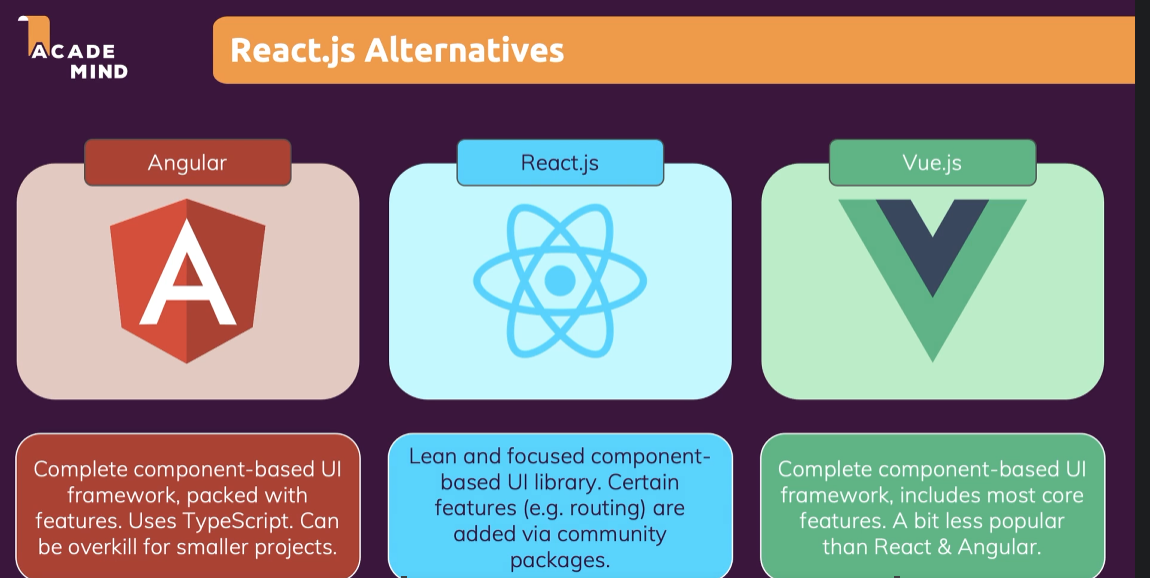
So React.js is a client-side JavaScript library.

Using bare JavaScript alone and describing action after action in excruciating detail is called an ***imperative approach*** to website design.

React.js can utilize two different high-level architectures:



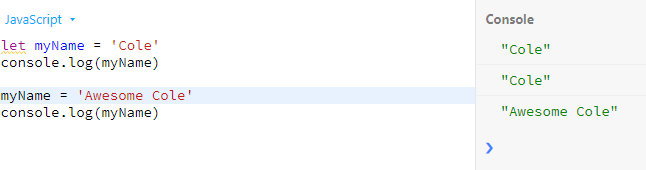
React vs. other JavaScript libraries/technologies:



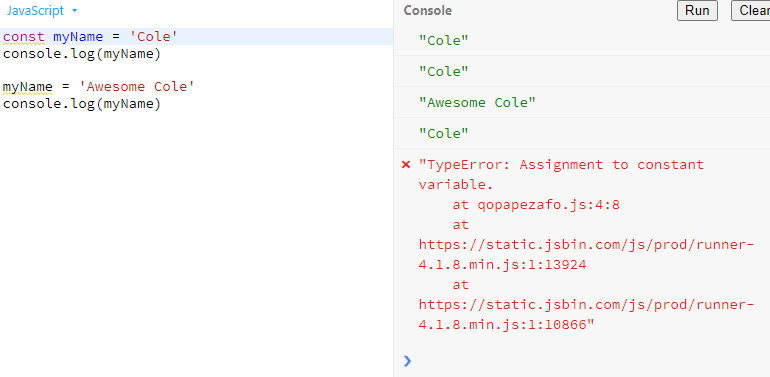
**Section2: Next-Gen JavaScript**

We will be testing JavaScript at <https://jsbin.com/?html,output>

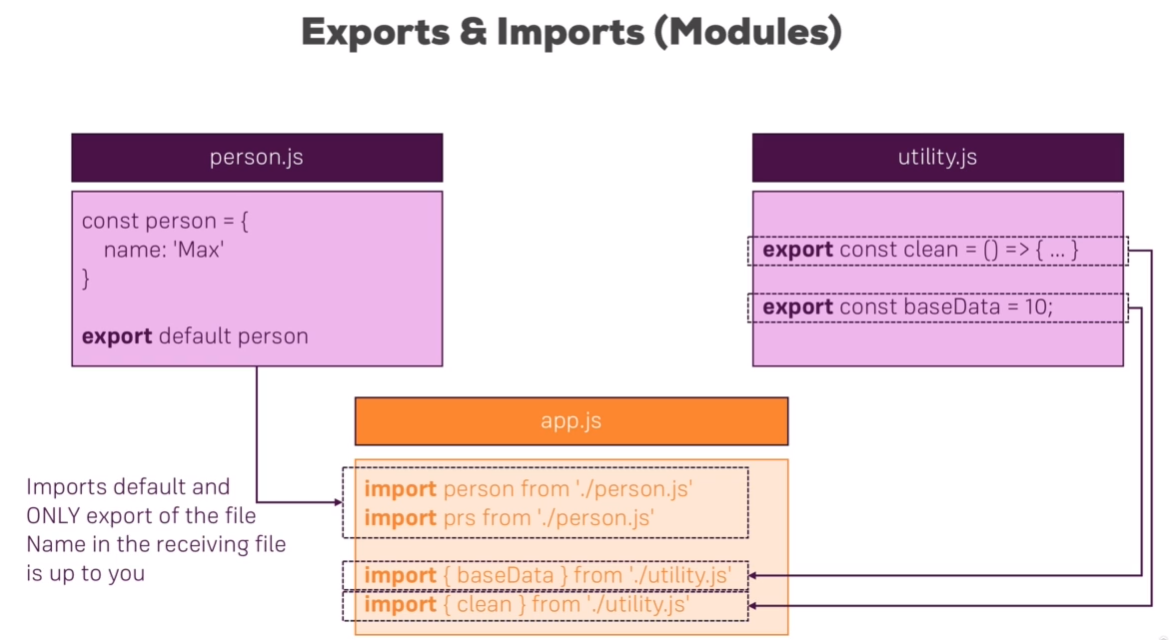
* the old JavaScript way to declare a variable was *var*
  + *var* replacements are *let* and *const*
  + *let* is the “new” var which is indicated for variable values
  + *const* is for truly constant values



versus const throwing an error

* + 

For exports and imports notice the keyword ***default*** and how it affects import statements



Some or all of this export/import syntax is considered “next generation JavaScript” and hence requires some special setup/compilation for it to run in a normal JS project.

You can also write the following to import everything and access the items like *bundled.dataObj*

*import \* as bundled from ‘./utility.js’*

Example of Basic class syntax and inheritance using the *super()* keyword

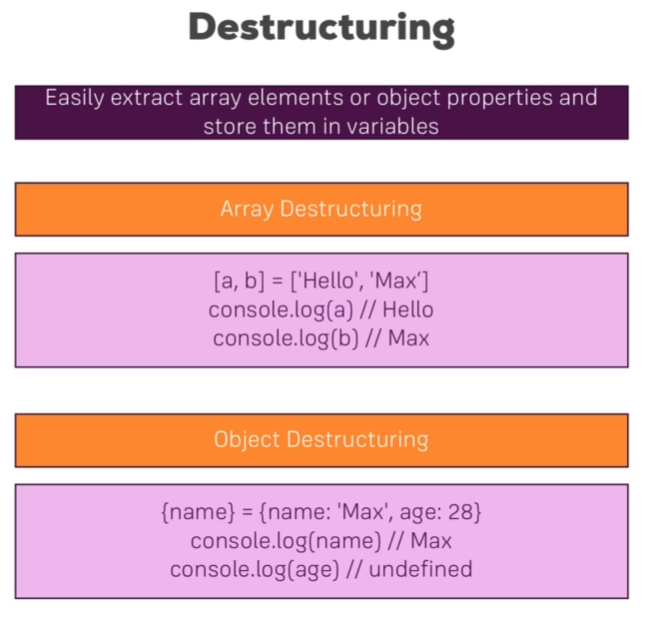
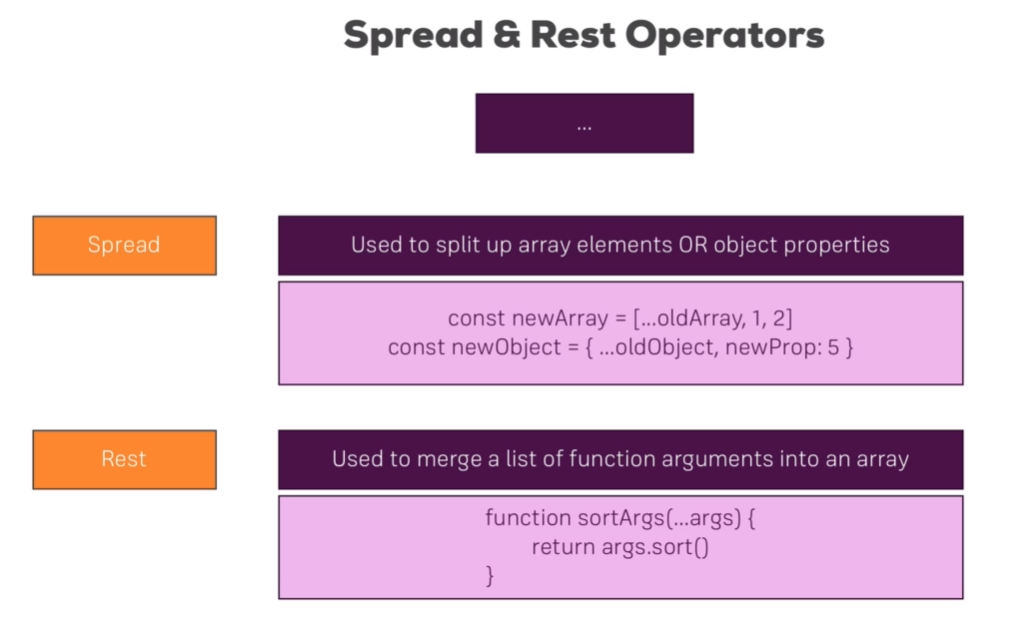


The following code is equivalent (with the addition of an overriding gender) except it utilizes the simpler syntax in next generation JavaScript.

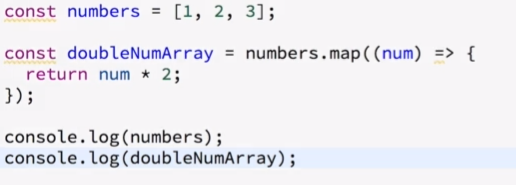


The “spread” operator is similar to zip in python I think.

The “rest” operator is similar to list/dict unpacking, specifically list unpacking I think, in Python



* Objects and Arrays in JavaScript are reference types and hence invoke pointers behind the scenes. Hence they are mutable. You can invoke the spread operator (…) in order to get a true copy because this will **just copy the properties and not the entire object** and hence apparent immutability.
* Using the .map function as below – its not nextGen JS but is normal JS – is a nice way to get a true/new copy

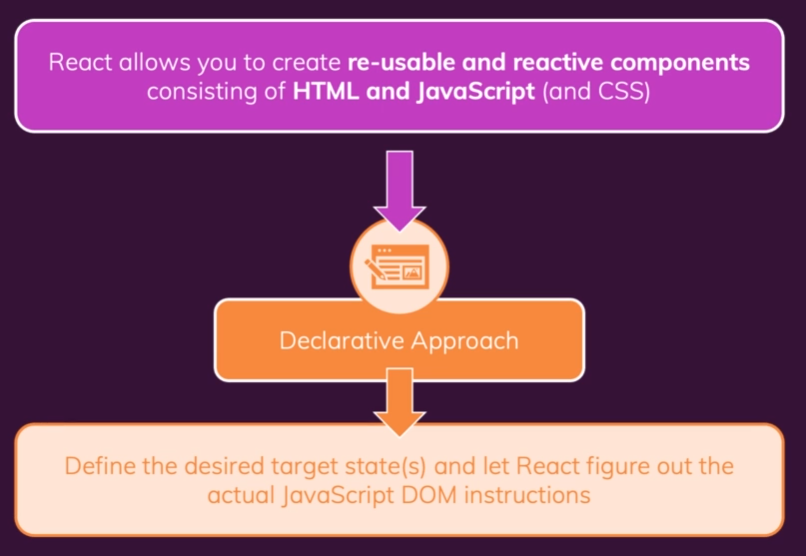


**Section3: React Basics and Working with Components**

We will start by building the Expense Tracker App he demonstrated.

React is all about **components.**

* Components are blocks of html, css, and possibly JavaScript that are:
  + reusable to avoid repetition
  + and provide separation of concerns
* Mostly though React is about combining HTML and JavaScript



You installed **NodeJs** on your Ubuntu machine with:

*curl -sL https://deb.nodesource.com/setup\_16.x -o nodesource\_setup.sh*

*chmod +x nodesource\_setup.sh*

*sudo ./nodesource\_setup.sh*

*sudo apt install nodejs*

*node -v*

Then you created a React project using the established ***create-react-app*** application which auto downloads on running the create command like:

*mkdir react\_projects*

*cd react\_projects/*

*npx create-react-app react-complete-guide*

You can start the default react application by then changing directories into your project and running the below commands. This should automatically open that application on localhost in your default browser.

cd react *react-complete-guide*

*npm start*

**Important Note:** He then had us download a “cleaned-up” starting project instead of starting from the pre-populated project that was generated. He only had us run that to show how we could typically start a new project.

We had to run the dependency install again for this “cleaned-up” project with

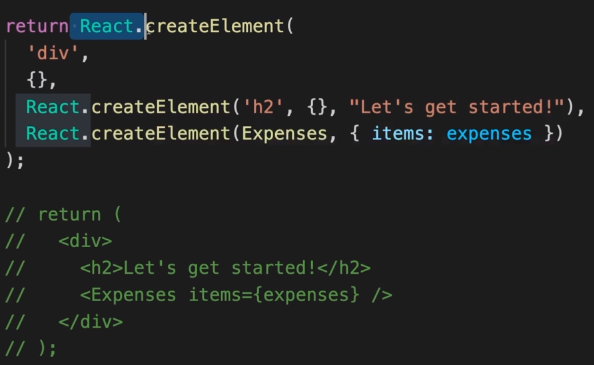
*npm install*

*npm start*

*index.js* is the first file to execute or run. And somehow the *npm start* command turned some parts of this *index.js* file which were technically invalid JavaScript and ran them in the browser anyway by some kind of transformation behind the scenes?

The html code “inserted” inside of *App.js* wouldn’t normally be acceptable JavaScript. But this is part of **JSX**.

JSX = JavaScript XML

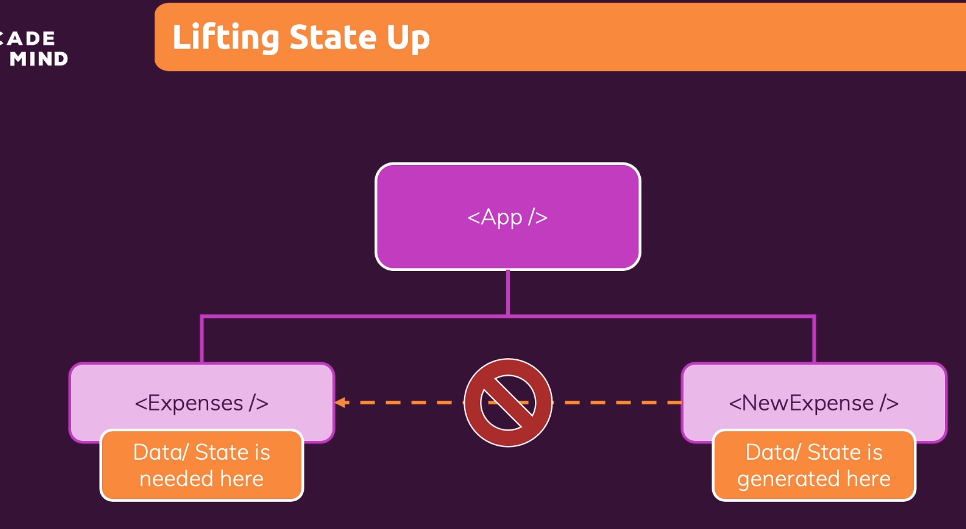
* This JavaScript JSX code is also transformed in the background as part of the npm server or something.
  + You can see the transformed code in the browser’s developer tools under the “source” tab and files with names like *chunk.main*
* Here is an example of old JSX code before it became nicer and didn’t require imports of React everywhere and more explicit syntax like the following:
  + 
* It is considered a best practice to place new components in react in different files.
  + *App.js* is special because it will be the so-called “root-component”
  + Remember react components are just special JS functions
  + Custom Components **must start with an uppercase** character
    - lowercase elements are built-in html elements
* Each component **must only return one root element**.
* You can add css styling using *className* in your divs.
* Example of replacing hard-coded data using {} operator inside tags:
  + <div> Date</div>
  + versus <div>{expenseDate.toISOString()}</div>
* You pass attibutes into child components using the “***props***” keyword which can really be named whatever inside the functions of the child components.
* Building a UI from smaller building blocks (like we did with the Expense stuff) is called **composition**.
  + In web development the term “card” refers to the container-look around some element on the screen that has like specific corner styling and shadowing, etc.
  + He went through adding an examples styling focused wrapper implementation of a *card.js* and *card.css* which I didn’t yet implement
* Arrow function equivalent sytnax:
  + function ExpenseDate(props) {
  + const ExpenseDate = (props) => {

**Section4: React State and Working with Events**

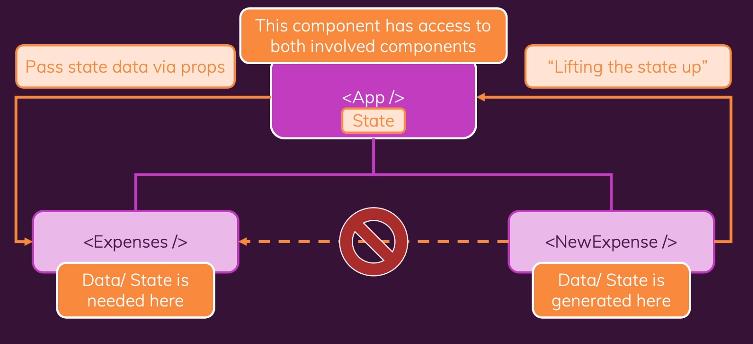
* When passing functions into event-handlers like button clicks you want to pass the function in as {*function}* instead of as *{function()}* because this passes the pointer instead of actually evaluating the function when the code is parsed. We instead want the function to be evaluated on the click of course.
* Many built-in html elements support even listeners and in React you can add one of these with *on<Handlername>* like in *onClick=…*
* In order to tell React that a component needs to be re-evaluated or re-rendered.
  + There are various React hooks to accomplish this.
  + The first example we are seeing is *useState*
    - useState should be passed a variable to monitor
    - it always returns a 2-long-array where [0] = variable , [1] = function to manipulate the variable
    - Practically this function will then **both** update the variable, and re-render the component in which useState lives
* You can combine state variables together so that multiple variables must be changed to change the state or you can do them separately.
  + Separately is much simpler.
  + See lecture 55
* Here is an example of both:
  + using the previous state
  + maintaining all of the previous state variables except updating the value of *enteredTitle*



* You can think of the <input> element in the <form> as a pre-built React component which translates to the HTML dom <input>
* There is no direct connection between sibling components so the following strategy won’t work:



* But we can “lift the state up” and then pass it back down to the sibling like:



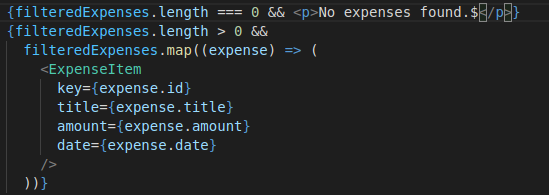
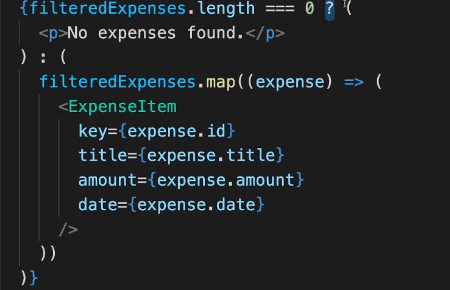
**Section5: Rendering Lists and Conditional Content**

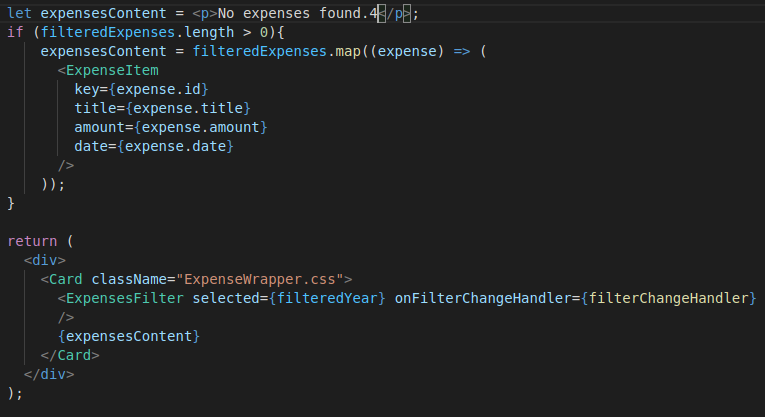
* “key” is an item attribute that you can add to either your custom components or built-in html components to help react identify/distinguish between multiple similar instances/copies of those items. See below example where we use our random “id” as this key.



* The React “filter” method returns a new array, a copied array instance in other words
* He mentioned how for-loops and traditional “if” are not allowed between the React {}
  + but you can use the ternary operator
  + YUCK!
  + In order to get around ugly ternary operator usage: he uses JavaScript && to output the second part if the condition before the && is true

**Example** --- 3 different ways to do the same conditional:

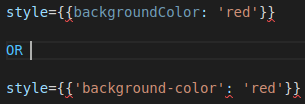




You have an idea of adding a “Clear Expense” button that will clear a selected item.

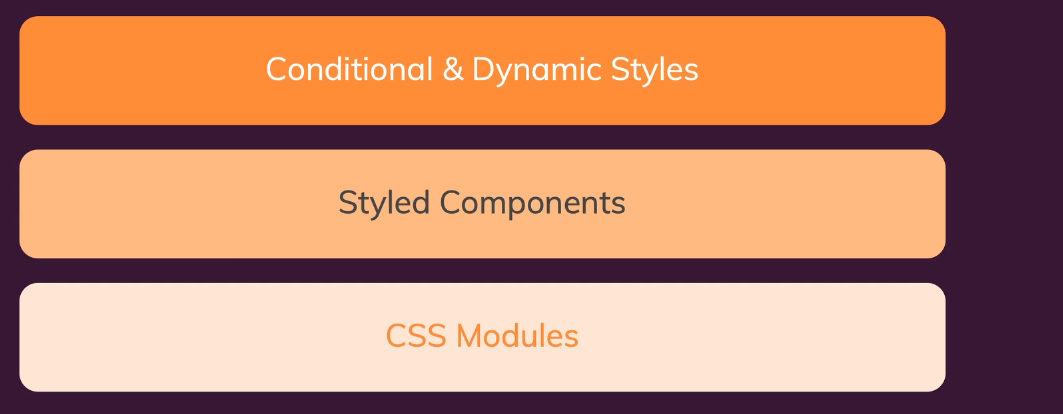
When you are attempting to “style” an individual piece of html using the *style* attribute of say a <div>, you pass *style* its own JavaScript object as in the example below.

If setting a property like background-color which contains a dash you have two possible options:

****

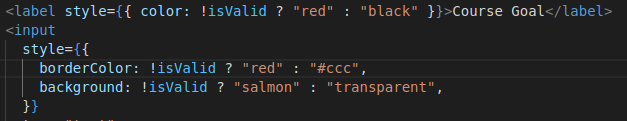
In *ExpensesChart.js* he switched from using a “for … in …” loop to a “for … of …” because apparently this is necessary for an array?

**Section6: Styling React Components**

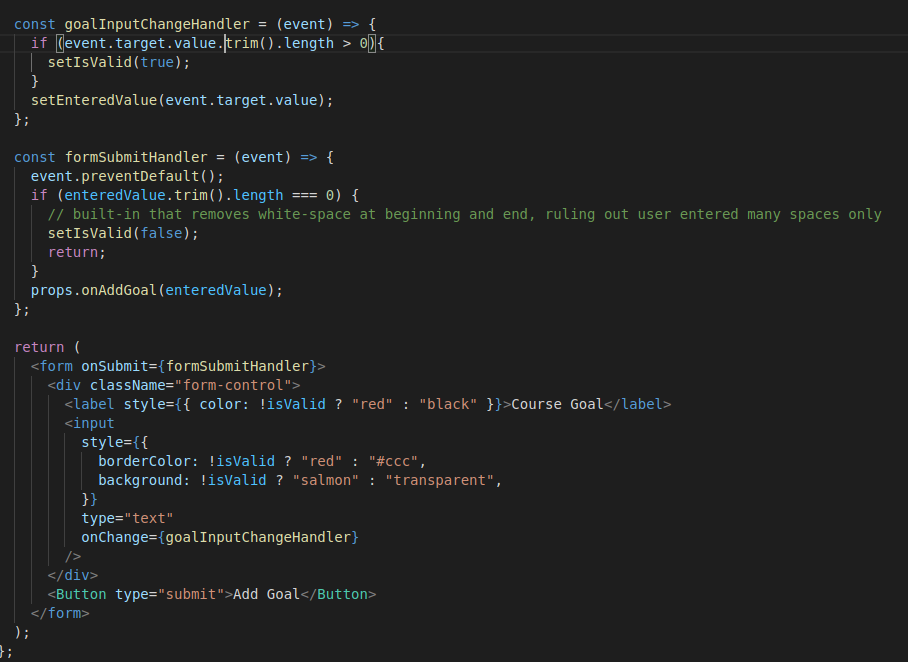


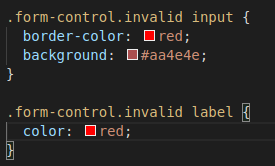
* Examples of **in-line styling** based on a validity flag (uses ternary-operator conditional) 4



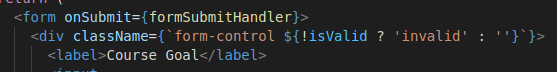


* Entire example for reference before we went another way:



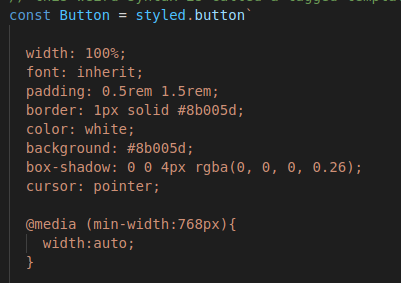


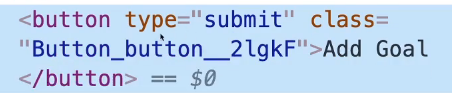
* Dynamically adding styling example:



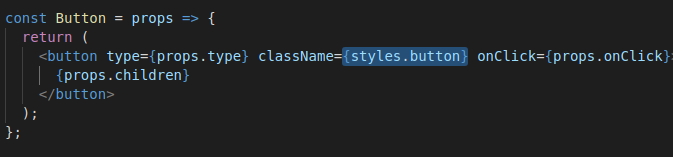
* There are two approaches to avoiding styling spillover because of global css styling tags. 1) use package “styled components” <https://styled-components.com/>

2) use “css modules” which is a feature that is ONLY available in projects that are initially configured to support it.

* + - React projects created with *create-react-app* are automatically configured in this fashion
* We will begin by looking at the approach in 1) or styled components
  + You need to run the following and then restart the react server
    - * *npm install --save styled-components*
  + You can technically add another component in the same file if it is only being used inside of that file
* Working with **Media Queries** for adjusting styles based on computer/mobile-device consumption of the webpage. Specifically we adjust the “width” of a button here.
  + 
* To use CSS Modules you import your css files differently like:
  + import styles from './Button.module.css';
  + You also have to rename your css file accordingly to the above import.
* The core thing CSS Modules does is that it changes the className to be unique by adding a hash like the below that you can see from the browser’s developer tools:



* Two examples:



pre “invalid” addition



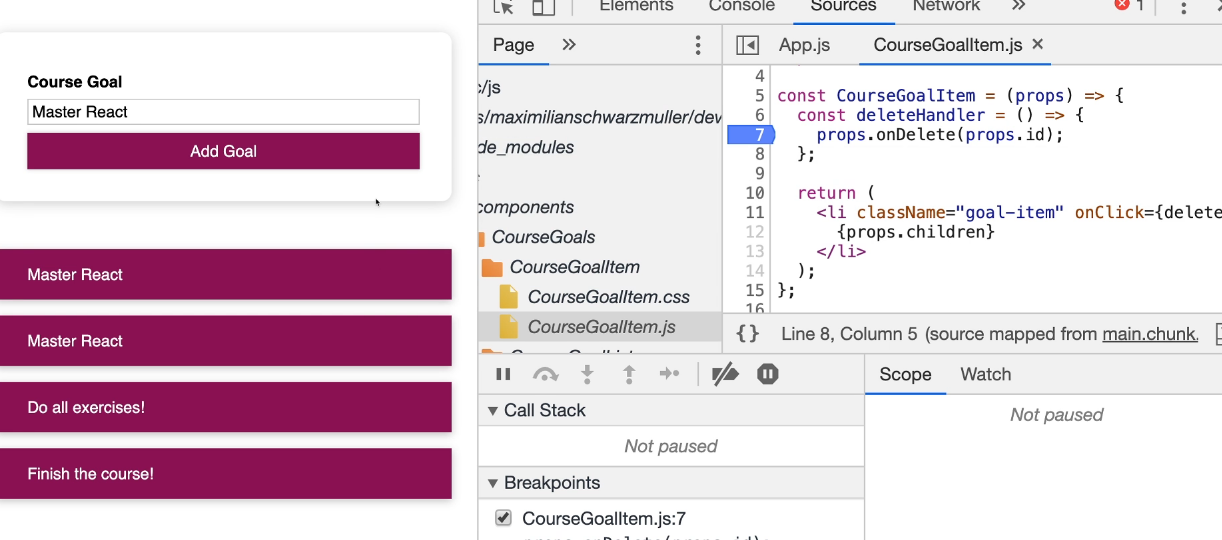
 post “invalid” addition (below)

* Using CSS Modules, in order to fool with media you just do this inside the relevant module.css file like in *Button.module.css* in this example.

**Section7: Debugging React Apps**



* How it looks debugging in the browser with breakpoints:

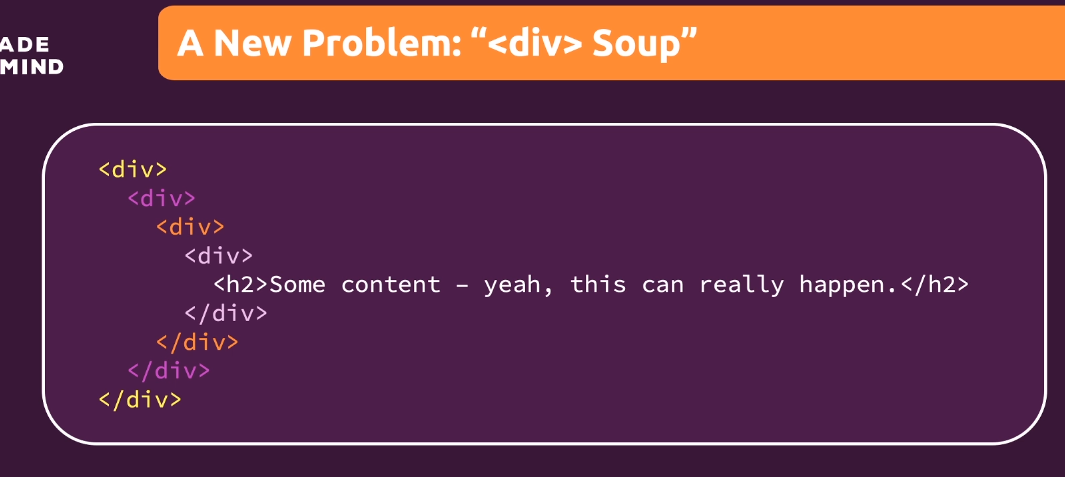


* There are also React Development tools available for Google Chrome
  + after installing the extension you should have two additional tabs available in the normal developer tools:
    - Components
    - Profiler
  + You can actually change the state/props values for whatever components when messing with the “components” view here, and it will update the UI according to how that state/property was coded.

**Section9: Working with Fragments, Portals, and “Refs”**

JSX Limitations:

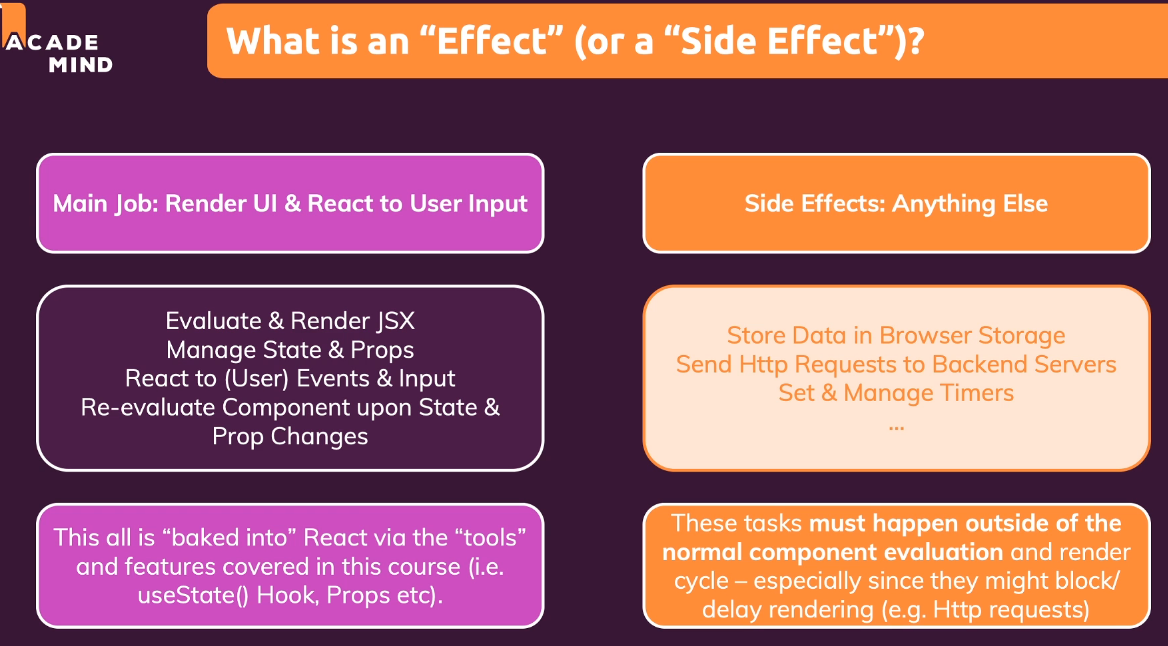
* You cannot return more than one “root” JSX element
  + Of course you can work around this by just wrapping adjacent root elements in a wrapping <div>
    - However, in larger applications you can end up with a ton of wrapping <divs> like in the image below

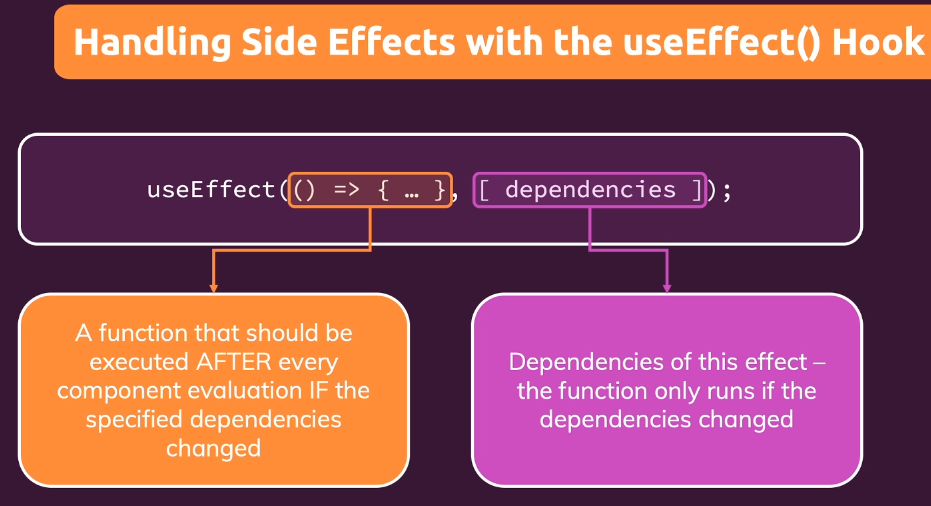


* + A not-as-great workaround would be to wrap in an array and add key tags to the adjacent elements
  + Another **superior**solution is to use an “empty” wrapper component which only returns *props.children*.
    - For this solution nothing is rendered to the DOM, but the JavaScript one-root-element rule is satisfied.
    - The wrapper component is actually a built-in and can be accessed with <React.Fragment> or in some projects just with <>
* Modals or overlays shouldn’t be deeply nested next to the pages they are supposed to overlay under whatever conditions.
  + We first modified public/index.html
    - And basically in the React component where you intend to use the **Portal** you will basically just be “portaling” some portion of that html to the actual public/index.html location
    - This is just like the code in index.js that we started with at the VERY beginning
  + You can use a Portal wherever you would like to move the html code somewhere else
* when you write something like *ref={nameInputRef}* that ref or *nameInputRef* ‘s value to the native DOM element that is rendered for the input it is now connected to
* Refs are probably better than manipulating State and all of the work state-manipulation entails if you are doing something simple like just key-logging
  + Using Refs (in this manner at least) is “uncontrolled behavior”

**Section10: Advanced: Handling Side Effects, Using Reducers, Using the Context API**

Refresher to the big picture of React (left); outside the normal scope of React components’ (right)

****

Part1: Look at the summary **video 115** for a great summary of **“Side Effects”**

* To look at local storage for a webpage go to *DeveloperTools → Application → Local Storage*
* You should add "everything" you use in the effect function as a dependency - i.e. all state variables and functions you use in there.
  + Exceptions to this rule (because they never change):
    - you don’t need to add state updating or “set” functions
    - you don’t need to add built-in api functions like “fetch” and “localStorage”
    - you don’t need to add variables or functions defined outside of the specific component like *myTimer* in the example below

let myTimer;

const MyComponent = (props) => {

const [timerIsActive, setTimerIsActive] = useState(false);

const { timerDuration } = props; // using destructuring to pull out specific props values

useEffect(() => {

if (!timerIsActive) {

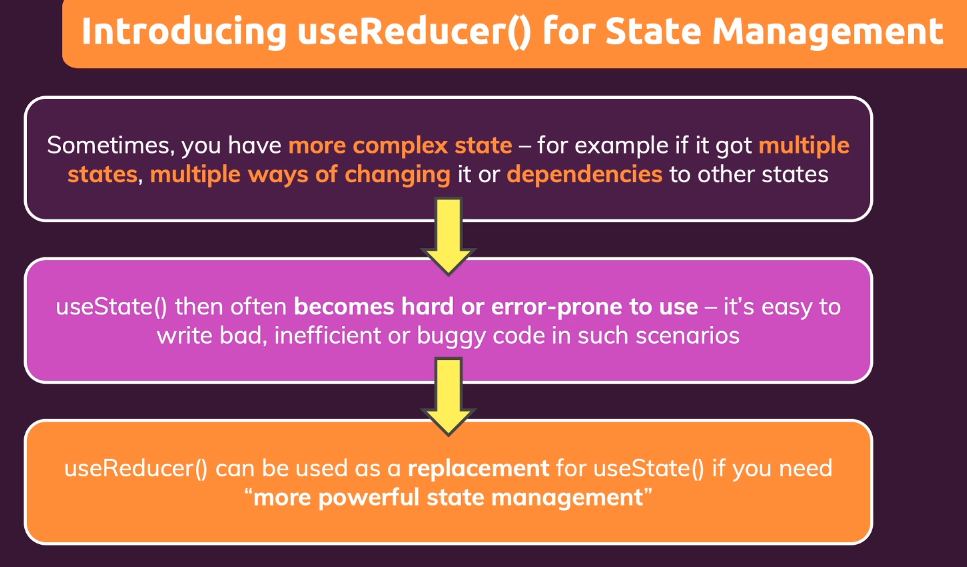
setTimerIsActive(true);

myTimer = setTimeout(() => {

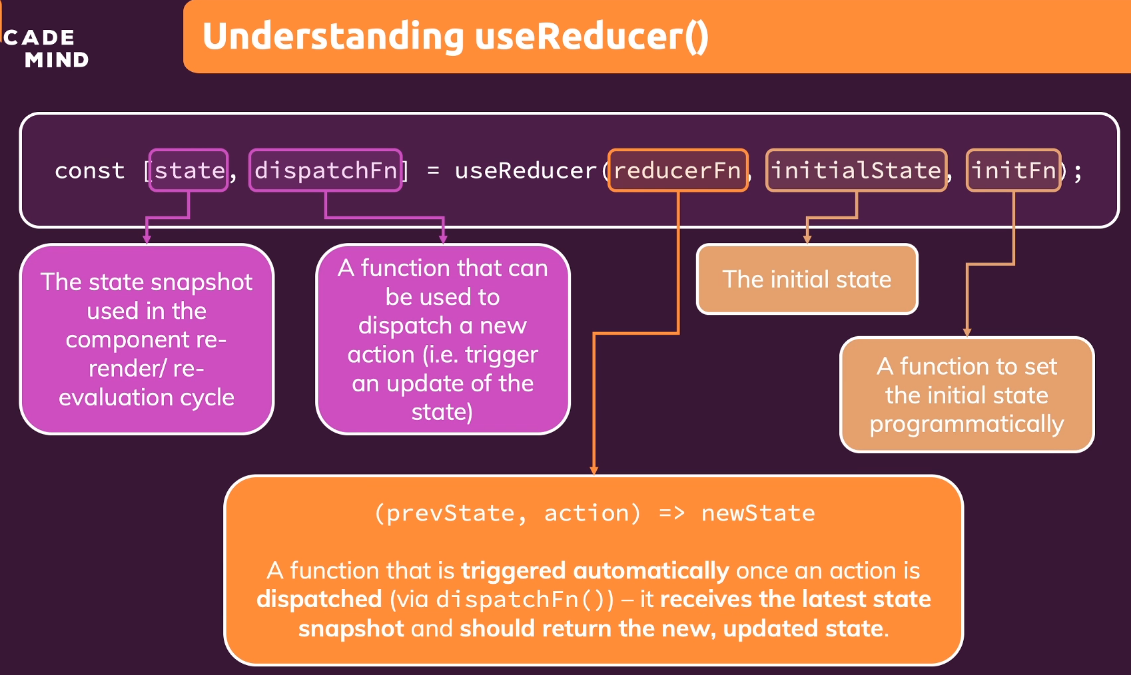
* Cleanup processes run:
  + does NOT run before the first side-effect execution
  + before every next new side-effect call begins execution
  + runs one last time before the component is removed

Part2: useReducer() --- see the summary **video 120** for a quick refresher

* You should NOT always use useReducer() because it is more complex and time consuming to setup; so for simpler state-management scenarios stick to useState()

****

* Good for the case when one state is updated based on a second/other state

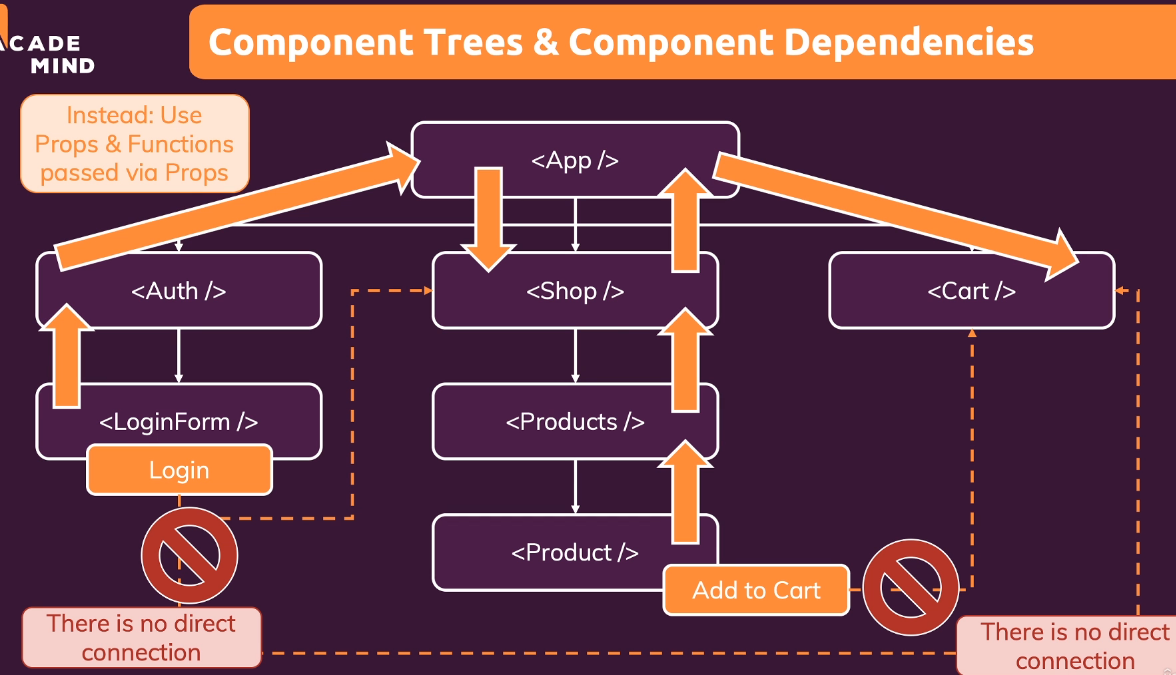
**

Comparison of *useState()* with *useReducer()*



Part3: The “Context API”

* The bigger your app gets, the more inconvenient it becomes to forward data along “prop-chains” to other components that are “far away” (see below example)



* Component-wide, behind-the-scenes state storage is what solves this problem and what the context API does.
* Two options for accessing the Context from the component “down-stream” are:
  + invoke AuthContext.Consumer()
  + import and employ the useContext() hook
* You can pass down variables, state, and also functions through Context
* Props are configurable, Context is NOT.
  + Context is for the purpose of forwarding something far down-stream with a very specific purpose

