# General Information

## What is ReactJS?

ReactJS is a javascript framework that makes the website feel like its an application. For example take Netflix in the browser, it feels like using a native application instead of a traditional website

Traditionally in web apps you click a link and wait for a new page to load.

And this is where JS comes to the rescue because JS allows us to run logic in the browser. We can manipulate the DOM with JS and this allows us to modify what the user sees without fetching a new HTML page

ReactJS is a client side Javascript library its all about building modern and reactive UI.

## Why react instead of just Javascript?

Basically we can write less code that does more and with JS we have to write every single step that needs to be created.

Ex: In a todo app > User click delete buton > crate model > create 2 buttons in the modal > add event listeners to these buttons etc. **This is an imperative approach**

Also react allows us to write code in a declarative, component focused approach

## Building Single page applications

We often create something called single page applications. React can control parts of the HTML page like a sidebar, widget but we can control the entire page with react even switch pages

A good example is Netflix where the Server sends only one HTML page thereafter react takes over and controls the UI

## React.js Alternatives

React is a lean and focused component based UI Library. Certain features like routing are added via community packages

Angular is another popular frontend framework that is also component based and it comes with more features and also uses TypeScript

Vue js is a mix between react and angular which si also component based with less features like angular but more than react

# Javascript Refresher

## Let, const, var.

Let and const are local scoped and var is global scoped.

Use let for something that will change in the future and use const for “constant” variables.

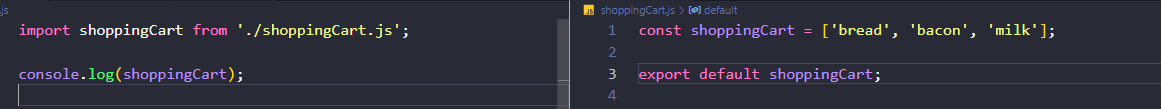
## Arrow functions

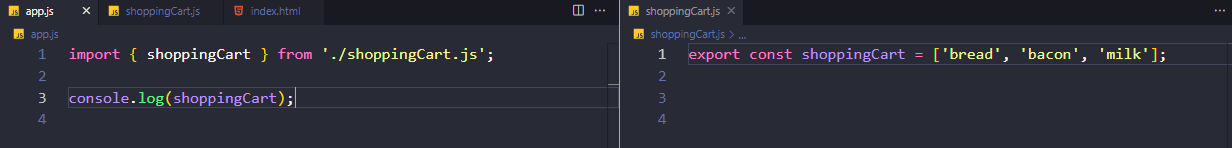
Function logMyName = (name) => console.log(name);

Arrow functions don’t have their own this keyword

## Export and imports

Below we can find default export (only one item will be exported from the exporting module)

Below are named exports



And another example of named exports  

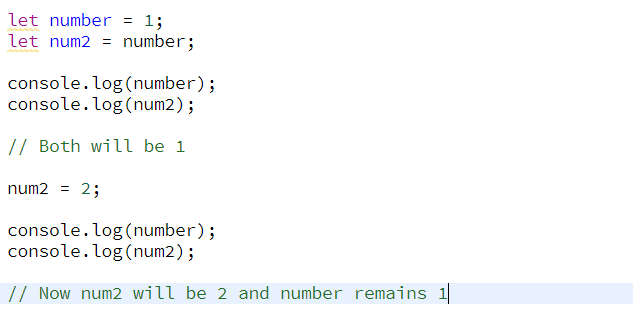

Don’t forget, in order for the modules to work we have to write type=’module’ in the script tag from the html file

## Classes Properties and Methods

Come back to this.

## Reference vs Primitive Types

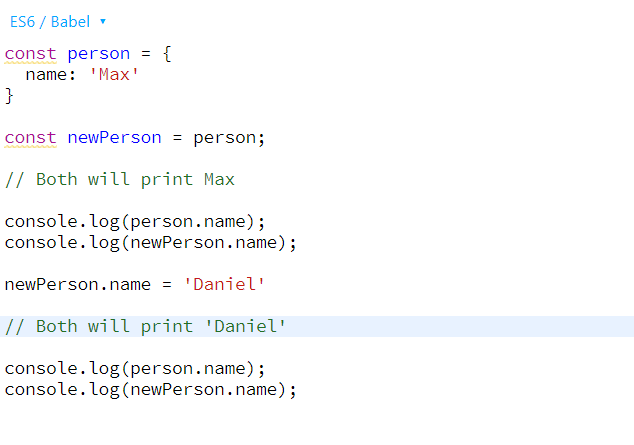
When we copy primitive types it will create a real copy of that primitive.



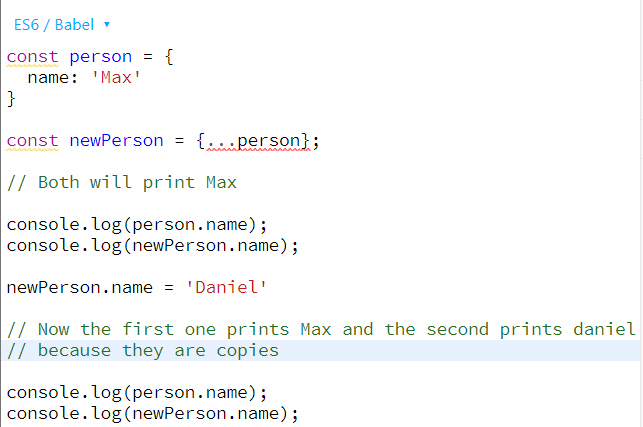
The two will be different because they are primitive types.

In the example below we use objects which are reference types. When we say newPerson = person we don’t actually make a copy we point that newPerson to the place in memory where person is. And now if we change one both will change because they POINT to the same place in memory.

This is very important in react because we might run into some bugs where we change something and that something changes everywhere so to circumvent this behaviour we want to make an actual copy of the object and we can do that using the spread operator



Example using spread operator.



# React Basics & Working with Components

## Module Introduction

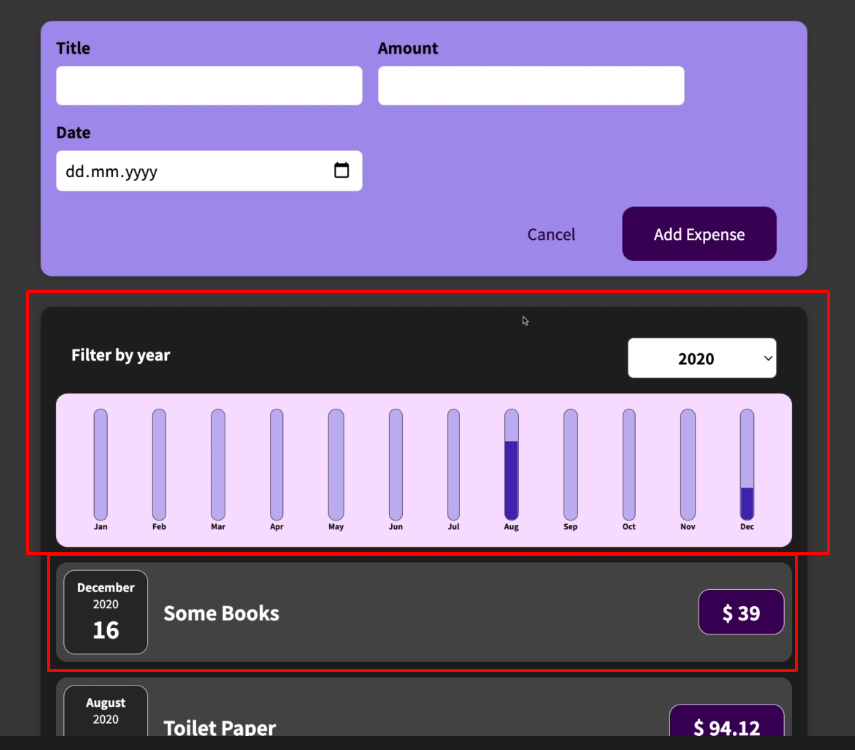
We will learn how about basics and most importantly how to build component driven user interfaces.

## What are components and why is React all about them?

React is a JavaScript library for building UI and we use react because it simplifies using UI.

React makes building complex, interactive and reactive user interfaces simpler and for this react embraces a concept called “Components” because all UI are made up of components

What is a component?



Everything here is a component and we combine them together to build a UI.

Why components?

For reusability => for DRY Principle

For Separation of Concerns => Don’t do too many things in one and the same place (function)

## React code is written in a Declarative Way

How is a component built?

In general is important to keep in mind that UI is about HTML, CSS and JS. And these components are about combining HTML CSS and JS when we work with react we combine all of these to create components.



React allows you to create re-usable and reactive components consisting of HTML , CSS and Javascript.

React uses a declarative approach to build these components which means that you will not tell react that a certain HTML element should be created and inserted in a specific place of the UI as you would be doing it with vanilla JS instead you will always define the desired end state and its react job is to figure which element of the actual webpage might be added or removed or updated.

## Creating a new React App

The easiest way to start a new react project is to use a tool called ‘create-react-app’. These are preconfigured folders with some basic react code files and most importantly a bunch of config files that helps build react files for production use. This will also give us a dev environment.

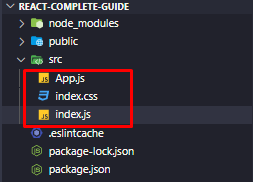


This is all the command line code we need to setup a new react app.

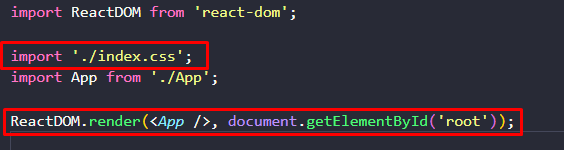
However for the project we used a simplified version of this which removes all the bloat.

## Analyzing a Standard React Project

Notice that we have 2 js and a css file.



Whenever we start the project the index.js file will the first file that will be executed.

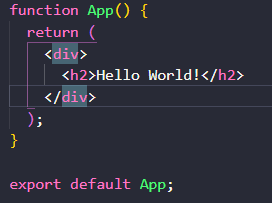


In this index.js file there is some weird code that in normal JS wouldn’t work for example importing a .css file into a JS file or the last line where we run some sort of HTML code in JS.



This weird syntax is called JSX.

Another strange thing is what is inside the App.js file



Notice this is a function that returns HTML Code that is also JSX invented by the React team that allows us to use it in our project.

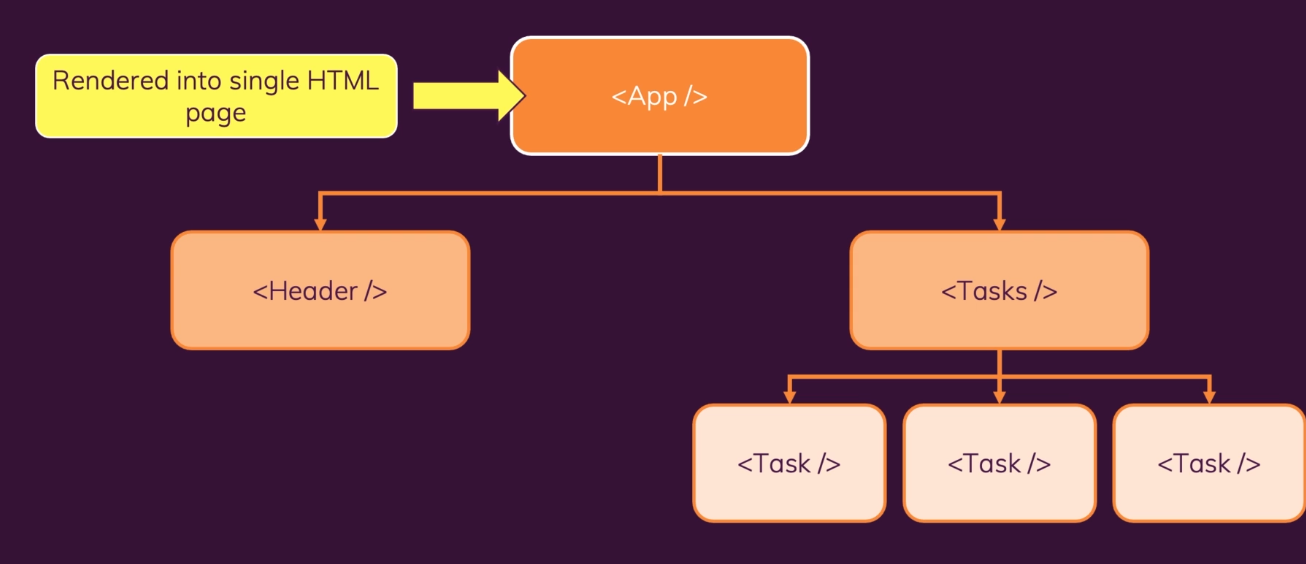
## What is JSX?

Stands for Javascript XML and is basically HTML code in Javascript. This only works because there are transformation steps behind the scenes.

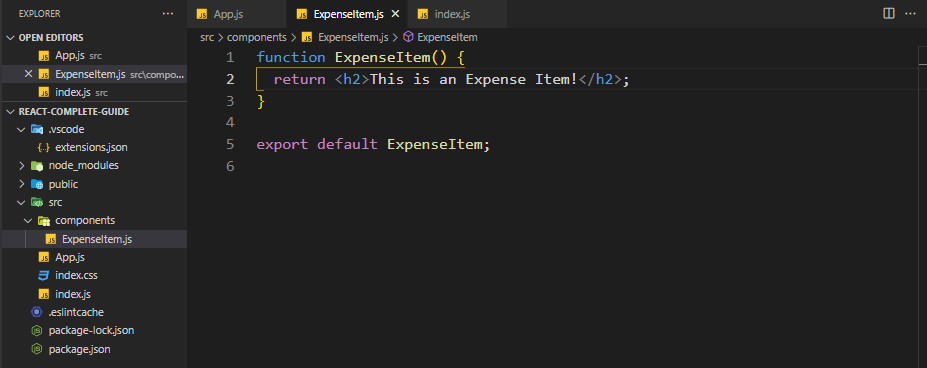
## Building our first custom component

To organize our code we will add a new folder in our src folder that will host all our components (might be hundreds in a big project).

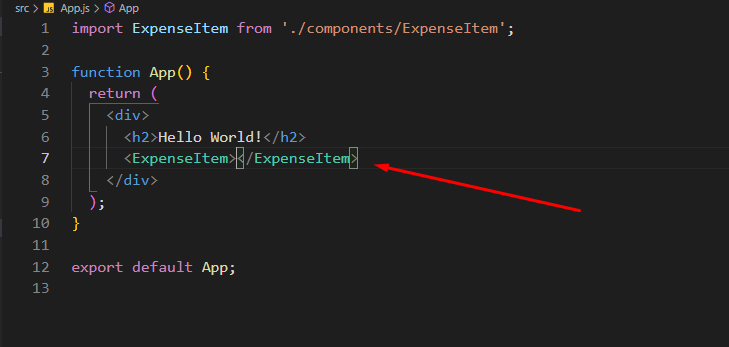
Ultimately with react we create a component tree



The convention of naming here is camelCase.



Now we are not going to import this in the index.js file because we only import our app.js file there but instead import it in the App.js file and insert it in the code there.

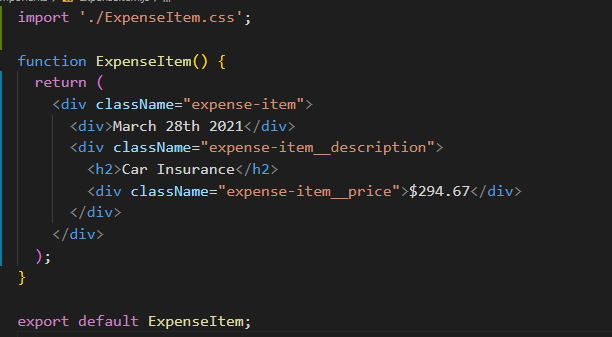


In React we are not allowed to have 2 root divs. Take the example below:



We cannot have another dive next to the main div but there are some workarounds in which we will dive in later.

## Adding Basic CSS Styling

For this we are going to use CSS which we will put next to our JS file. The css we will get from the course.

After that we have to import the CSS file into our component and instead of adding class to the JSX we add className.

We can also use SASS but for now I’m going to use CSS

## Outputting Dynamic Data and Working with Expressions in JSX

This being Javascript we can use normal variables and functions to output some sort of dynamic data for now. Keep in mind that for date we have to use toISOString() method because otherwise it will return an object and react cannot display that

## Passing Data via ‘props’

## In React we can make our components reusable by adding parameters and a concept called props. So in react our custom components can have attributes

## In every react component we will have a parameter called props which will be an object that data is stored in

## The data for our expenseItem components will be in the App.js file

## Here we have some dummy data from the course and above is the way we write the parameters that our newly displayed component will use

## Ex. title={expense[0].title} etc.

## Now this is only the first part of the job the next is to make sure that our component accepts this data’

## In the props objects we get key, value pairs.

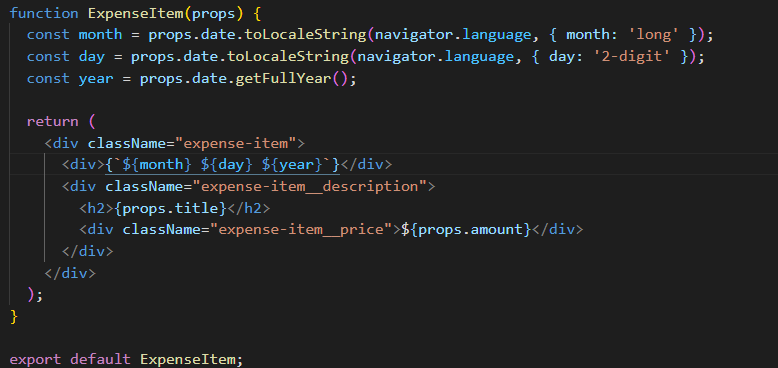
## The key which you access in on your prop object has to have the same name as you picked for your new component attribute.

## And this is how you share date between react components.

## You will use props all the time, it allows you to make your components reusable and it allows to pass date from another component to this component

## Adding “normal” Javascript logic to our components

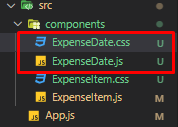
It is a good practice to separate the logic from the return function so we can format the date inside our function and in the return statement just enter what we formatted. I’ve used toLocaleString instead of internationalization API here because that’s what the teacher used in the course.



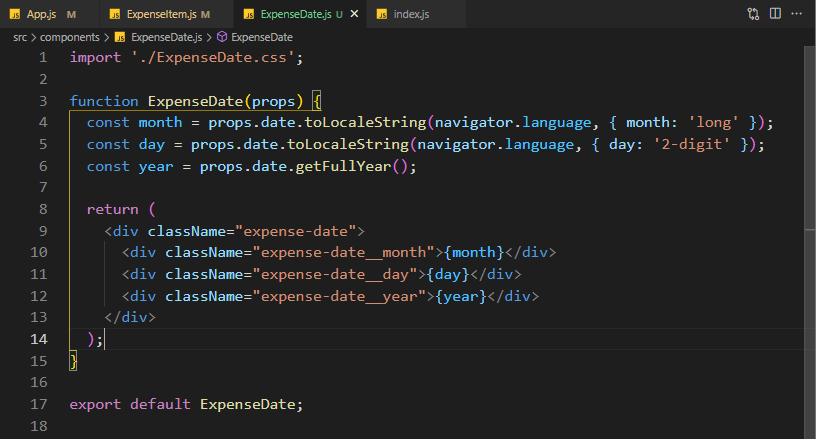
## Splitting Components into Multiple components

You will notice that in any project, your components will become bigger and bigger that is why react has this component concept. There is not a hard rule when to add a new component but we can argue in our expense component that it is getting a bit too big so the calendar item might be considered as a separate component.

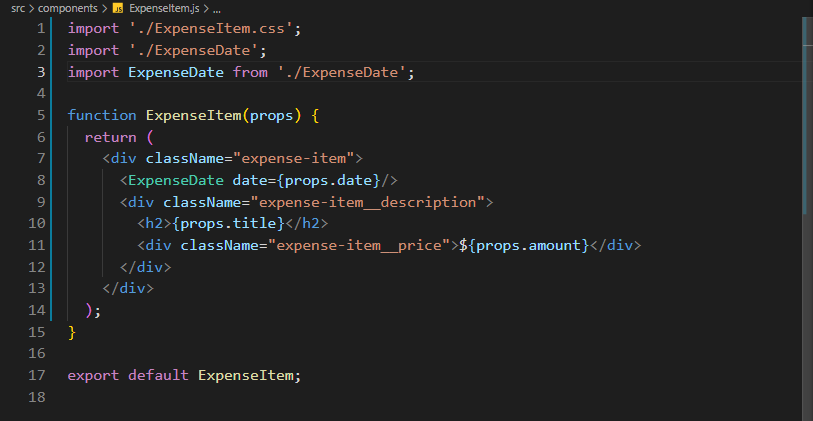
So like we did before we create a new JS and CSS File



The CSS is provided by teacher and the js file we will make ourselves



And now in the ExpenseItem where the date is needed we just write the component with a simple closing tag (we can do this if there is nothing between the tags)



This can get a bit confusing because we are funneling data trough multiple components.

First we get the data from App.js which we pass to ExpenseItem.js which we pass to ExpenseDate.js

It is totally fine to pass data between multiple components using props that’s how props work

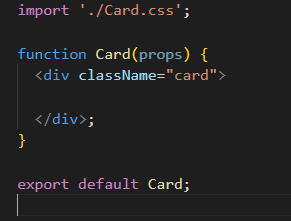
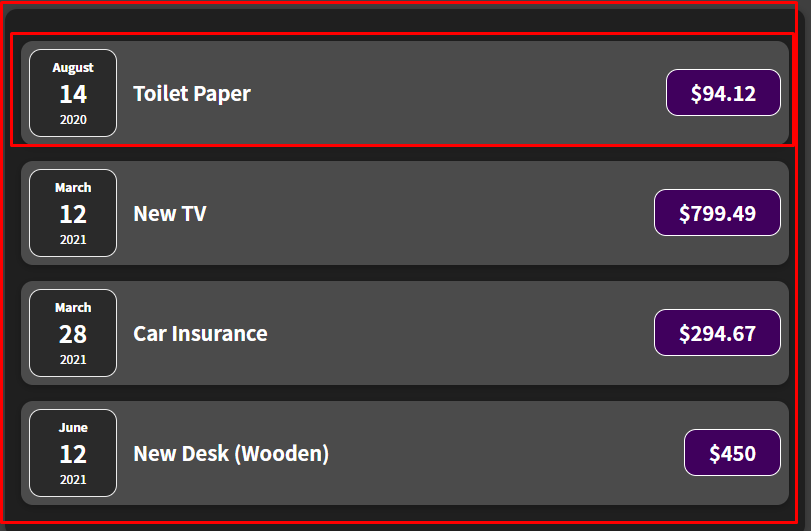
## The concept of “Composition” (“children props”)

The concept of composition is building an UI from small building blocks like we did above.

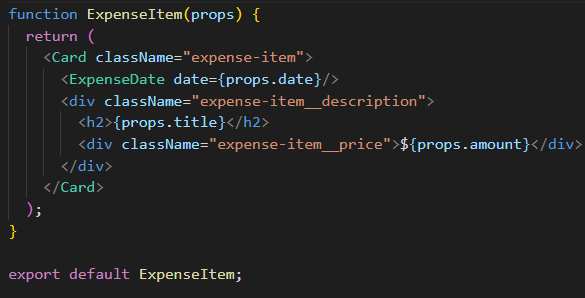
Expenses used the ExpenseItem component which uses the ExpenseDate component

But there is one more interesting aspect of composition which we haven’t seen yet.

What if we want to create a component that just serves as a shell around any kind of other content. Like a card around our ExpenseList or ExpenseItem



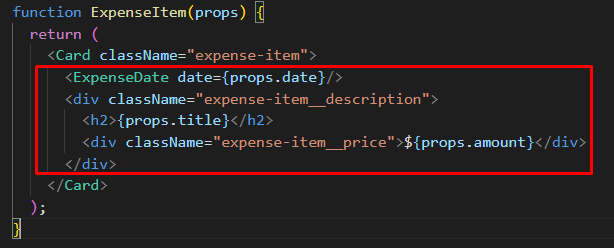
This is our card component which is a shell for now so the next logical step would be to replace the outer divs with the card component in the ExpenseItem and Expenses



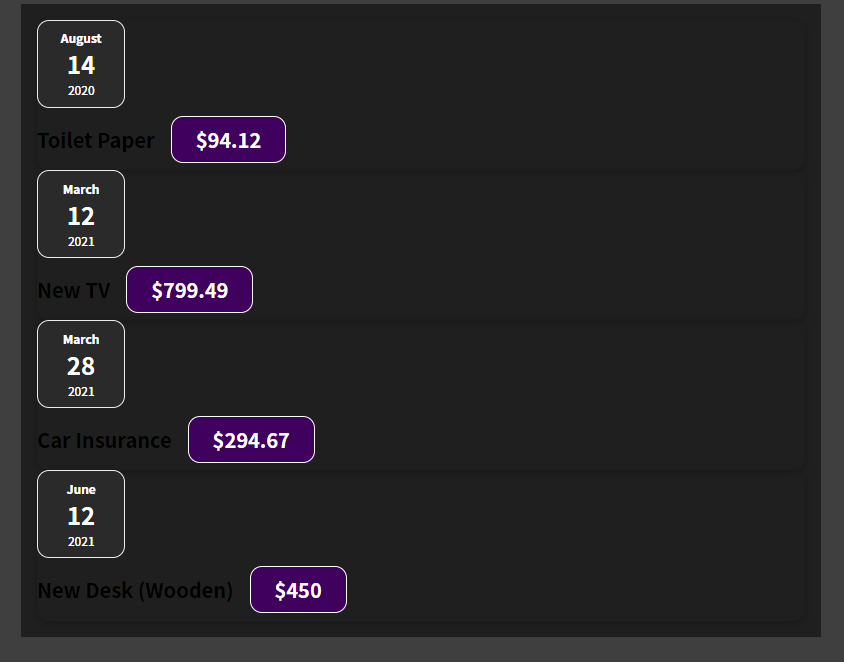
This wouldn’t work like this for custom HTML elements like our card the solution is to use props.

We will use a special prop in react which every component receives and that is props.children and the value of this children obj will be the content between the opening and closing tag of your custom component.

Below is the value of props.children inide our card

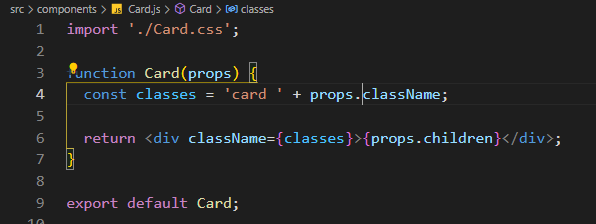


Adding this will somewhat fix our code but it will be still weird



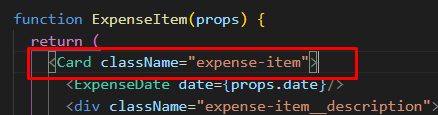
This is because we extracted some CSS but not all, our default HTML elements support classes out of the box but our custom components only support what we tell them to support.

So if you want to make sure that a classname can be set and used on your card component we have to make some tweaks.



Tweaks above explained :

* We have a variable classes that will have the ‘card ‘ class by default followed by the class passed in where the card is used

 this is props.className so in the end it will be card expense-item

* In the second part of t he code we put props.children between the divs. As we know children is a default keyword which every components has and like mentioned above props.children will be the content between the tags.

Why do all this?

In this example not much but we still managed to extract some code duplication (the css code we had both on ExpenseItem and Expenses) into our card class. Also we managed to transform 2 divs into our card class.

In larger projects where we encounter more complex components like modals this will come in very handy

## A Closer look at JSX

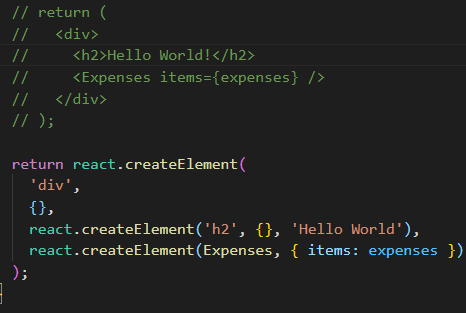
In older react projects we had to use import react from ‘react’ in every file. Now we don’t have to anymore it gets done automatically

Below is what is happening under the hood of JSX

First argument is element/custom element name.

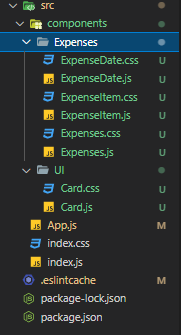
Second argument contains any attributes

Third argument is can be as long as we want and its whats between the opening an closing tags of our first argument element.



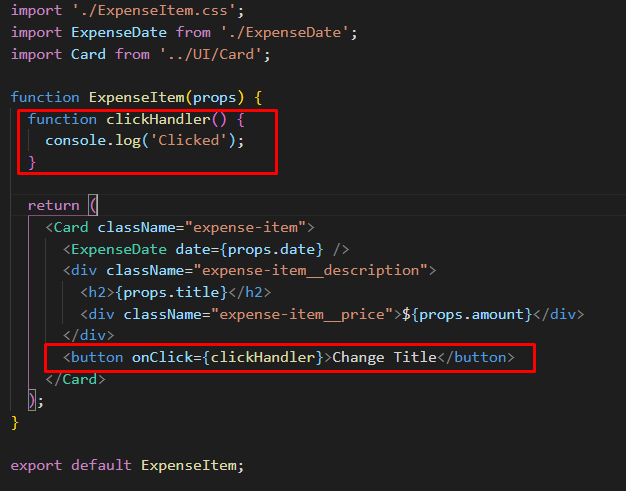
## Organizing Component Files

As our project grows we want to organize them into subfolders and not just throw everything in the components folder



# React State & Working with Events

## Listening to Events and Working with Event Handlers

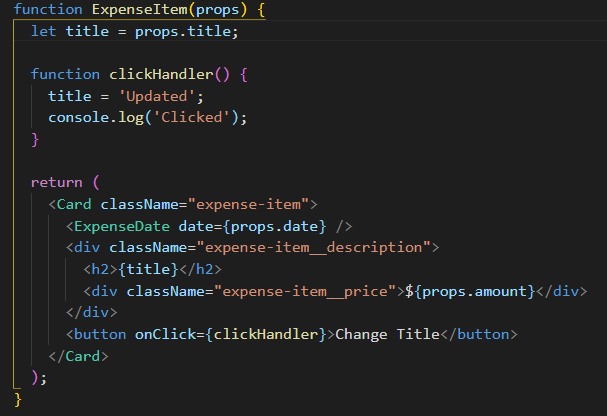


Adding event listeners is fairly simple just add onClick on the button. Since the button element just inherits from HTML Element we can use all sorts of properties and methods on it and all of this are available in react as well

It is a good convetion to add Handler at the end of an event handler.

## How component functions are executed

We want to change the title when the button is clicked

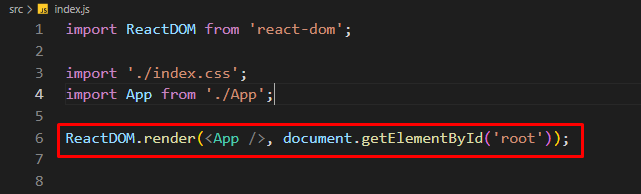


Using this example won’t work because this is not how react works. We have to dive in how react parses the JSX code and how it brings it on to the screen to know this.

For now we have to keep in mind that our component is a function, the only special thing about that function is that it returns JSX since it is a function something/someone has to call it and notice we never called components we just used these function as HTML elements.

By writing out our components react will executed all the JSX code and whenever it runs into another component it calls that component as well.

This happens only once in our CURRENT application and it happens once when the following method is ran



Now we need someway of telling react that a component changed and we need that component rerendered and this is when state comes into play

## Working with “State”

State is not a react specific concept but is a key concept in react

In the scenario above we want to use state because when the click handler is clicked the title of the clicked element should change thus implying a state change.

To tell react then when a component changes it has to rerender again we have to import something from the react library.

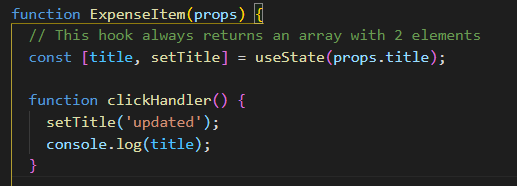
import { useState } from 'react';

This is a so called React Hook, one of the most important ones. They can be recognized by having use before them and they must be used inside a React Component Function but not inside a nested fuction instead directly inside the function (there is one exception that we will talk about later)

The useState hook always returns an array of 2 elements and we can use modern JS destructuring to assign variables to them.

The first element will be the value from the props and the second one will be a function that we will use to change the first value. We can name them everything we want but I used setTitle because it is self expalantory.

Now in our clickHandler we can call this setTitle function and pass in the what we want to change the title to as the first argument of the function.



So why do we do it this way ?

**Because calling this function doesn’t just simply change the value to some variable but instead it is a special variable to begin with which is managed by React somewhere in memory and when we call this state updating function, this special variable will not just receive a new value but the component function in which we called this function will be executed again. And finally it will re render the component with the changes made.**

One more thing to note, notice that we have a console.log(title) after the setTitle function. This will still log the initial value of the title because setTitle doesn’t immediately change the value but instead schedules it for a change.

If you have data that might change and changes to that data should be reflected in the UI then you need State because regular variables will not the trick

## A closer look at useState hook

First of all the state is on a per component instance basis so we have separate state even when we create a component more than once.

Thus far, we update our state **upon user events** (e.g. upon a click).

That's very common but not required for state updates! **You can update states for whatever reason you may have**.

Later in the course, we'll see Http requests that complete (where we then want to update the state based on the Http response we got back) but you could also be updating state because a timer (set with setTimeout()) expired for example.

## Adding Form Inputs

Now we that we know about state and event listeners we will add a form for user input.



This is how our form will look in the end and notice that we have an onChange handler on the title input field which updates everytime the user changes something in the Title field. We can access this value by using the event paramenter which is there by default on every event handler

## Working with multiple states

Now in the end we want to retain the user input even if the component is re executed and for that we can use state again.

Note: The component will always update when updating the state.

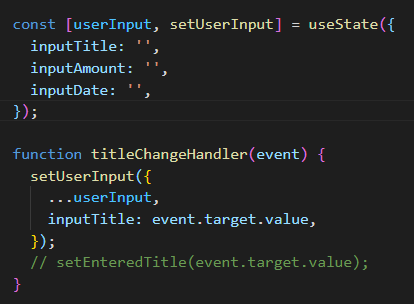
You can have multiple states per component and in this example we are going to use useState(‘’) with an empty string because by default it will be an empty string, in the example before we used useState(prop.title) which was the initial state of the prop



This is the example of using multiple states (note this is not the final code yet just an example with what we know until now);

## Using one state instead – What’s better?

Instead of using 3 separate states we can use an object as single state where each property is a the value

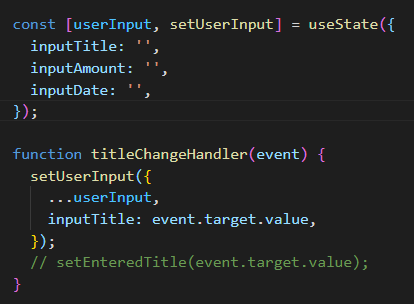


To note that every time we set the user input for one of the inputs we have to copy the other properties with destructuring otherwise it will return an object only with one property

It is up to you which one to use but I prefer using multiple states instead of a single one

## Updating state that depends on a previous one

In the example above when using multiple properties in a single state we might ran into some issues



The way how the handler function is set up we depend on the previous state.

Whenever you update state that depends on the previous state we should use the following method

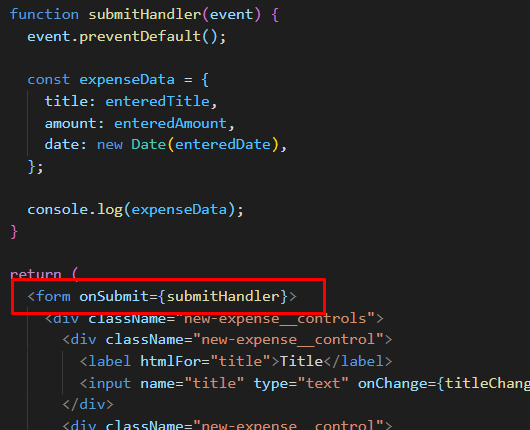


**Now why should we do it like this instead of the previous example? In many cases both will work fine but keep in mind that react schedules state updates and therefore if theoretically if schedule a lot of updates at the same time you could be depending on an outdated state. So by using this method react will guarantee that it is using the latest state update keeping all the scheduled ones in mind**

## Handling From Submission

We want to display the entered data as a new ExpenseItem and for that we need the data when the user clicks on the submit button.

For this we can use the onSubmit method on the form and whenever the form is submitted that method will execute (don’t forget to prevent the default refresh behaviour when clicking on the submit button , we don’t want this because we will be handling it with javascript not as an HTTP request)



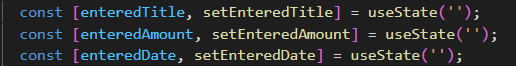
## Adding a Two-Way Binding

How can we clear those inputs? That is part of the reason we use state and not global variables outside the component to store the inputs.

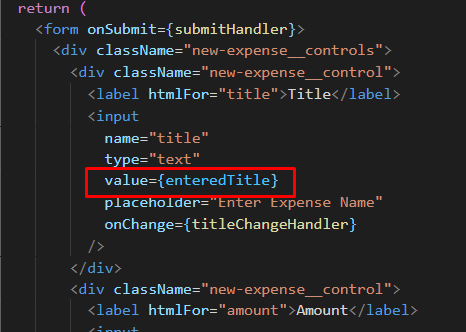
Now we can implement something called a 2 way Binding which simply means for input we don’t just listen to changes but we can pass a new value into the input to reset or change the input programmatically

First we are going to place the value attribute on the input which will be equal to an empty string when the component first renders because remember the initial state of the value is an empty string

Initial state set to empty string in the begining



And here we add the value attribute to the input tag



Now when the form submits we will use the set methods to set the state back to an empty string thus provoking a rerender of the component and setting in html to an empty string as well.

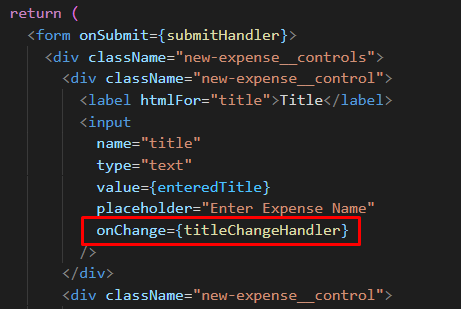


## Child-to-Parent Component Communication (bottom up)

Now having the data from the form is nice but we don’t need in the form, we need in the App.js component and ultimately our goal is to add this new expense to the expenses array from App.js and enrich it by adding an id to it.

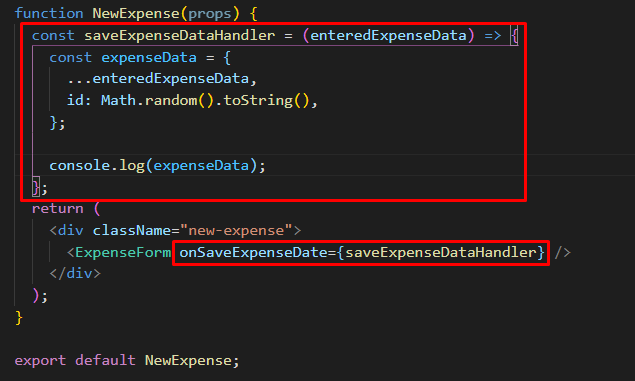
Now we know how to pass data to children by using props but how do we do it in the other way? We already used this but without knowing it.

In expenseForm we are listening to user input with the following function



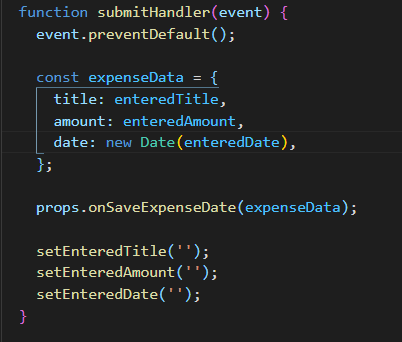
Whenever the user types then titleChangeHandler executes and there we get the default Event object which the browser gives us now we can think about the input element as a component as well, it is not our a component but a prebuilt one provided us by react and translated to the inputDOM element. We also set some props on that component like name, type and onChange prop. This onChange prop is not that special, is just a prop that wants a function as a value then internally it will be set on the DOM as an event listener. Now this is a pattern that we can replicate on our custom components, we can create our own event props and we can expect function as values, this allows us to pass functions from a parent component to a child component then call that function inside the child component with the data from the child component. And this is how we communicate from child to parent.

Long story short : To communicate up we can pass a custom prop to our parent element which accepts a function and that functions arguments will be the data we need from the child component. Then in the child component we call that function with the data.



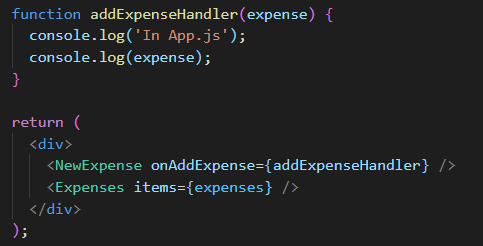
This is one of the parent components that has the prop onSaveExpenseData which passes on the function saveExpenseDataHandler, now this props function expects a parameter called enteredExpenseData. It will copy this enteredExpense data into a new object and add an ID on top of that.

Below we call that function when the form submits thus passing in the data

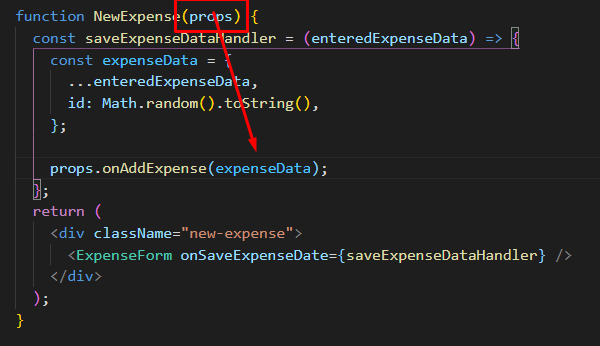


Now the last step is to get the data in the App parent component.

First we set the function prop that accepts the data as its argument



And now we use that prop in NewExpense.js

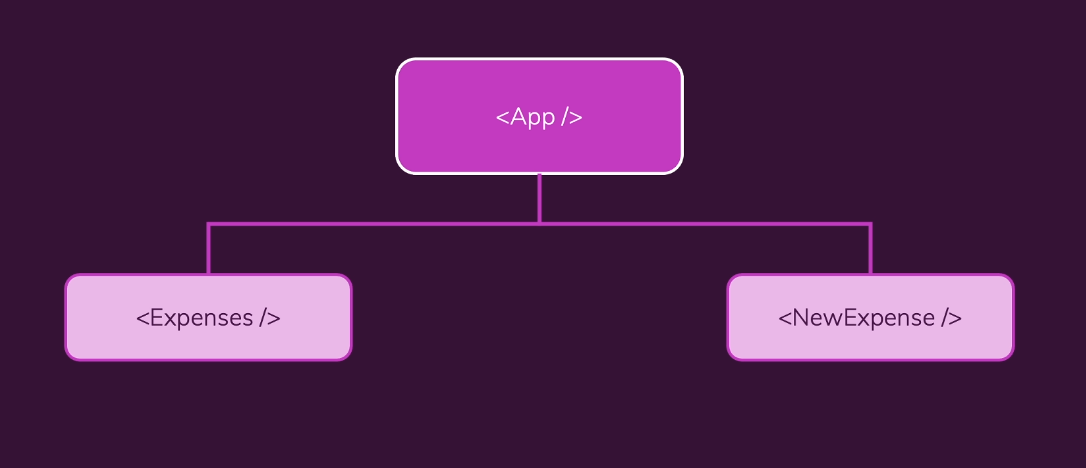


For now we just want to see if the data arrives in the parent app and later we will take care of updating the expenses list

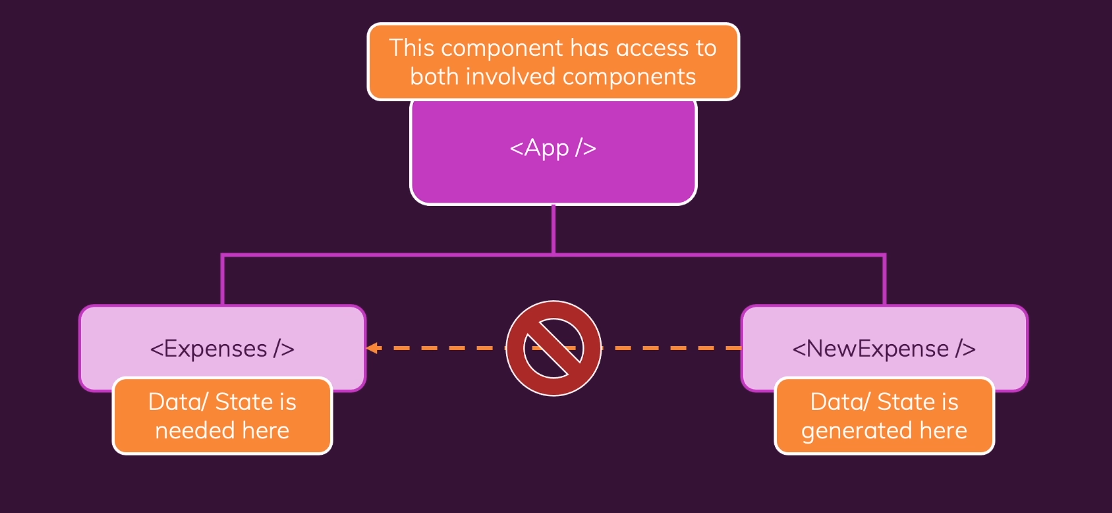
## Lifting the state up

In the last lecture we learned about moving data from a child to a parent component by utilizing props to receive a function from the parent component.

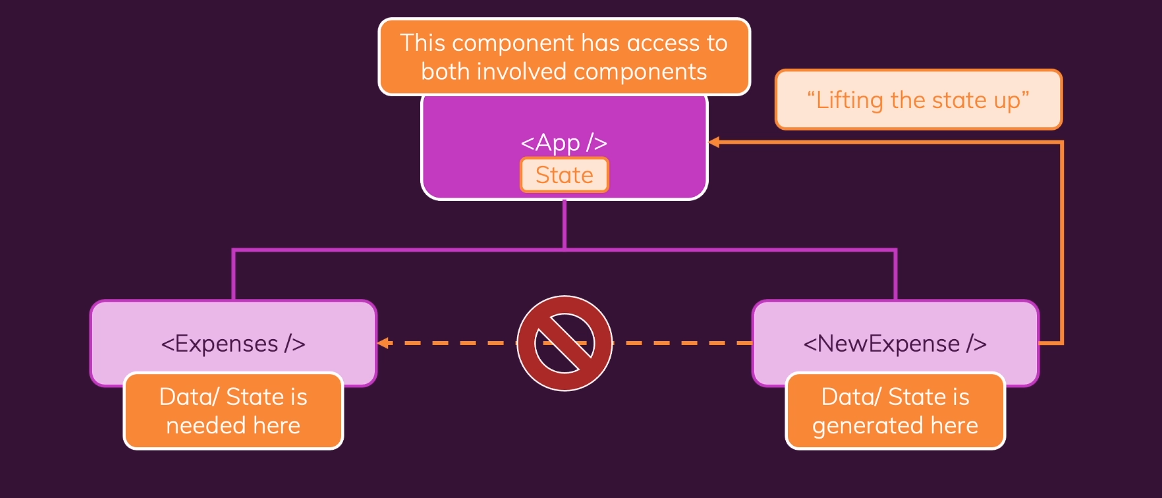
There is a concept called lifting state up. Consider the next tree



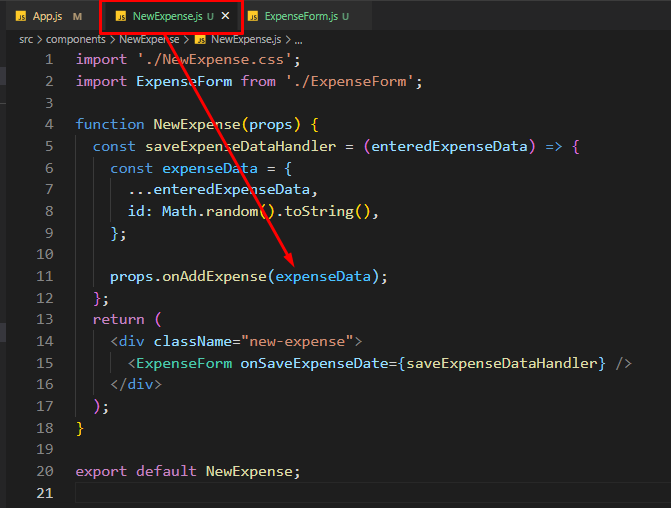
In this example the NewExpense component is a component that generates some data/state it is common that you generate data in a component but you might not need that data in that component (we don’t need the form data in our form, we need it in our expenses)



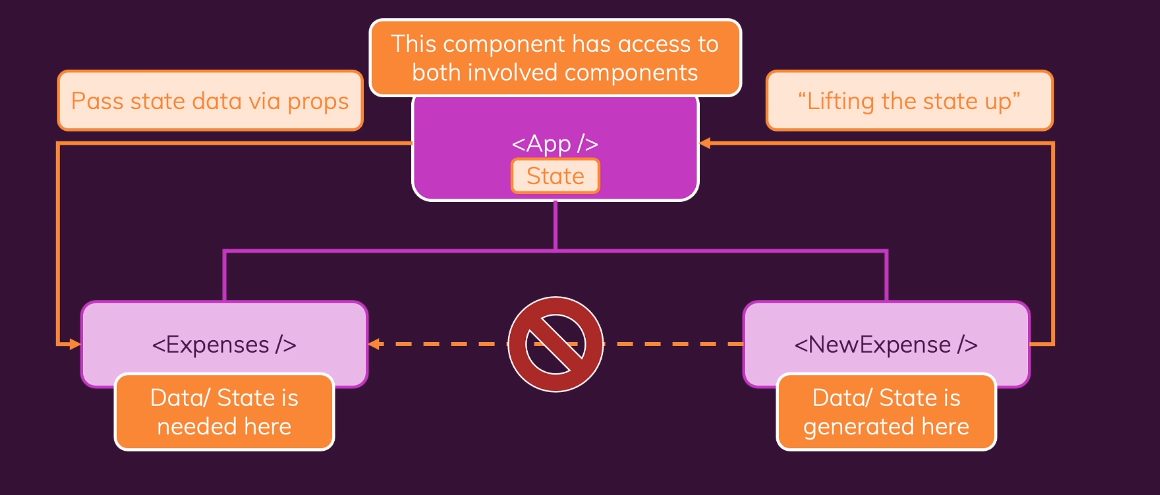
We want to utilize the closest common parent that has access to both components, in our case the App component. Now we can store our state in our App component by lifting our state up



We are already lifting our state up by passing the state generated in the NewExpense into our App component. We are doing this by utilizing props, by calling the function we receive on the onAddExpense prop



**Calling the function alone doesn’t lift the state but by calling it while passing in the state as an argument does.**



Usually we don’t have to lift the data up until the root component but until we have a component that has both access to the component that generates data and the one that needs data

## Controlled vs Uncontrolled Components & Stateless vs Stateful Components

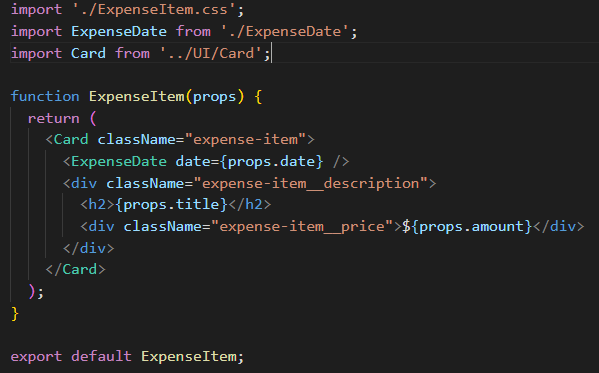
Whenever when we use two way binding we create a controlled component meaning a value used in a child component is passed to the parent component trough props and it is received from the parent component **so the logic is handled in the parent component.**

Stateless vs Stateful components.

In any react apps you will have a couple of components that manage some states then you will have other components that do not manage any states. Components that are stateful means they have state and the other ones, the stateless ones (presentational ones) only display data.

A good example would be the ExpenseItem (if we removed the button which was only there for demo purposes)

Stateless component



# Rendering Lists & Conditional Content

## Module Introduction

Up until now we learned how to create React components, how to pass data between them and now is the time to work with Real Dynamic Content.

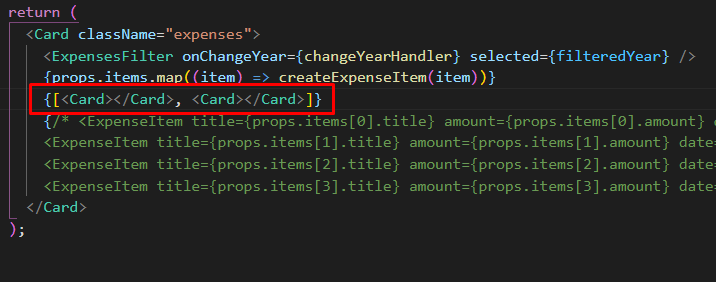
We will learn:

* How to output dynamic list of content
* Rendering Content under certain conditions

## Rendering Lists of Data

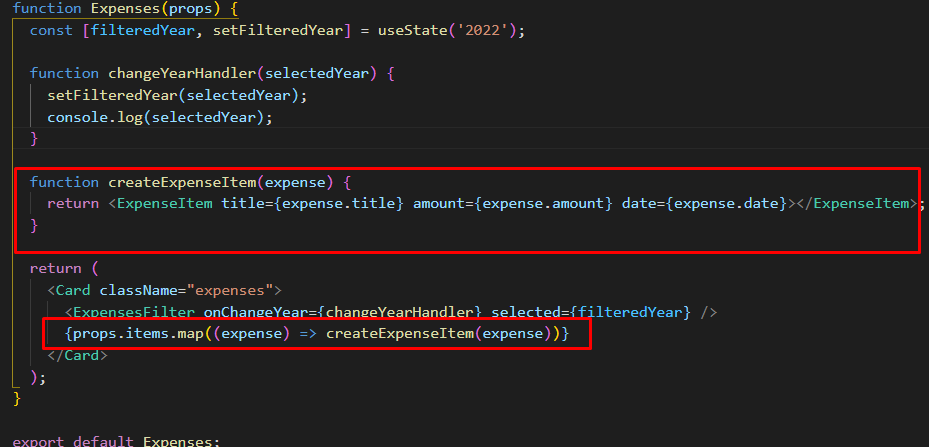
We are going to utilize the build in map method to programmatically display all the Expense Items from our expenses array.

The rendering happens in Expenses component so we have to pass the data via prop to it which we have already done.



Notice the highlighted syntax, if we have an array of JSX syntax, react will just renders those components that’s why we use map because it returns an array as well.

And this is how I did it



## Using Stateful Lists

Now that we rendered our lists programmatically how can we update this whenever a new expense is added ? Using stateful lists

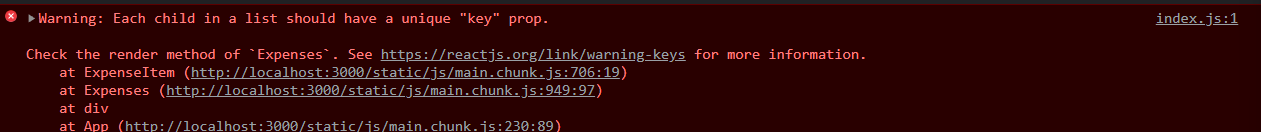
For this we want to refactor the code in App.js where the expenses are and make them stateful.



We put the expenses outside the app to unclutter the App component then we set the initial state of the expenses to the dummy data.

After that we use the data that we got from our form (trough state lifting) and used the setExpenses functions to update the expenses whenever the form is submitted.

Notice that we used the special prevState argument which we will use all the time when our state depends on a previous state.

We are almost done but we got a warning in our dev console regarding some unique prop key

## Understanding Keys

So why are we getting this key warning?

React has a special concept when it comes to rendering lists of data a concept which exists that react is able to render such lists without performance loses or bugs.

I’m going to explain what happens when we add a new item now.

Whenever we submit the form react renders the newly added item as the last item in our list of expenses and updates all items and replaces their content such that it matches the order of items in the array, this is not good for performance. But this is not the only one problem that can occur, imagine if each of these items had state changing in them then the old first item will be overwritten with the new first item hence any state changes we had in there would be lost.

Why does react behave like this? Because it has no other way.

React doesn’t differentiate items in our list so we need a way of telling react this and we do that by going to the map method and add a special prop called the key prop. By adding this key prop we can help react identify these individual items.



Like this we specify the key prop and set it to the expense.id (which we generate everytime we create a new prop) if we lack an id we could use the second parameter of the map method which is the index but that is not recommended.

So after adding the key react is able to identify where to put the item and how long the array thus updating the list in a more efficient way

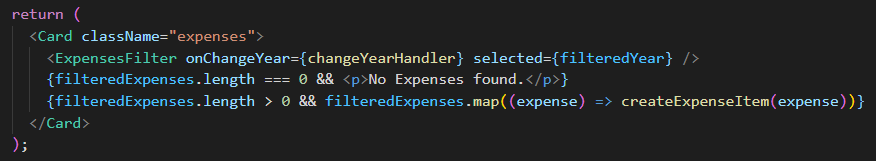
Always add keys when displaying such lists.

## Outputting Conditional Content

Our filter is working now but we have years where we have no data and we might want to display a message when there is no data.

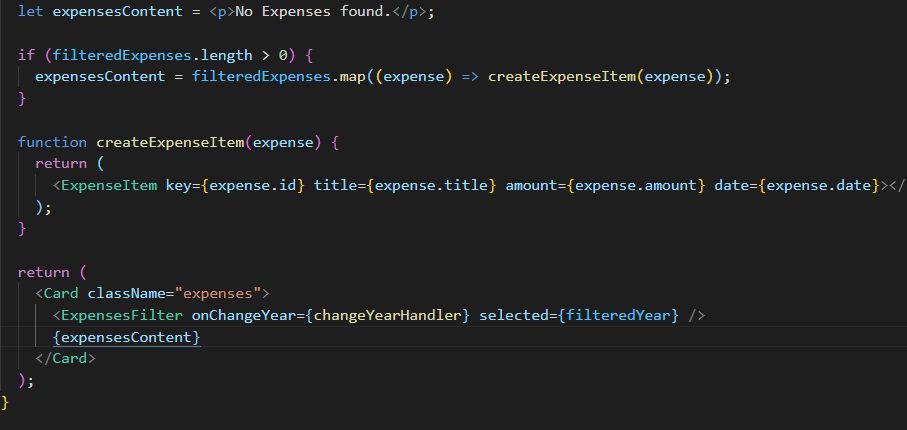
Conditional content is about rendering different content under different conditions

Method 1



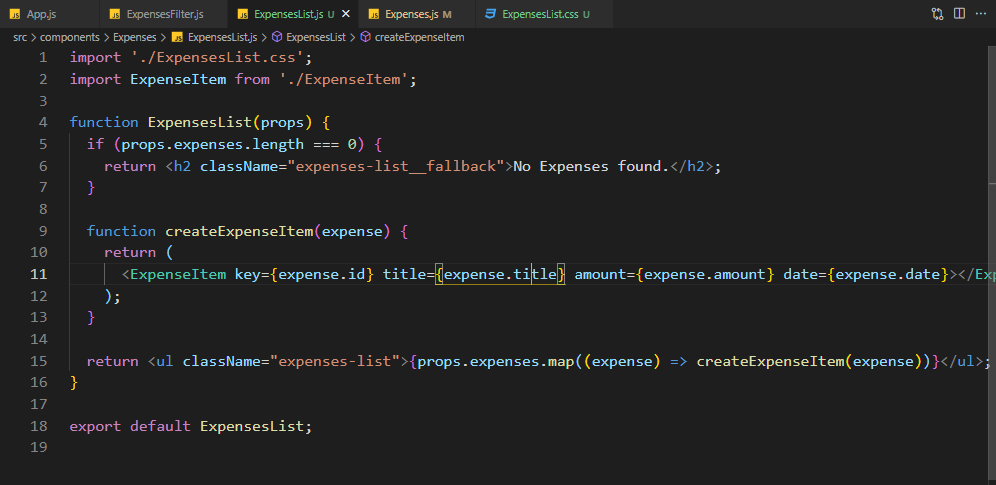
Here we used shortcircuiting of the and operator.

Method 2



## Adding Conditional Return Statements

Now we are going to restructure the expenses component by moving the list logic to another component



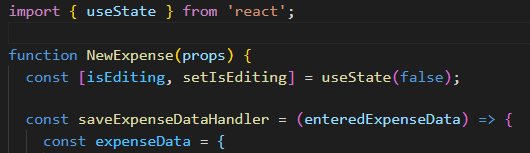
The component will be called ExpensesList and we did some refactoring to make the code leaner. Now we have 2 return statemtents

## Hiding / Showing buttons in React

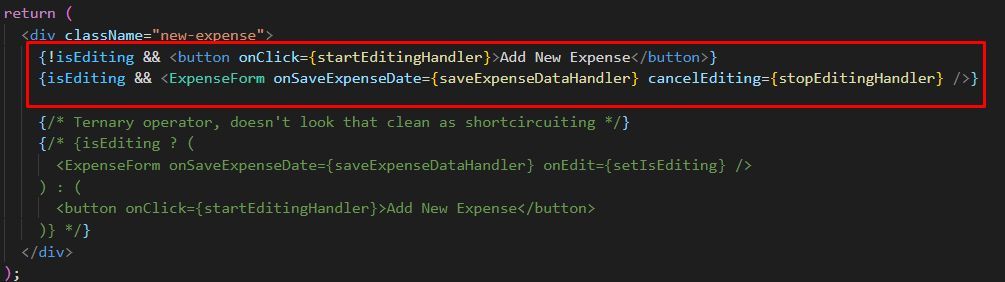
We had a task where we had to hide the form and only when we click on a button we show it. Now I approached this the vanilla way and started doing stuff in CSS and by the time I got to using old school event listeners I realized this is not the way to go.

We have react lets leverage its powers.

The first thing we need is a state which lets react know if we are editing or not and by default we are going to set it to false. And we need this state in the component where we display the form aka NewExpense.js



And now we implement the logic to display the form only when isEditing is true

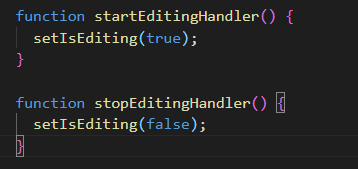


So if we are not editing then only the button Add new expense will be shown otherwise the form will be show.

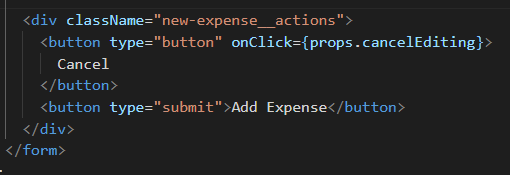
Also we add a handler to this button and whenever it is clicked it will set isEditing to true.

When the form is submitted or the cancel button is clicked then isEditing is set to false

The two handler function



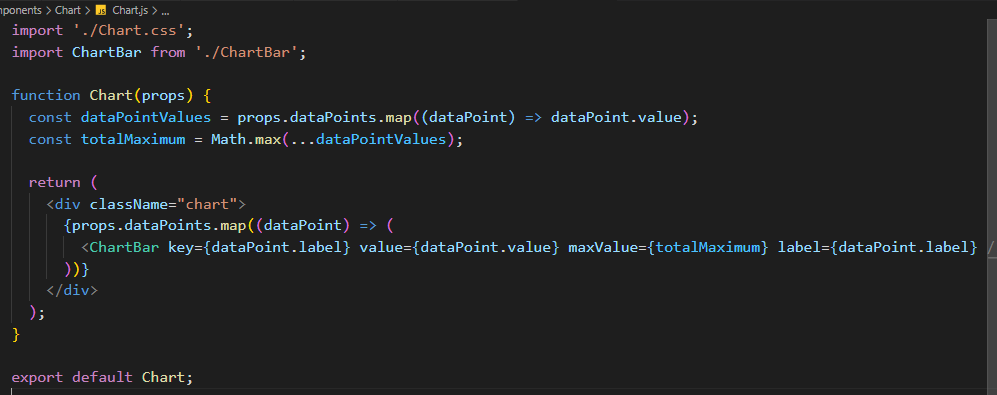
The stopEditinghandler will be passed to the ExpenseForm as a prop so we can call it when cancel button is clicked



## Adding a Chart

The next step in our project is to add the chart, for this we are going to create thre new separate components, Chart and ChartBar, ExpenseChart

The chart will be the “container” and for each month we will have a chart bar



The chart gets its data from the Expenses component and it only needs the filtered expenses.

In the Chart we need the maximum expense value in order to display each monts total in relation to the maximum value.

Below we will find the ChartBar component. Here if the maximum value is greater than 0 (meaning that there is an expense in the selected year then the bar will be filled with CSS according to the percentage we calculate.

Notice how we implemented CSS styles (this is not an inline style). First we open curly brace to indicate that we want to use Javascript and then we open another set of curly braces to create a JS Object where we specify the CSS code as a key value pair



Next we create another component which we will use in Expenses.js (not necessary)

Here we will have an object containing all the months with the value 0.

In order to increment the value of this chart we will loop over it and we can use the expenses.date.getMonth() method to increment the month from the chart since getMonth is 0 base and so is the array.



# Styling React Components

## Introduction

Building Webapps with react is not just about composing components together but making sure that the logic also functions correctly. Lets not forget that react is also about styling components

Styling is important part of building components and there are different tehniques for setting styles dynamically.

We will learn about:

-Conditional & Dynamic Styles

- Scoping CSS files

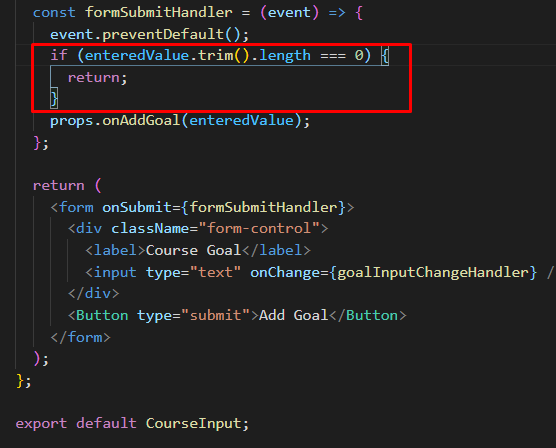
- Styled Components

- CSS Modules

## Setting Dynamic Inline Styles

We have to download the starter files from the course and run NPM install

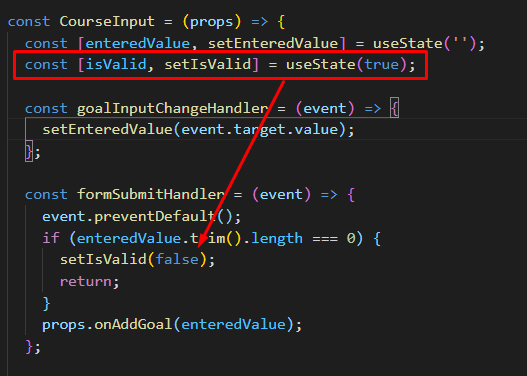
After setting up our first task is not to let the user to submit an empty goal



For that we used the condition above but we want to let the user know that he didn’t enter anything and for that we will dynamically change some inline styles.

Setting inline styles is not the best option because they have the highest priority but we will show how to do it nonetheless.

For this to work we are going to create a new state that checks if form is valid



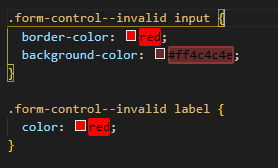


Then we check with a conditional if its valid and apply it inline

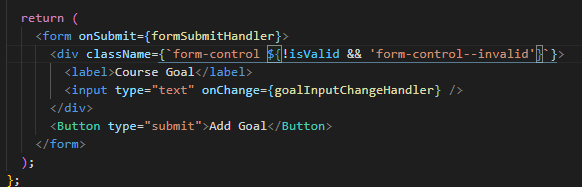
## Setting CSS Styles Dynamically

We are going to apply dynamic styles by adding classes to the form if the value is invalid

For this we are going back to the classic way of adding classes in CSS



If the form is invalid then the following rules will apply to input and label



Then all we have to do is to dynamically change the classNames with javascript

## Introducing styled components

Currently we are using regular CSS files which we import into our components but these are not scoped to that element only. Meaning if we import into a component a CSS file that for example has a class called .container{} and another element in the DOM has also the class container they might interfere. This is not necessarily a problem if we take care in our CSS code but we can scope CSS files to individual components but in larger projects it might happen that a name is used twice.

There are 2 approaches to fix this:

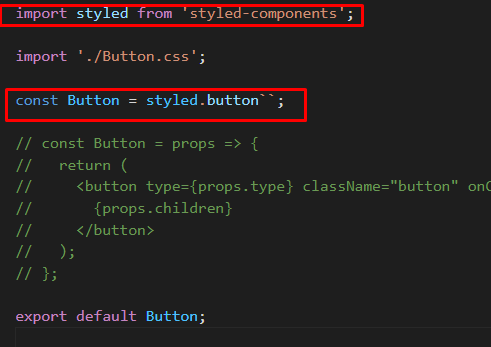
1. Use a package called Styled Components. This is a package that helps you build components that have styles only attached to them and not other components

<https://styled-components.com/>

Install the package

Npm install styled-components –save

This is how we used but before I’m going to explain something

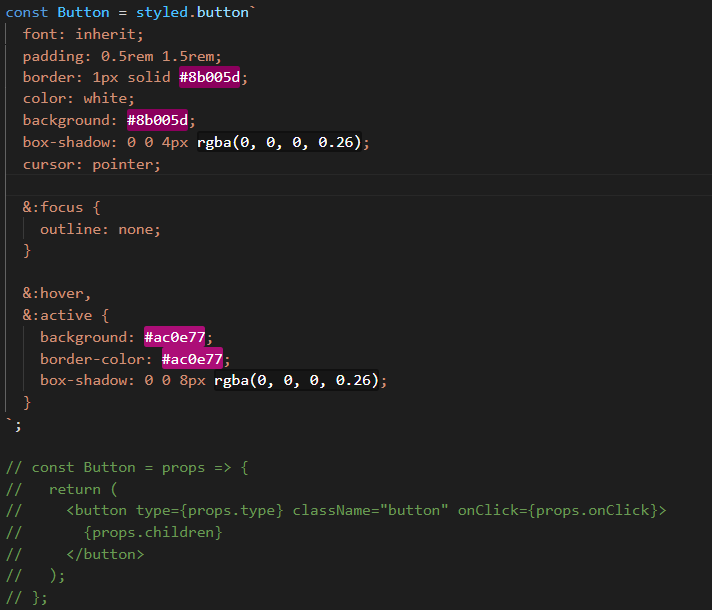


Notice the weird syntax on the Button function. I call this a method because styled.button is a method but is not called with () but instead with `` (tact template literal syntax – this is js code). For now all we need to know is that what we pass between those backticks will end up in the button method and the button method will return a new button component.

We can use style.div, style.p, style.h2 etc

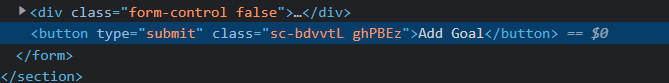
The cool thing about backticks is that we can write a multiline syntax without no problem so we are going to copy the button css in between those backticks

One single thing we did is remove .button selector because we do not need that since it will automatically apply to the component it is used on and for pseudo class selectors we just use the & symbol just like in SASS



Notice that the code we used before had some props on it. The className will be automatically applied to the button by styled.button, the onClick is handled in the parent app and props.children happen automatically in the background.

And this is how it looks in developer tools, notice how the classes are unique

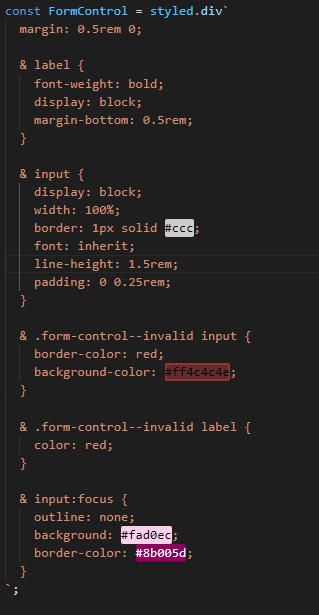


## Styled Components & Dynamic Props

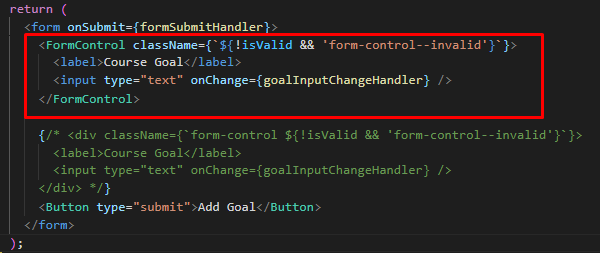
If we want a styled components generated dynamically

Now we are going to create another component but this time inside the CourseInput component. Usually it is not recommended to create two components in a single component file but this time should be fine because we only use it here

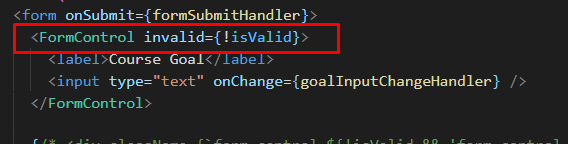
Below is our component, we deleted form-control and added & where necessary



And notice below now we also deleted the initial form-control class because that is applied by default.

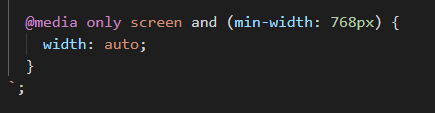


There is another method to change the className we can set props on FormControl and use them inside the backticks as a conditional





## Styled Components and Media Queries



Nothing special just add them between curly braces

## Using CSS Modules

Styled components look ugly in my opinion but we are going to try approach number 2

CSS modules are only available in projects that are set up for it because some code transformation has to be done before displaying the page good news is that React is already set up this way.

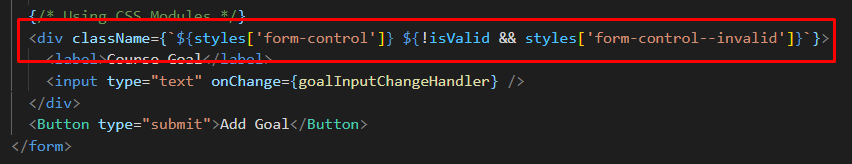
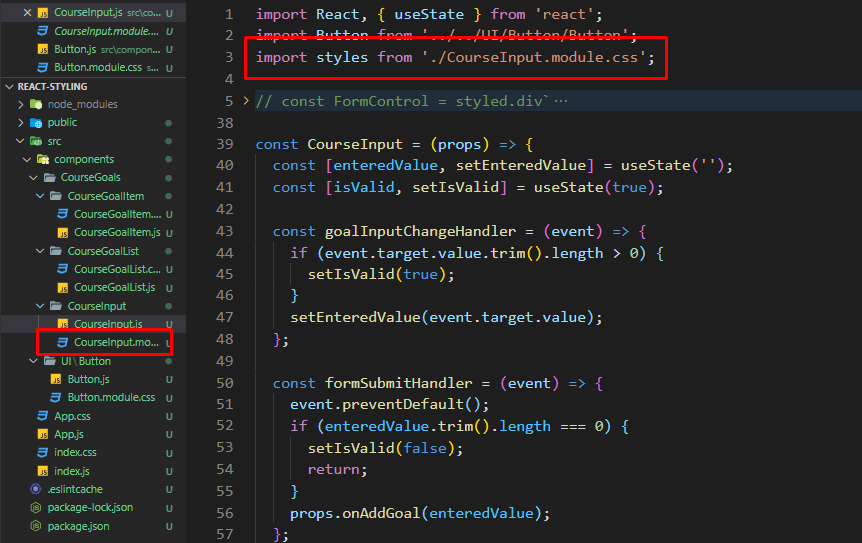
In order to make CSS Modules work we have to do a few things

1. Rename our CSS file to myFileName.module.css
2. Import styles ‘./Button.module.css’
3. Import styles from ‘react’ (not really needed)
4. Change the className to className={styles.button}

The styles does some transformation under the hood so it will return an object and that object contains all our CSS classes so if we want to use a class from our CSS file we just type className={styles.myClassName}

What does CSS modules do ? Under the hood it changes the classnames to unique ones. So if you import one CSS file into multiple component it will create unique variants of those classes

And this is how we apply it to the form



# Debugging React Apps

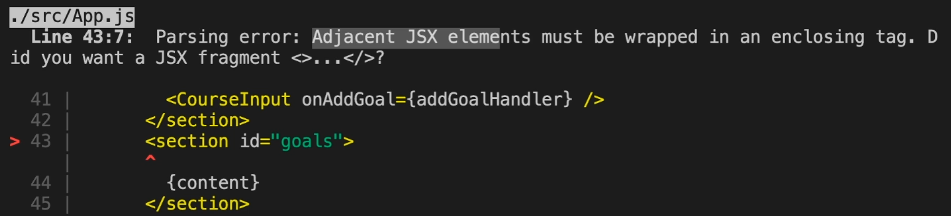
## Introduction

What will we learn

* Understanding error messages
* Debugging and Analyzing React Apps
* Using the React DevTools

## Understanding React Error Messages

We got a test project from the Teacher where there were intentional bugs so we try to fix these



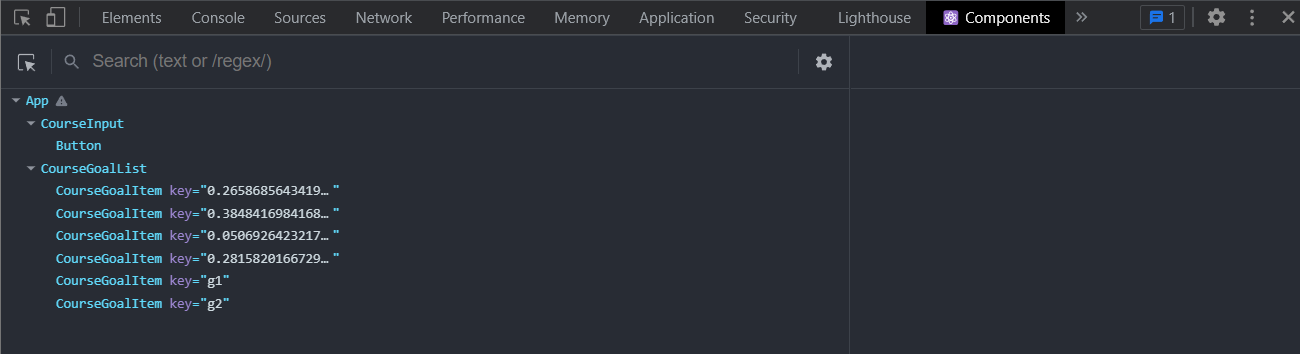
This is the first error because we have 2 adjacent elements, we can’t have that in react so we have to wrap them in a div. By the way we can’t do this because under the hood react uses React.createElement() and we can’t have two of them in a component

## Working with breakpoints

Use the fking debugger

## Using React DevTools

There is a dedicated tool for react development. Download it from google and this will give us 2 more tabs in the developer console



Notice we can see all the components

# A complete practice project

## Module Introduction

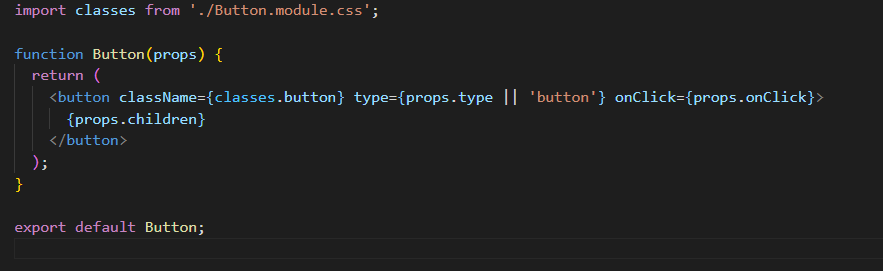
Here we will be creating a practice app that encapsulates all the stuff we learned until this point

First time I did the project on my own and managed to make it all work and now I go trough Max`s example to see what did he do differently.

Some notes to add

## Creating a button component

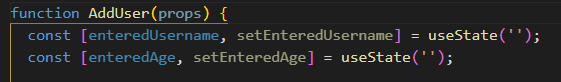
In order to create a reusable button component we have to do the following steps.

Notice below that we added some props, without these props we cannot use the props we set below. For example when we use Button in a different component and want to set the type of the button to submit it won’t work unless we have the type set in the main button component.

## Differences between my method and his

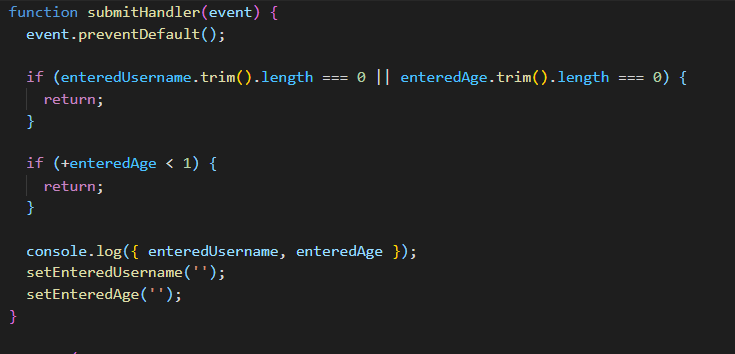
Instead of 2 different states for enteredUser and enteredAge I used a single state that had an object and I also forgot to use previousState when setting it.

Max used 2 actually separate states



## Basic Data Validation

We are going to handle the basic data validation in the form itself

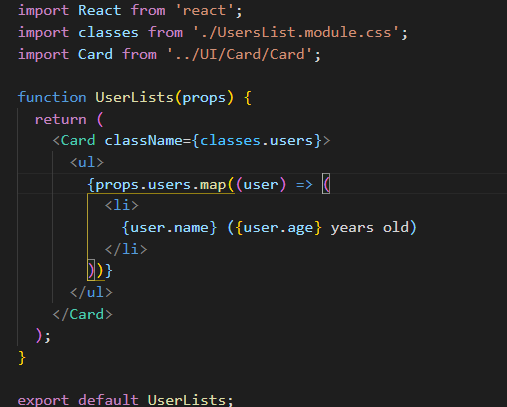


We check if the user didn’t enter more than 1 char in both input fields, If they did then the form will validate .

And we check if the age is higher than 1 . Normally Javascript would convert our enteredAge (which is a string by default to a number) but to be sure we add the + sign to convert it to a number

## Adding User Lists Component

It’s pretty straight forward, very similar to my approach now we have to manage a list of users via state



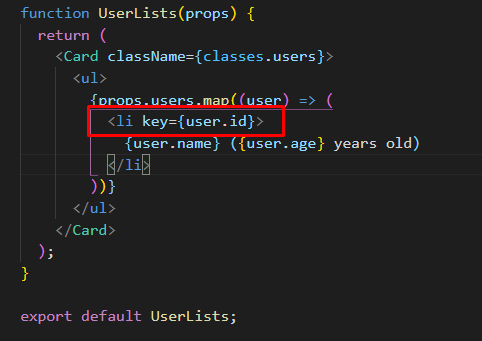
## Managing a list of users via State

Now we are going to lift the state up of the AddUser component to add new users to our list. And we lift it to a component where we have access both to the AddUser component and to the component that manages our lists of users which is App.js

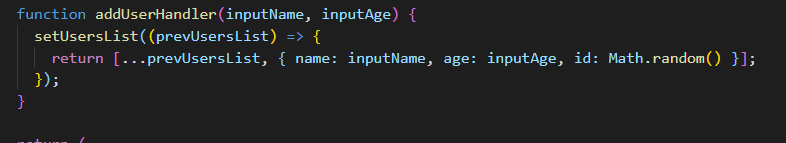


Good thing to note is that it’s better to use on before a prop that acts like a function because its more appropriate. Take the example above where the prop name we gave was onAddUser instead of something like addUser.

Where do we put the unique key identifier ?



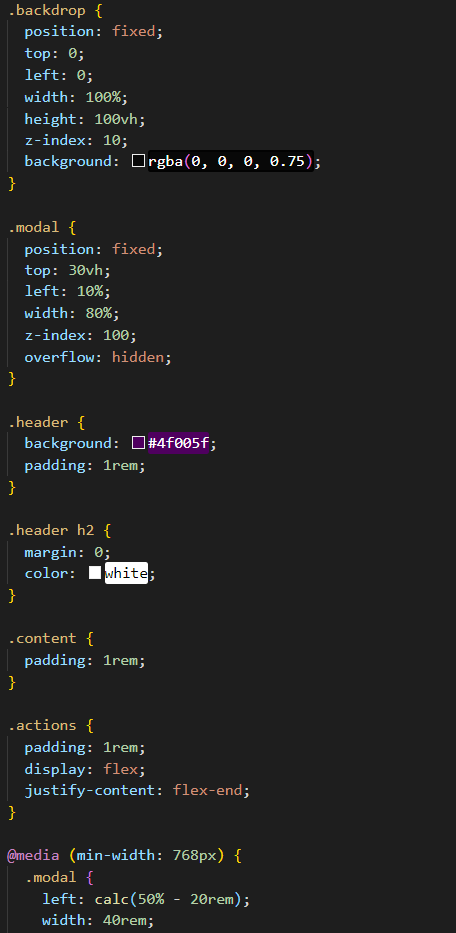
We put it on the li and the key itself is just a Math.random() number that we add when we update the state of the list



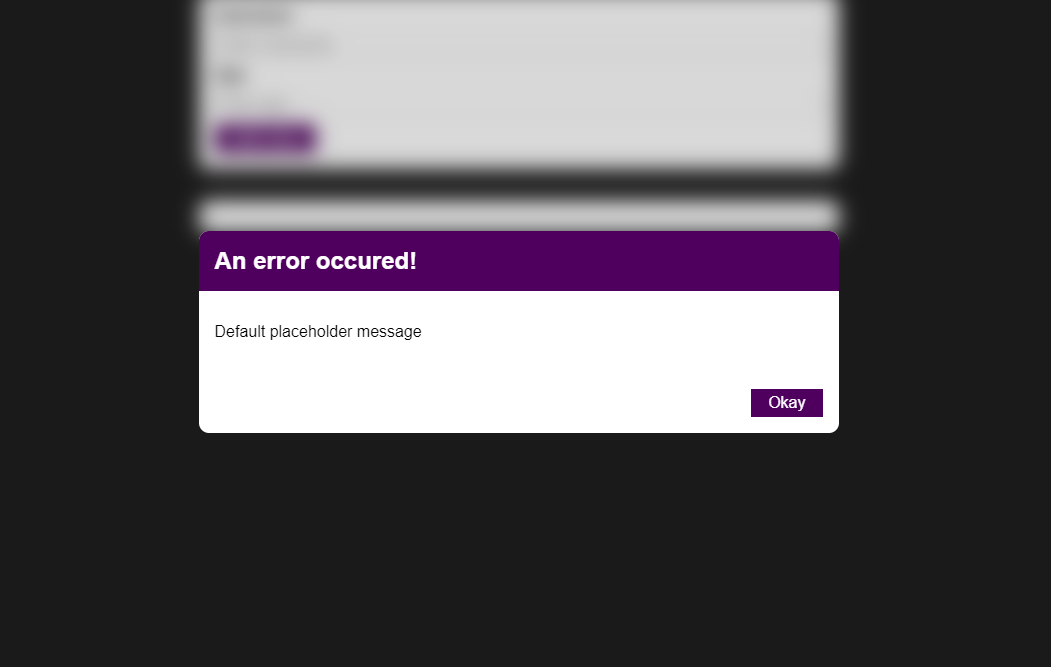
## Adding the “ErrorModal” component

This is how the code looks

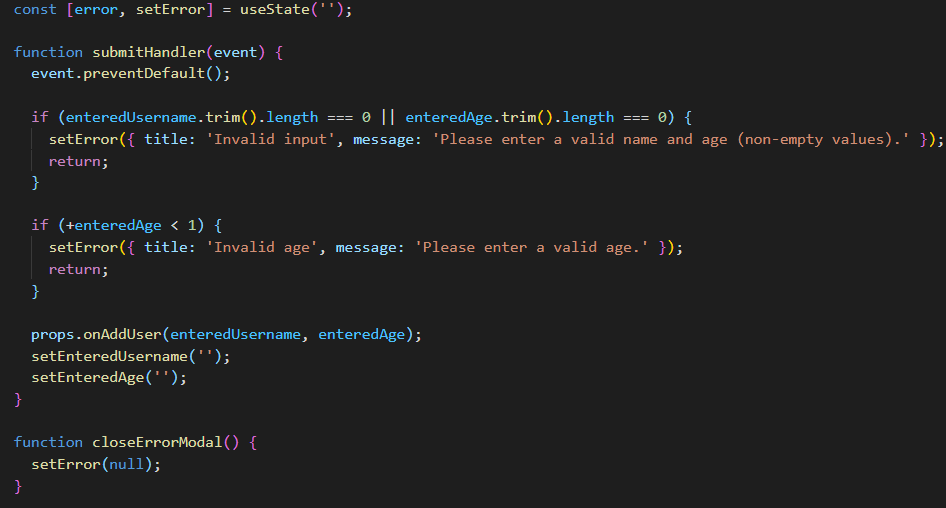




Max argued where should we put this modal. In my project I put it in the App.js component but Max put it inside the AddUser component because that component is triggering the modal.



This is how the modal looks but now we have to manage the state of this modal. We need a state manager for the error modal because its basically screaming state the state of the from will be given by the from submit handler this will set the state to true or false



Now whenever the form is invalid the setError method will create an object with the appropriate error.

And we are going to shortcircuit the and operator to display the errormodal or not

The only thing left to talk are the handlers which basically setError to a falsey value thus causing the ErrorModal to disappear, but we want to add this handler on the Okay button and also on the backdrop.



# Working with Fragments, Portals and Refs

## Module Introduction

Here we cover more tools that help us problems.

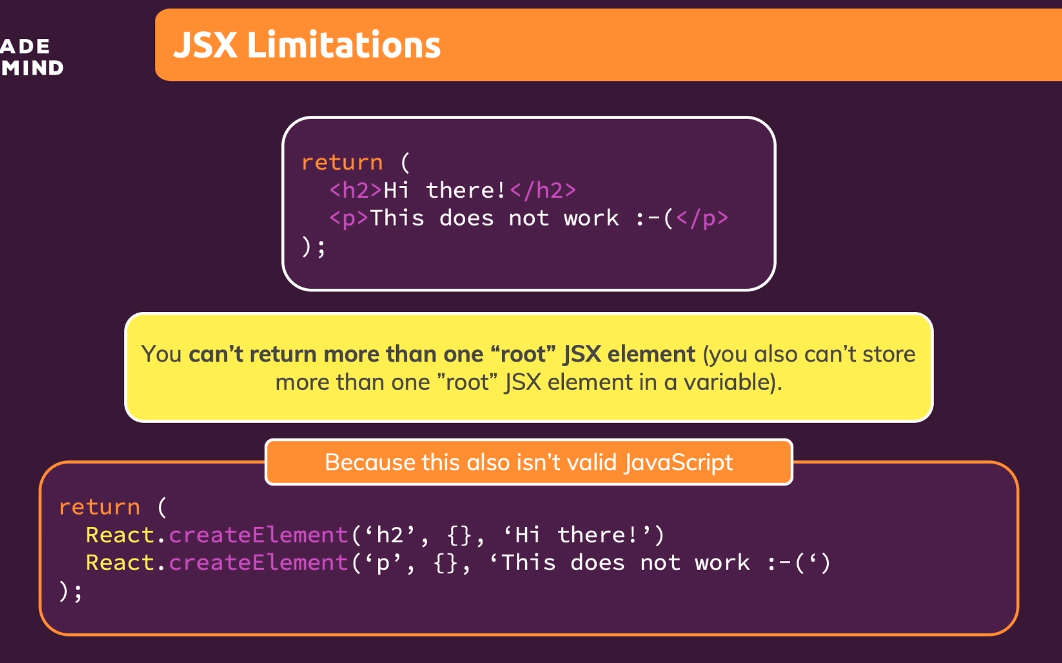
-JSX Limitations and Fragemnts

- Getting a cleaner DOM with portals

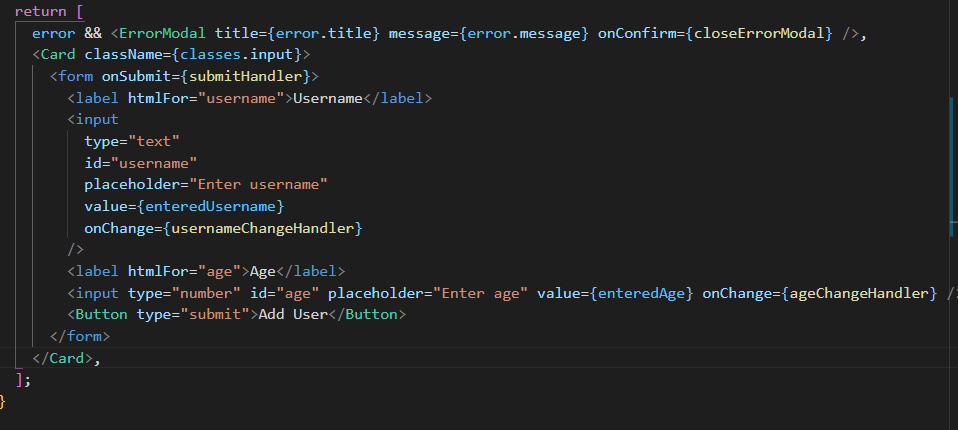
- Working with refs

## JSX Limitations & Workarounds

In React we worked a lot with JSX but it has certain limitations like having adjacent root level elements will return an error



The first workaround is what we used until now => Wrapping the two adjacent elements into a single div. But if we do not want to use a div we can put them into an array but you will get an error because everytime you are working with elements in arrays react wants a key for that element



This solution is a bit cumbersome so we will be sticking to the div wrapping but the problem with this is that we end up with the div soup problem

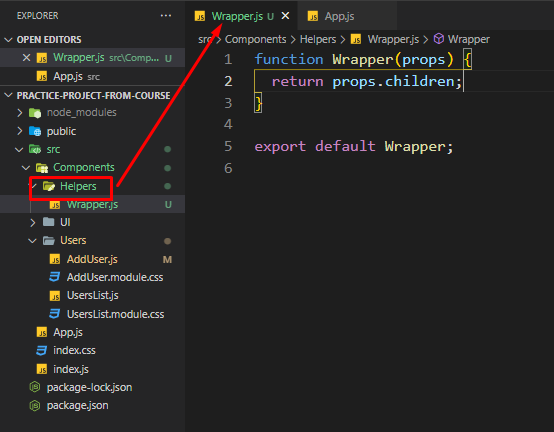


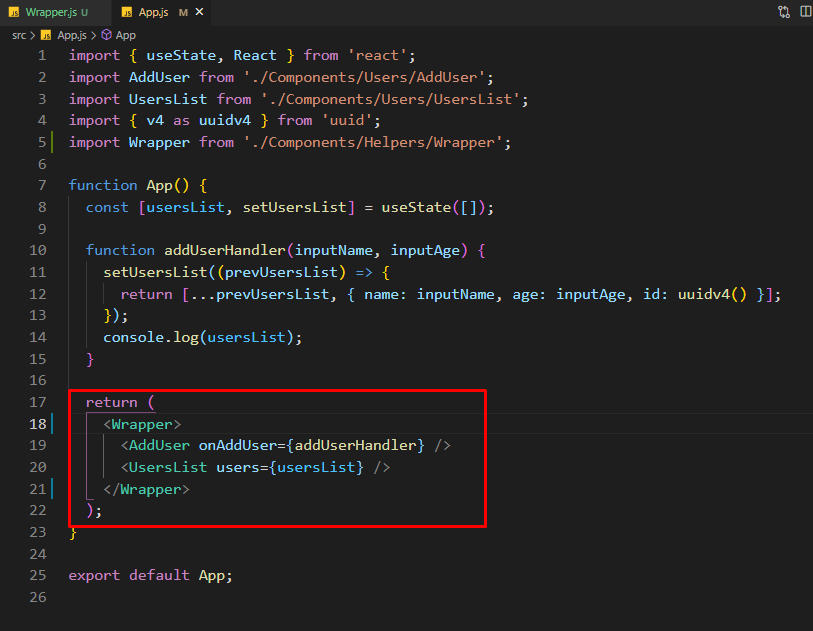
And these divs are only there because the limitations of JSX. In bigger apps in your final HTML page you can end up with a lot of divs without any semantic meaning because of this limitation. You can ignore this but it can cause performance issues because react has to check those divs.

This approach is ok but not ideal.

## The solution -> Creating a wrapper component

We will create a wrapper component that only returns props.children (the content between the opening and closing tags)

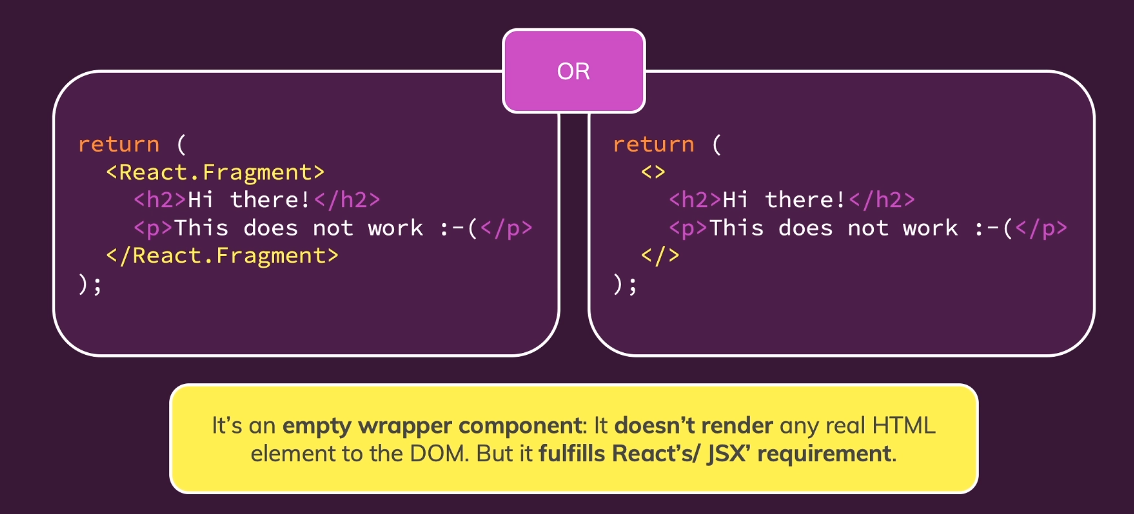




Just to be clear this is an empty component, all it does to hack this workaround. Also to be added that we made a new Helper folder for this wrapper

## React Fragments

Because this workaround is so obvious react has a built in wrapper for this called React Fragments



We can either include React.Fragment or use <> </> empty tags but for empty tags our project has to be set up in a specific way.

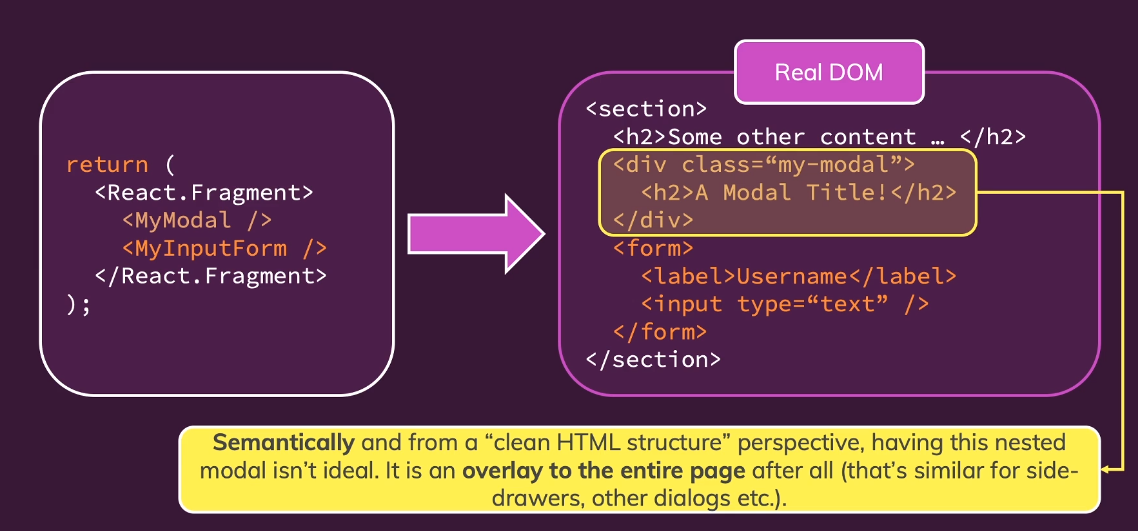




## Introducing React Portals

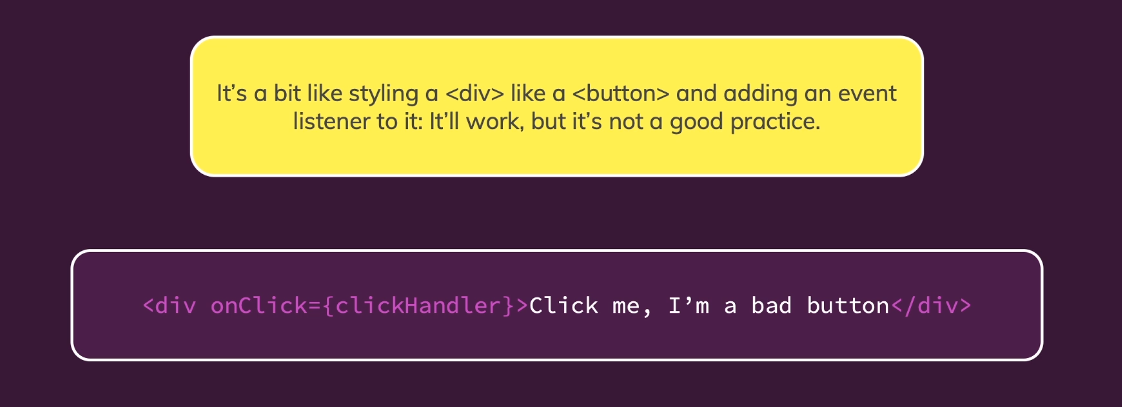
Fragments allow us to write cleaner code and not end up with unnecessary tags. React portals do something similar . Consider the following example;

Basically the whole idea of portals is that the rendered HTML is moved somewhere else in the DOM.



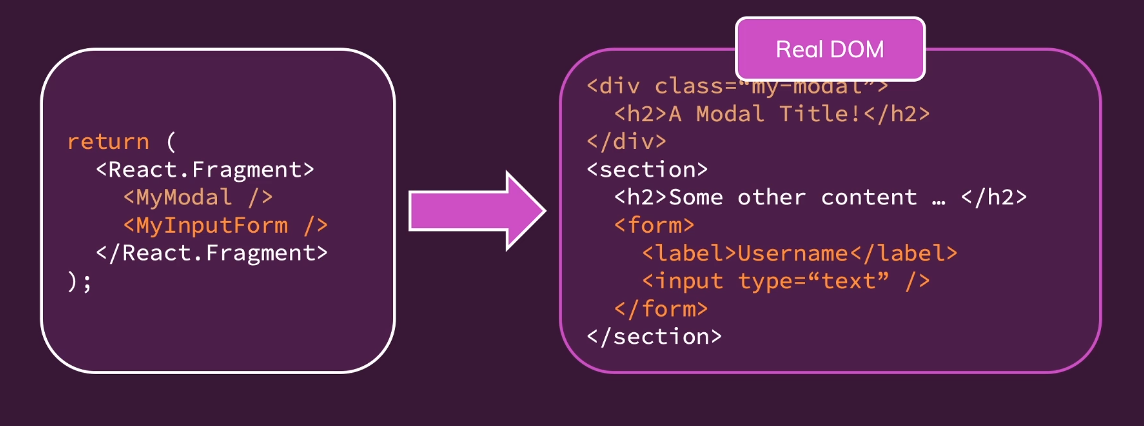
What is wrong with this code?

Technically nothing only semantically. A model is something that goes above every content and it can even lead to real problems with styling because if a screen for example had to read the content it might not see this as a general overlay. This is not only for modals, it is typical for all kinds of overlays like : side drawers, dialogs , pop-ups etc.



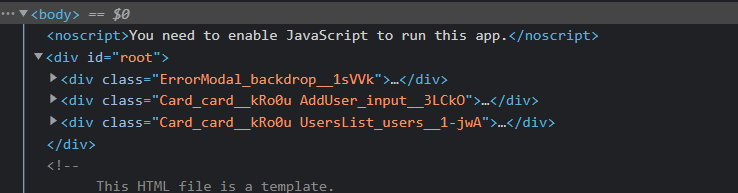
Here it is when portals come into play.

We can use portals to keep writing the same code as we did before but to render it differently in the real dom

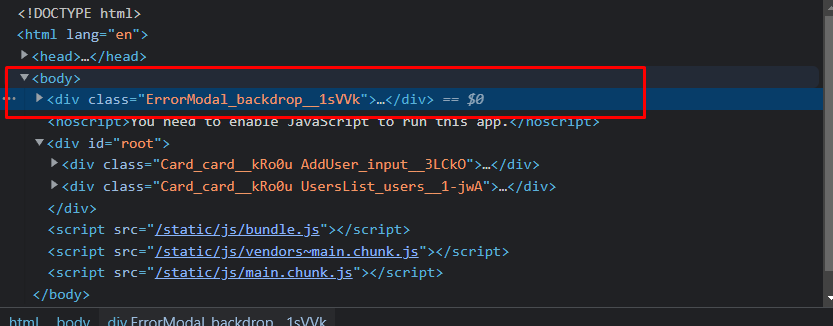


## Working with portals

First of all I will show the initial place of render of our modal. It renders next to our cards instead of our root div. This is a simple example but in a more complex projects these would be deeply nested in some other components.

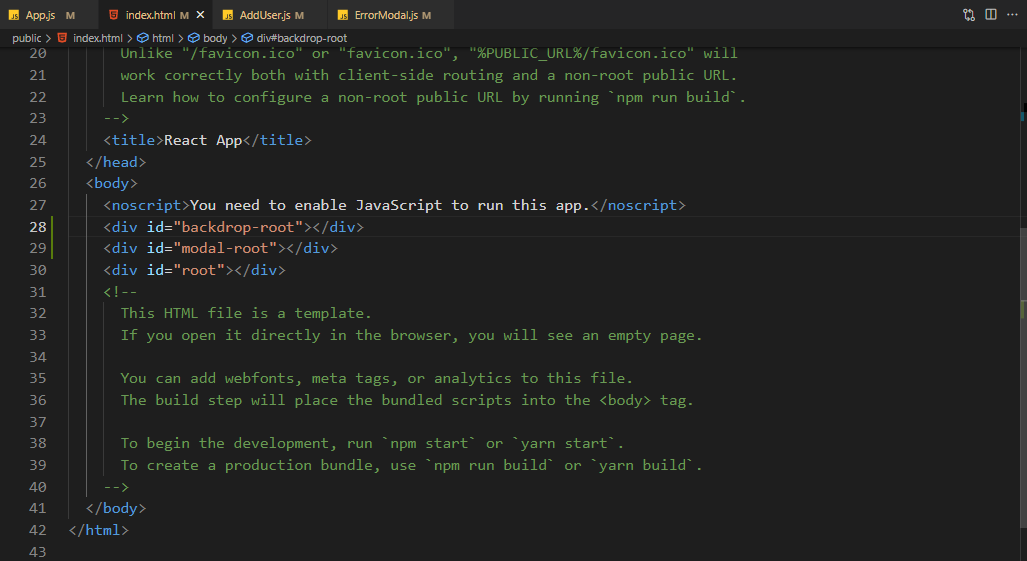


Ideally we would like to have the model a direct child of the body.



This is something we can achieve with portals. Portals need 2 things : The place we want to component ported to and to let the component know we want it ported there.

To mark that place we head into our public folder into the index.html



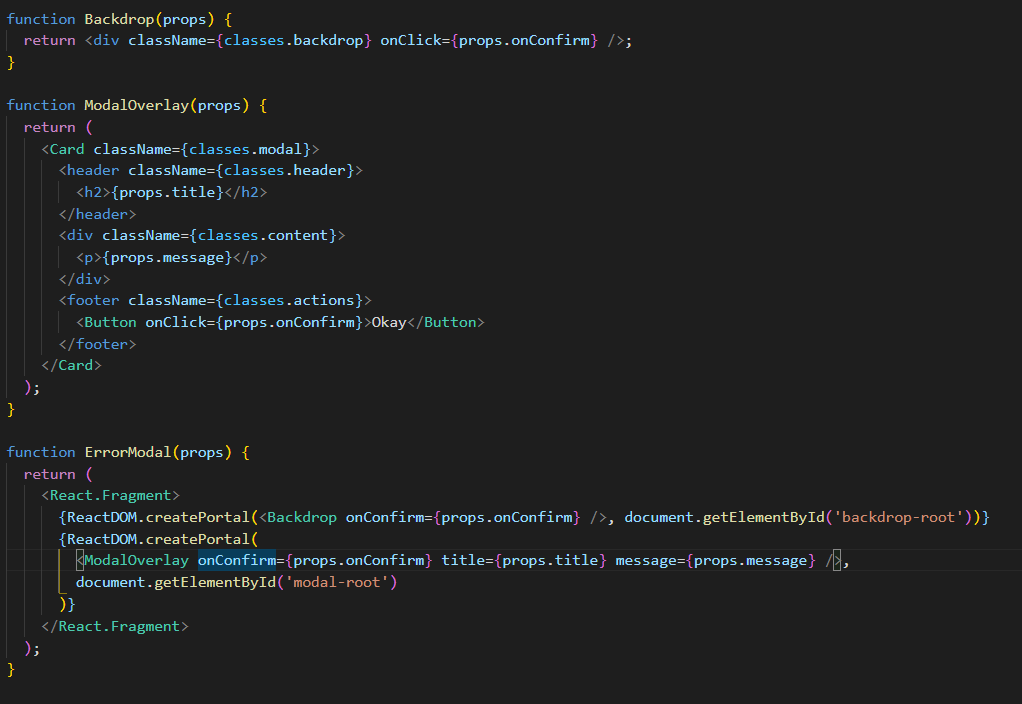
This is a common technique for other components like this. We will also do some splitting to make using portals easier.

Notice how we split our ErrorModal into 2 elements.

Below are the imports we need



Quick thing to note is that ReactDOM.createPortal still uses JSX between the braces



## Working with “ref”s

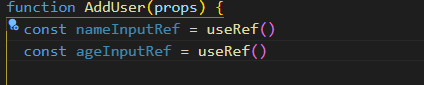
Ref is short for references. In their basic form let us get access to other DOM elements and work with them.

An example where we are going to use refs is in our AddUser form ,here we have 2 states that manage the username input and the userage input. These states updated on every keystroke and that is a bit redundant

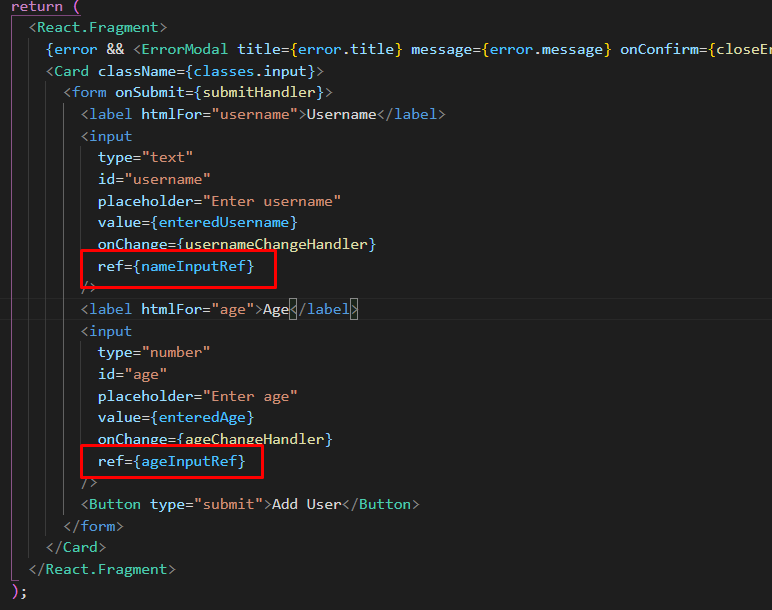
With refs we can setup a connection between an HTML element and JS Code

Creating a ref





Now we let react know that we want to use these refs

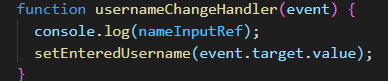


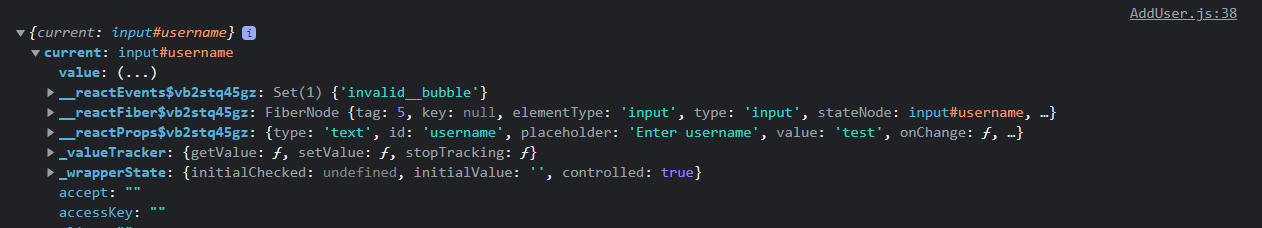
Now the refs are connected to the elements. Now when the first time react reaches the code where we put the ref and renders it, it will set the value of set ref to the native DOM element that is rendered based on the input.

**So the thing inside the nameInputRef and ageInputRef later will be a real DOM element.**

Whats the benefit now?

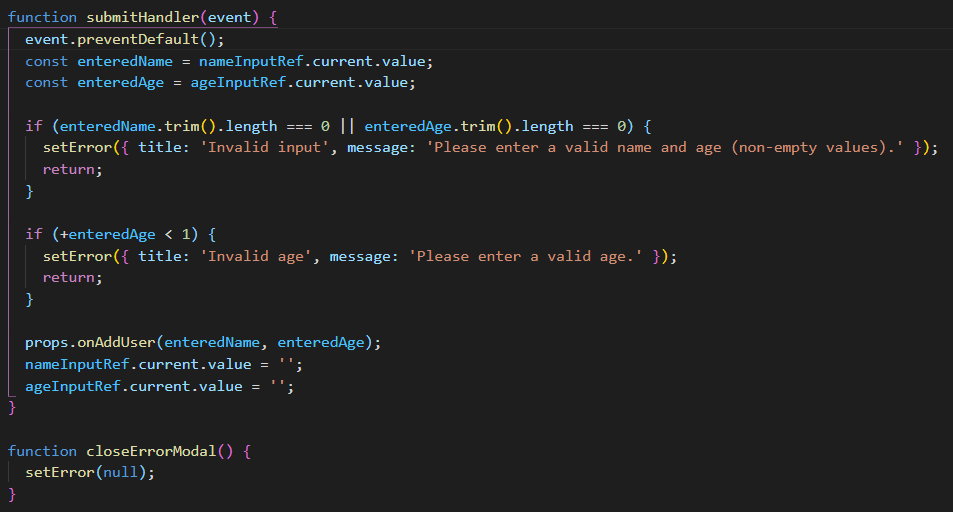
If we console log ref when we type something into the input it will return the currentDOM node of that element





Changing this node is not recommended, we let React do that but reading data is totally fine. Now why use refs over states. In our case states are a bit overkill for managing 2 inputs and they also update on every change.

We are going to use refs to get the entered data from the two input fields since when the submitFormHandler executes it will get the value of the refs which are basically the currentDOM node of each input element



Notice in the code above that we removed the states of both inputs thus we also lost our 2 way binding and reset logic, now Max said that we shouldn’t use refs to change the DOM but in this case we can do it since we only change the input value and not add other components or anything complex.

Also in the future if we want to read some data from the DOM we should use refs instead of useState because useState is just overkill (unnecesarry code) for reading a simple value from the DOM. They can even do more than that and we will learn about them later in the course.

## Controlled vs Uncontrolled Components

In the lecture before we showed an alternative way of getting the user input and also a hacky approach to still manipulate the input fields. This approach of interacting with DOM elements specifically with input elements also has a special name, **we are talking about uncontrolled components when we access values with a ref because their internal state is not controlled by react**

Why is this ? Aren’t we setting it with react? Technically yes but no .

When we use refs even tough they are part of React we set the value on the DOM not from react

So the controlled approach was the on from before where we had a state for each input and on the submit we would set both input back to an empty string with useState

And the uncontrolled component is when we use refs to get the input from both boxes then set the DOM content of the input to an empty string

# Handling Side Effects, using Reducers & using the context API

## Introduction

* What (side) effects are?
* Managing more Complex State with reducers
* Managing App-Wide or Component-Wide State with Context

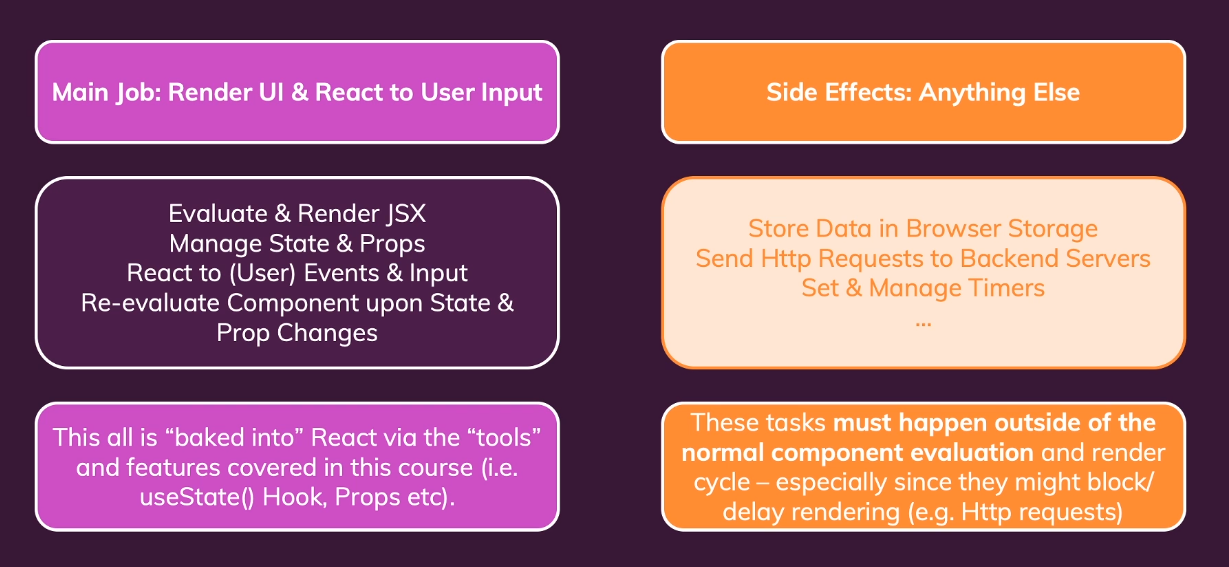
## What are "Side Effects" & Introducing useEffect

What is an effect (also called a side effect)

Lets not forget that React has one main job and that is to Render UI , React to user input , re-render UI if needed

Now lets as example a UserInput component, all it does it get the input data from the user, use other components if needed (errorModal) and than use that data to render something to the screen therefore side effects are everything else that might be happening in our application like sending an HTTP request, storing something in localstorage, timers , intervals but these tasks are not related directly to bringing something to the screen.

Obviously you might be sending an HTTP request and waiting for some data but that’s not something you need react for, react doesn’t care about that and its not related to drawing something on the screen. **These tasks must happen outside of the normal component evaluation**



We will have a better tool to handle side effects, the useEffect() hook.

The useEffect hook takes in two arguments :

* A function that should be executed after EVERY component evaluation IF the specified dependencies change
* Dependencies , an array full of dependencies which if changed the function of useEffect will run



## Using the useEffect() hook

Lets take the following test page as an example



A good useEffect on this page would be to send an HTTP request to the server to check the email and password. Another effect we will use here is to stay logged in even after the user reloads the page. Currently the logged in state is managed in the form and when we reload the input shows up again.

So we will store the logged in state somewhere where it persists even after reload and better than that we check when the app starts for that persistent data if it is there

We will use localStorage for this.



Take the example above. This looks fine at first glance but it produces an infinite loop. App is called, it sets the loggedIn state to false then gets to the next line , gets the isUserLoggedIn variable from local storage and if it is ‘1’ then calls the setIsLoggedIn method which triggers another re-render thus causing an infinite loop. This is why we need useEffect()



We are going to put our code inside the function of the useEffect() hook.

Important this useEffect() function is executed by react AFTER the component ran. It will not just run after the whole component ran but it will only run if the dependencies change.

1) useEffect hook without mentioning any dependency array like.. useEffect(someCallbackFuction); runs for every render of the functional component in which its included..  
2) useEffect hook with an empty dependency array like this..  useEffect(callbackFunc , [] ) is executed only for the the initial render of the the functional component. And then it will not run in the further renders of the same functional Component..  
3) useEffect hook with some dependencies inside the dependency array like this.. useEffect(callbackFunc , [dependency] ); will run for the initial render as well as when the render happen due to change in dependencies mentioned in the dependency array...

And lets not forget the isLoggedIn has nothing to do directly with the rendering of the app even if it might seem tought , it is just a side effect.

Take the following example

Function add(a,b){

Let sum = a + b;

Return sum

}

This is a pure function with no side effects

Function add(a,b){

Let sum = a + b;

Console.log(sum)

}

This has a side effect , console.log is the side effect.

Take the example above and think about react and what it does usually + what is localStorage and if localStorage by itself has anything to do with rendering a component.

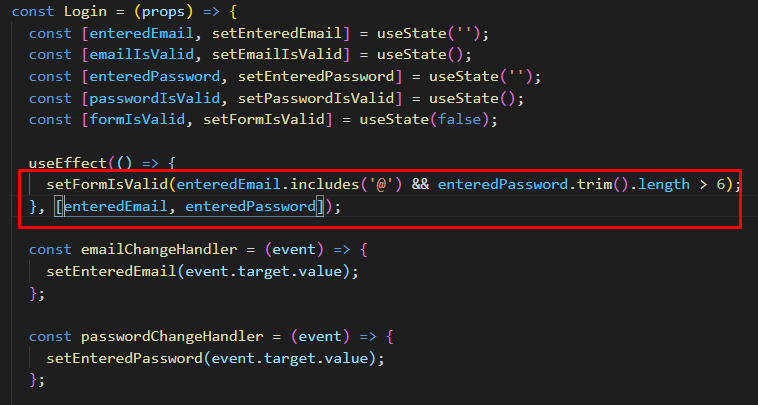
## useEffect & Dependencies

We are going to use the useEffect hook in our form validation to restructure our validation logic.

The form is setup in a way that it sets the form to a valid states (highlights input boxes with green) if 2 conditions are met: The email input includes an @ symbol and the password is longer than 6 characters. Now this validation occurs everytime the user types something in an input box



But we can use an effect for a single validation check



The highlighted function will run any time a change occurs in one of the dependencies.

So notice that setFormIsValid is doesn’t use something that is not related to react like local storage or HTTP request but instead functions and variables inside our Login component. useEffect is there to handler side effects and often those are http requests and other stuff but a keystroke is also a sideffect

**useEffect is a super important hook that should be executed in response to something. This something could be the component being loaded, an http request or any other thing.**

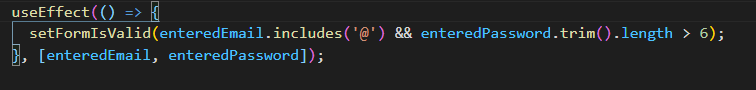
## What not add & not to add as dependencies

* Don’t need to add state updating functions , react guarantess that those don’t change
* Don’t need to add built-in APIs or functions like fetch(), local storage. These browser APIs / global functions are not related to the React component render cycle and they also never change
* Don’t need to add variables or functions defined outside of the components .Such functions or variables also are not created inside of a component function and hence changing them won't affect your components (components won't be re-evaluated if such variables or functions change and vice-versa)

So long story short: You must add all "things" you use in your effect function **if those "things" could change because your component (or some parent component) re-rendered.** That's why variables or state defined in component functions, props or functions defined in component functions have to be added as dependencies!

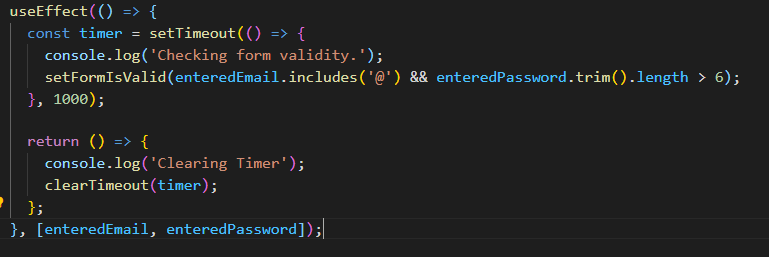
## Using the useEffect cleanup function

Take the example below



This useEffect is ran on every keystroke because enteredEmail and enteredPassword changes on every keystroke this doesn’t have a huge performance impact here but imagine in a real app where you would send an http request every time to the server to check if the username is valid or not. To avoid this we can use a technique called debouncing.

Debouncing is a programming practice used to ensure that time-consuming tasks do not fire so often



Take the example above and notice that in the useEffect function we have a return statement , that return statement is ran before our timeout but only the second time the useEffect() is called. So basically the following is happening on each keystroke: Previous timer gets cleared by the return statement then the new timer is called this will keep clearing until we keep pressing.

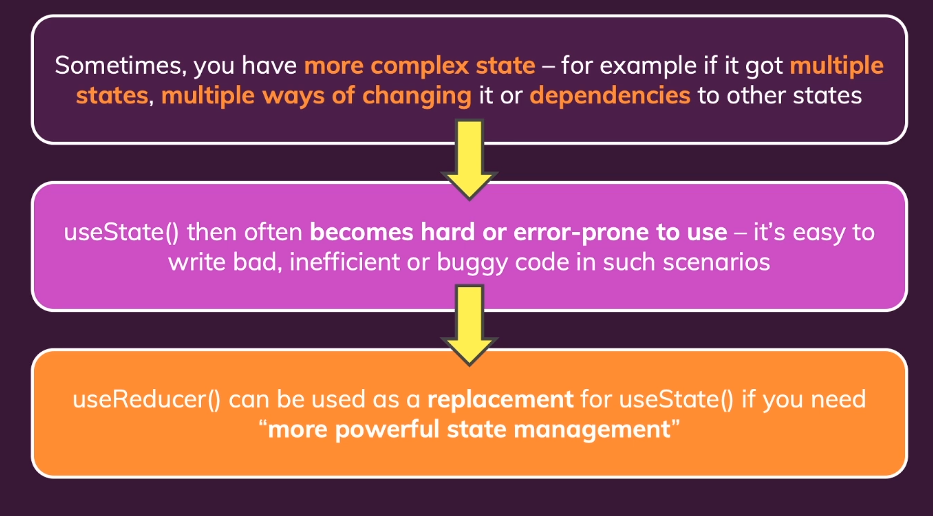
One more thing to add is that useEffect runs ONLY after the component render cycle.

## Introducing useReducer & Reducers In General

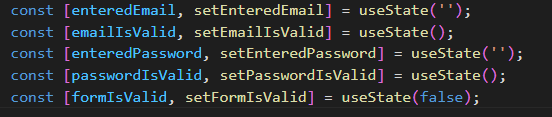
useReducer() is good in a scenario where you update a state which depends on another state or if you combine similar states (state that have to do with similar stuff)

useEffect is the second most important hook besides useState but the next one is useReducer.

This is another builtin hook that will help us with state management, it is a bit like useState but with more capabilities. Sometimes you have more compex state for example if it got multiple states, multiple ways of change it or dependencies to other states.

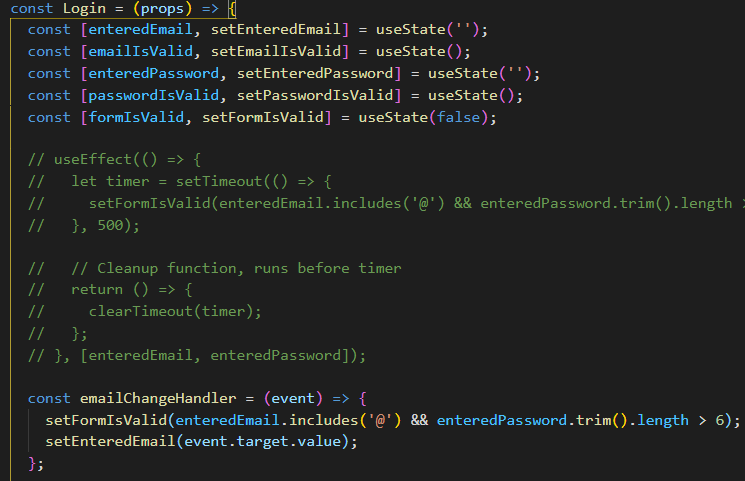


For a good example we will go back to the Login.js file. So basically useReducer is a more complex useState and in some cases like the one I will be showing below it is worth setting up a useReducer



You could argue that this big blob of states could be one state that manages the whole form state or even couple them together like if email is valid and the enteredEmail si valid

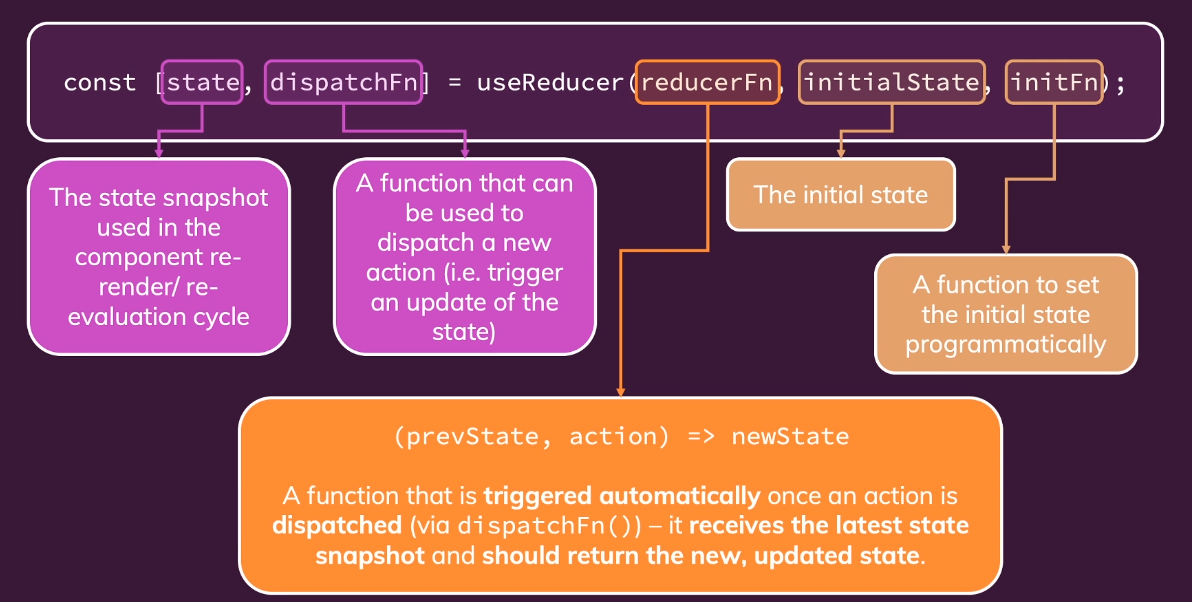
And to demonstrate useReducer we are going to go back to the old code where we didn’t use useEffect



Notice that we use setFormIsValid which is a function from useState and it depends on a previous state but instead of depending on one previous state, it depends on two previous states.

Don’t forget we used before the function form with the prevState argument but here we cant use it because we depend on two other states and not the previous state of form validity which is true or false.

## Understanding useReducer()



Simple useReducer example from outside the course

We have to manually create 2 things here :

* Create a reducer function
* Set the initial state

We can set the initial state to what we want but usually we should use an object because if we use a reducer it will probably be a more complex state.

We can put the reducer function outside our component and this function takes in 2 arguments . The state and the action. The state is obviously the current state and the action is an object that we set on the dispatch function.





Use reducer in our example

function emailReducer(state, action) {

  if (action.type === 'USER\_INPUT') {

    return { value: action.payload, isValid: action.payload.includes('@') };

  }

  if (action.type === 'INPUT\_BLUR') {

    return { value: state.value, isValid: state.value.includes('@') };

  }

}

function passwordReducer(state, action) {

  if (action.type === 'PASSWORD\_INPUT') {

    return { value: action.payload, isValid: action.payload.trim().length > 6 };

  }

  if (action.type === 'PASSWORD\_BLUR') {

    return { value: state.value, isValid: state.value.trim().length > 6 };

  }

}

const Login = (props) => {

  const [formIsValid, setFormIsValid] = useState(false);

  // Reducer for managing email state

  const [emailState, dispatchEmail] = useReducer(emailReducer, { value: '', isValid: null });

  const [passwordState, dispatchPassword] = useReducer(passwordReducer, { value: '', isValid: null });

  useEffect(() => {

    let timer = setTimeout(() => {

      setFormIsValid(emailState.isValid && passwordState.isValid);

    }, 500);

    return () => {

      clearTimeout(timer);

    };

  }, [emailState, passwordState]);

  const emailChangeHandler = (event) => {

    dispatchEmail({ type: 'USER\_INPUT', payload: event.target.value });

  };

  const passwordChangeHandler = (event) => {

    dispatchPassword({ type: 'PASSWORD\_INPUT', payload: event.target.value });

  };

  const validateEmailHandler = () => {

    dispatchEmail({ type: 'INPUT\_BLUR' });

  };

  const validatePasswordHandler = () => {

    dispatchPassword({ type: 'PASSWORD\_BLUR' });

  };

  const submitHandler = (event) => {

    event.preventDefault();

    props.onLogin(emailState.value, passwordState.value);

  };

  return (

    <Card className={classes.login}>

      <form onSubmit={submitHandler}>

        <div className={`${classes.control} ${emailState.isValid === false ? classes.invalid : ''}`}>

          <label htmlFor="email">E-Mail</label>

          <input

            type="email"

            id="email"

            value={emailState.value}

            onChange={emailChangeHandler}

            onBlur={validateEmailHandler}

          />

        </div>

        <div className={`${classes.control} ${passwordState.isValid === false ? classes.invalid : ''}`}>

          <label htmlFor="password">Password</label>

          <input

            type="password"

            id="password"

            value={passwordState.value}

            onChange={passwordChangeHandler}

            onBlur={validatePasswordHandler}

          />

        </div>

        <div className={classes.actions}>

          <Button type="submit" className={classes.btn} disabled={!formIsValid}>

            Login

          </Button>

        </div>

      </form>

    </Card>

  );

};

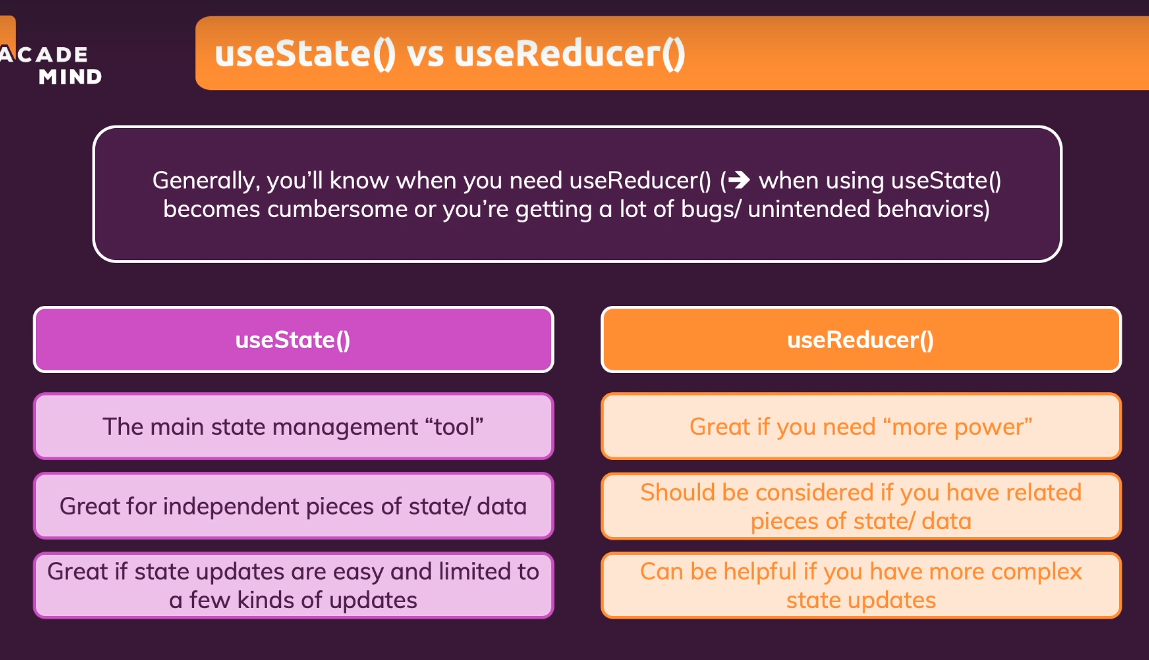
The code above is but if I ever come back here for need of explanation it will be ok.

Notice how for the setFormValid we use a simple use state because we just want it either true or false.

Now we want would like to implement useReducer because we have 4 other states that we want to handler : email input, email validity , password input and password validity

We group this into 2 reducers : emailReducer and passwordReducer where we handle the logic above. This is the main thing to remember from here: **useState for simple states useReducer for more complex state**.

## useState() vs useReducer()

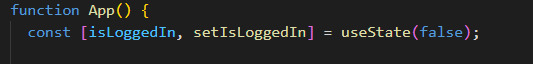


## Introducing React Context (Context API)

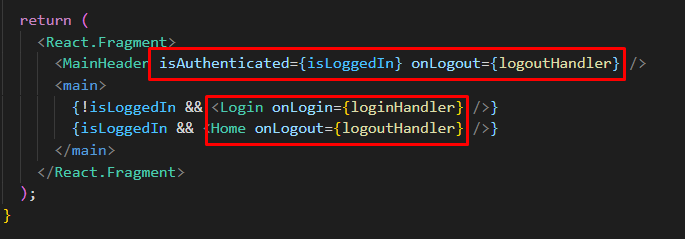
A bigger problem that we might face in bigger applications but even in our little project that can be solved in a more elegant way.

**In short: React Context API is used for passing down props to other components without having to pass it down in each chained component by wrapping the Provider of the data in the ReactContext**

We are taking “about” a problem where we are passing a lot of data trough a lot of components with props



Take the isLoggedIn state, this state we pass on to many components that need this state

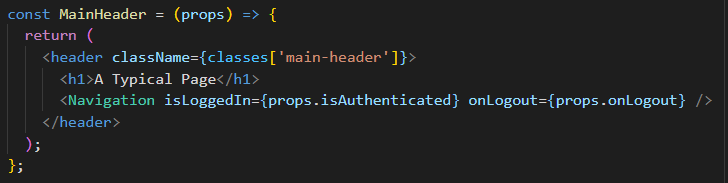


All of these use the isLoggedIn or setIsLoggedIn state.

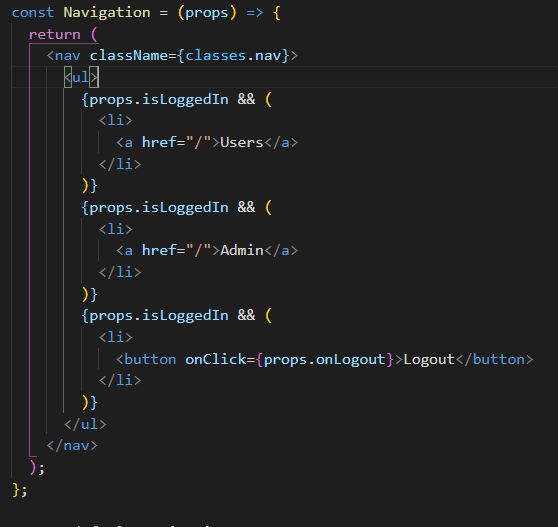
Take the following example



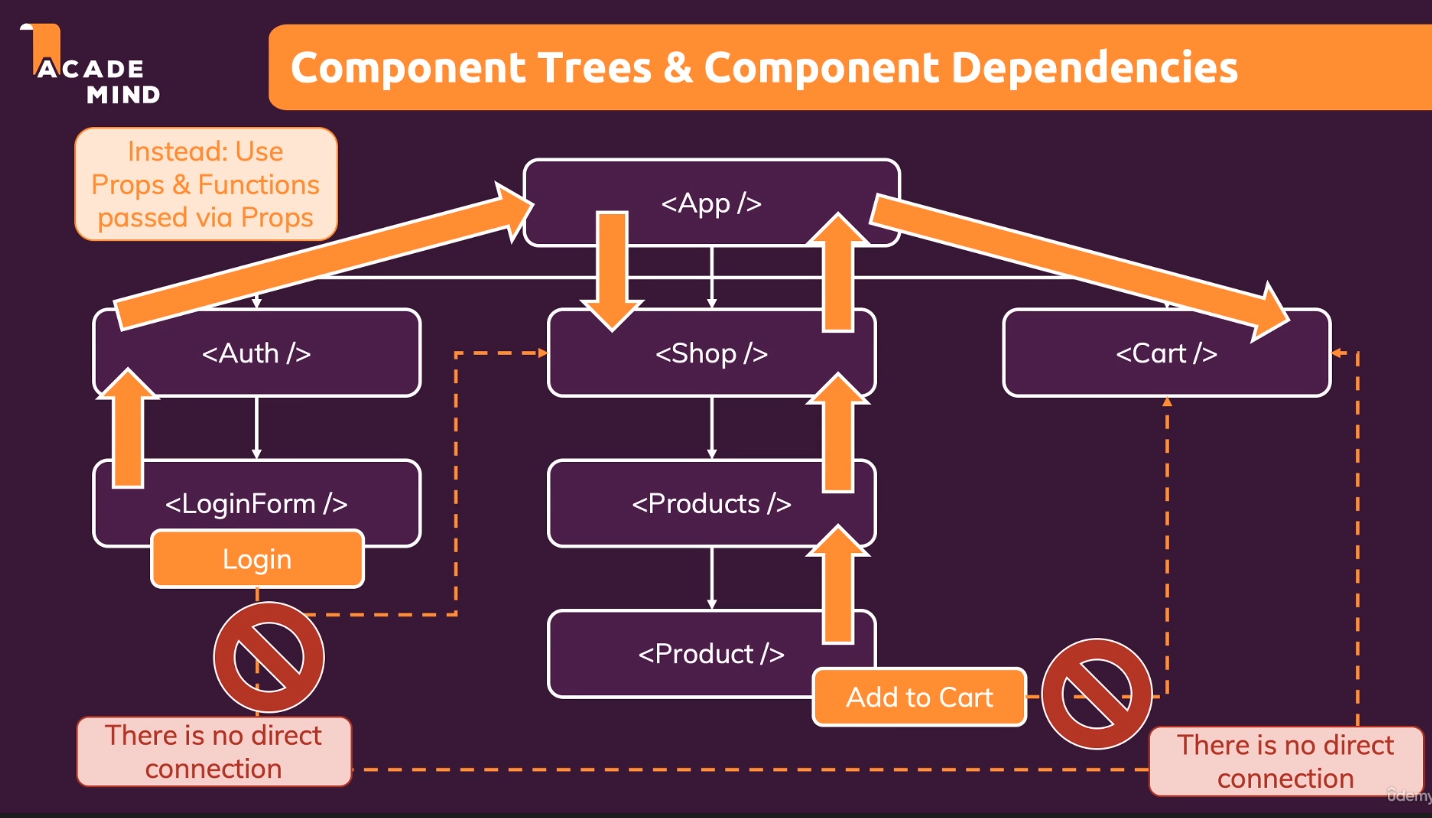
Here we pass these two props down to MainHeader



And in MainHeader we pass it down again to <Navigation> and basically this is the component that needs the data to render different links



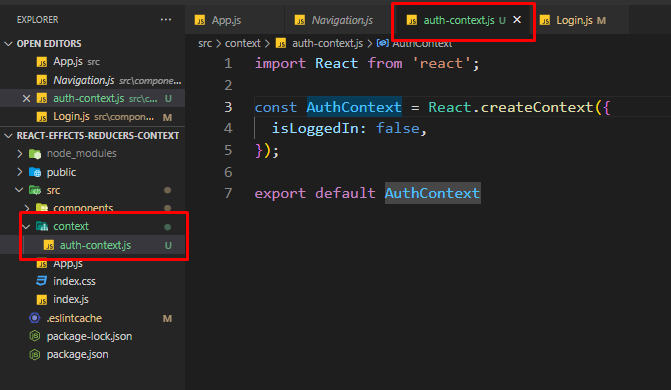
We are forwarding props from App > Header > Navigation , this isn’t a problem but in bigger apps this chain can become very long.



It would be nice for a component to only receive the data it needs from a parent and not other data for that we got a component wide behind the scenes state storage which allows us to solve this problem.

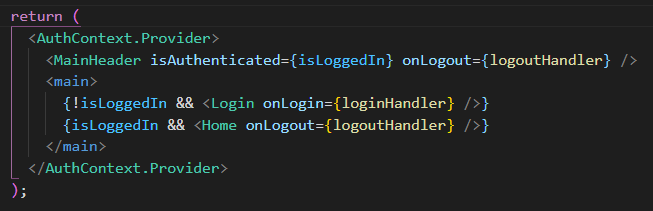
## Using React Context API

First we have to create another component for our context



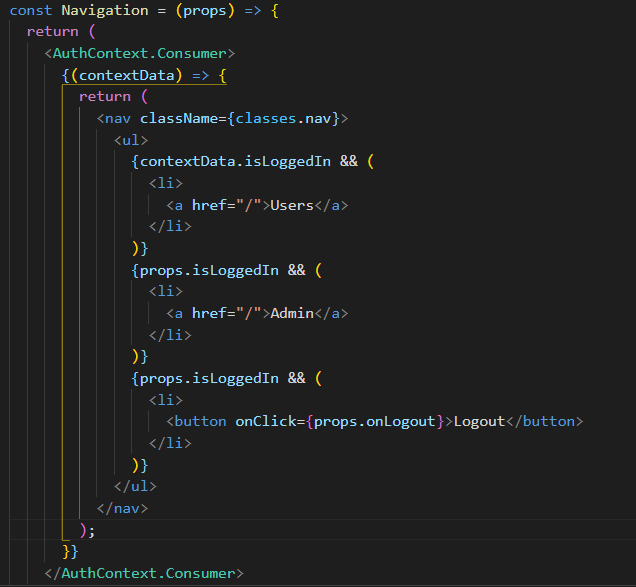
We are going to use the kebab-case for the context naming and then we import react in order to use React.createContext() which accepts an object as an argument.

Now this is not a component but we can access a property when we use it in our JSX code that is a component and wrapping other components with this AuthContext will make ALL their descendants have access to what have inside auth



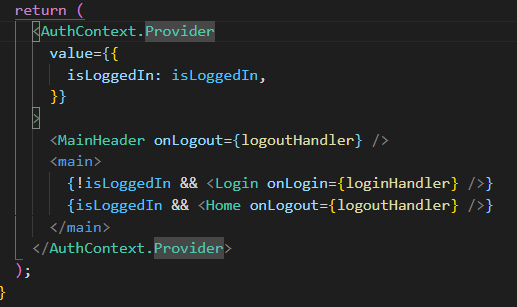
Now in our App.js we wrap everything with AuthContext.Provider

Now in order to have access to the object provided by AuthContext we have to listen, and we can listen in 2 ways. By listening with a consumer or using a react hook



This is our consumer the Navigation component. Now the Context consumer works a bit differently. Between the brackets should be a function that uses the object we pass into our context as an argument and returns the JSX code we had before, however we will get an error because we have to do one more thing.

Technically the provider isn’t needed in this scenario we could just skip it because we have a default value (very confusing in lecture as well).



To use the first way we are going to set the AuthContext data directly on the component

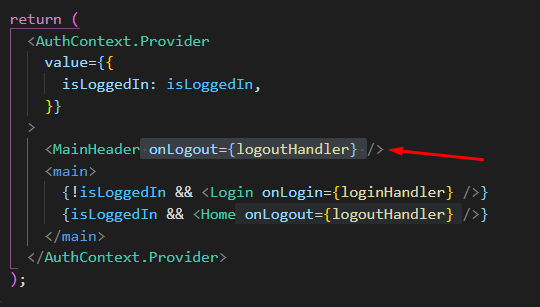
## Tapping into Context with the useContext hook

We will remove the consumer from our child component

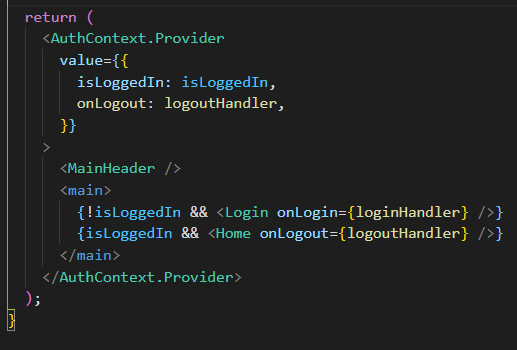


## Making Context Dynamic

It would be cool if we could just use <MainHeader without any props because MainHeader just passes props down to our navigation component.



But we are in luck because we can use dynamic context to pass down functions.

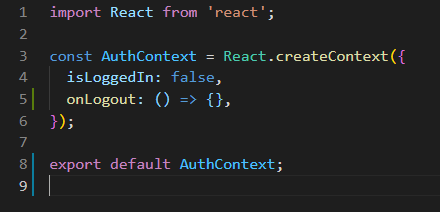


Here we also pass down the logooutHandler and now we don’t need props.

Now using props is still the default way of passing down data to other components but if we have to pass them trough multiple components then it is recommended to use props.

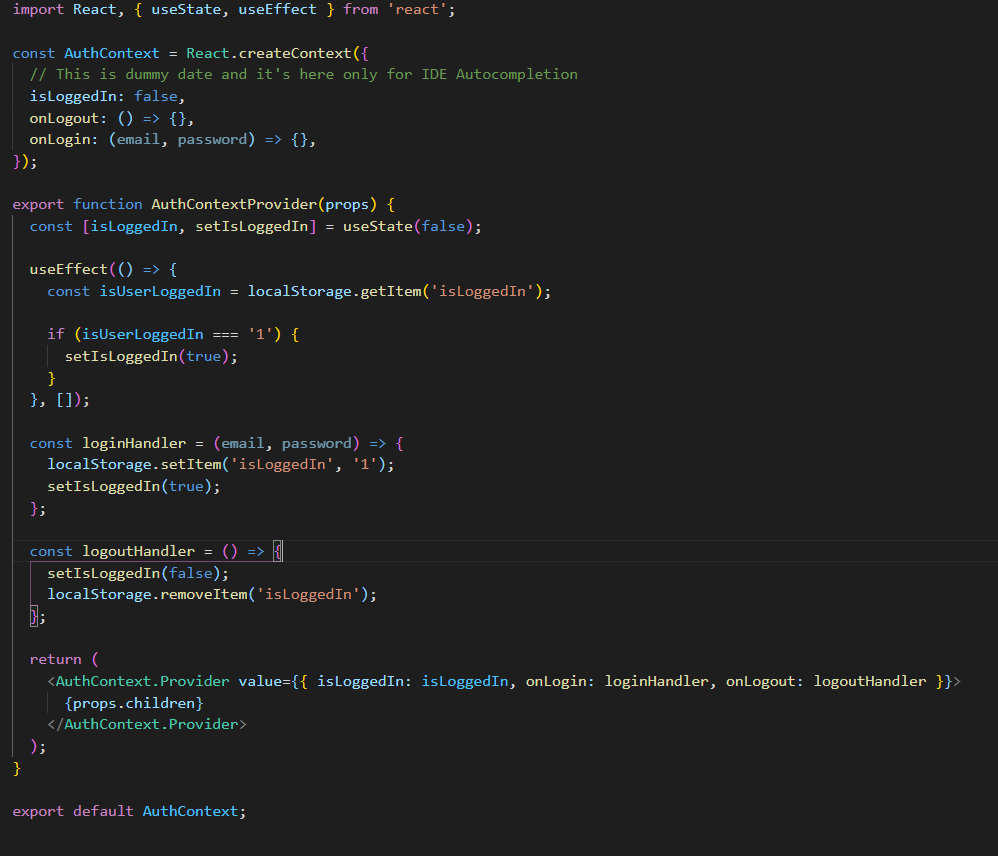
## Building and using a Custom Context Provider Component

If you are using functions in your context it is recommended to also add your functions as dummy functions to your default context when creating them. (This is also there for better IDE autocompletion)

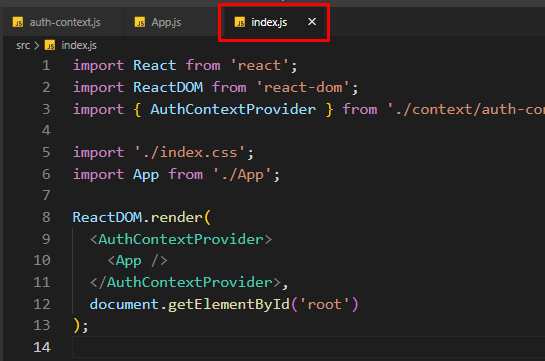


Refactoring our whole code to show another way of managing state in React.

We move all our state managing code into the AuthContext file

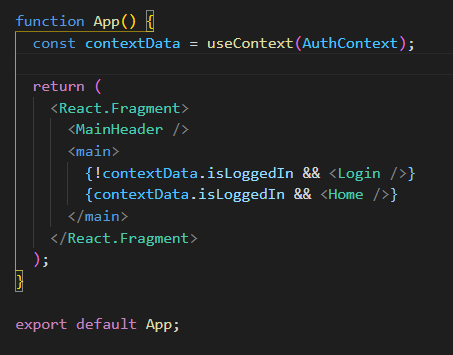


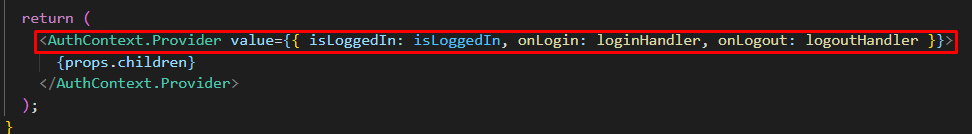
Now, this is not a mandatory way of separating code but it can be an option. In order for this to work we are going to wrap the whole App.js component with AuthContextProvider



And then we can just invoke

Const contextData = useContext(AuthContext);



Remember the data comes from the auth-context js file where we wrap all the dynamic data in the value prop

## React Context Limitations

It can be great for App wide, component wide state but it’s not a replacement from component configuration. Take the following example with a reusable button component



We could setup a button component with Context but that means we cannot use a button for anything else because we can have only one onClick value and that might not be what we want.

Things to note :

* React context is not optimized for high frequency changes (changes in seconds)

There is a better tool for component wide high frequency changes called redux in which we will dive in later

* React context also shouldn’t be used to replace ALL component communications and props

## Learning the rules of hooks

React hooks are all functions that start with use

Rules :

* Only call React hooks in React functions or custom Hooks (more later on this)
* Must only call React hooks at the Top Level of React functions
* For useEffect make sure you ALWAYS add everything you refer to as a dependency unless there is a good reason not to do that example the function of a useState hook.

## Refactoring the input component

We will dive into another hook but in order to do this we will refactor a component



This is why we need props because the props can be configured inside the parent where we use this component. Whereas with context we can only have a variable with useContext() and we cannot change it only use it

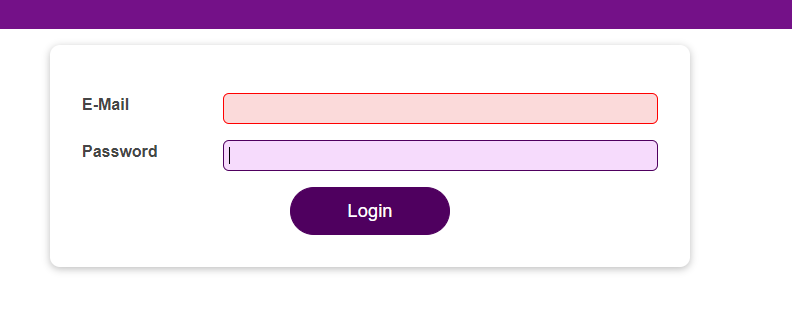
## useRef example in our input component



Take the example above, here we have our refactored input component where we called the useRef() hook so we can point at the DOM element of the input component and then we used useEFfect(which runs after the component is rendered) and used the current.focus() method which focuses the input field.

To note that the ref is supported on all built-in HTML components

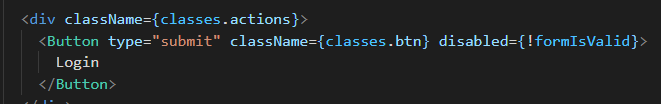
Now this is not the behaviour we want to achieve but it can be useful sometimes and it is good for demonstration purposes



Notice how our last input component is focused. The behaviour we actually want is that when we click the login button focus the first invalid input field.

## Diving into “Forward Refs”

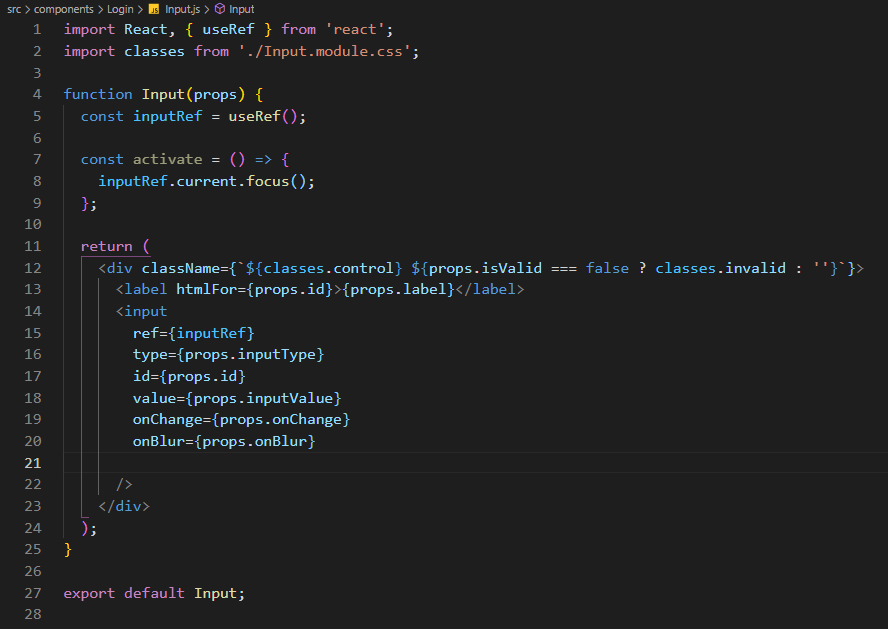
The next hook allows us to interact with the input components imperatively which means not by passing some state to it which changes something in the component but by calling a function inside a component which is something that we shouldn’t do often because it is not the typical react pattern but it might come in handy.



Lets remove disabled from button and add a logical check in the submitHandler if some form is invalid and login is clicked then focus the first invalid input



Now we do some mods in our Input.js file as well



The code above won’t work, we have to add some more stuff in order for it to function the way we want it.



Notice that we added 3 things.

First we transformed our function declaration to a function expression to make it easier to use React.forwardRef. We need this React method for this to work.

We added another argument to our functional component which is available on all functional components and it is the ref argument. We won’t be using this 99.9% of the time but this is a good use case.

Now in order to connect this ref argument to the ref of the used component we have to use the useImperativeHandle method which is actually making the connection between this component and where it is used.

This method takes in 2 paramenters. First the ref and the second one is a function which should return all the data we want to use in our parent component, in our case the activate function.

**We shouldn’t use this method at all but this is a sole example where we could use it.**

**Why not just add a ref on each input component and call focus on them?**

Because we cannot add refs on functional components.

# Practice Project: Building a Food Order App

## Module Intro

Here we are going to create a food order app where we will use all what we learned until now.