# 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.

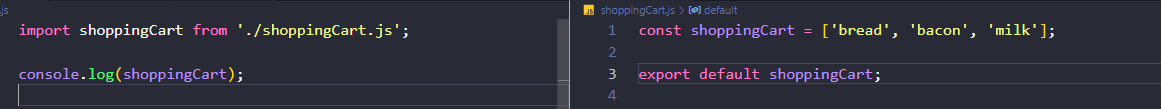
## Arrow functions

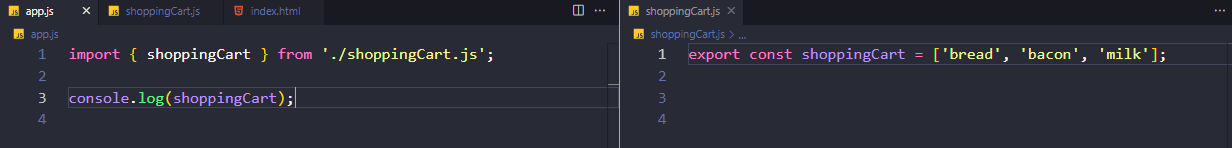
Function logMyName = (name) => console.log(name);

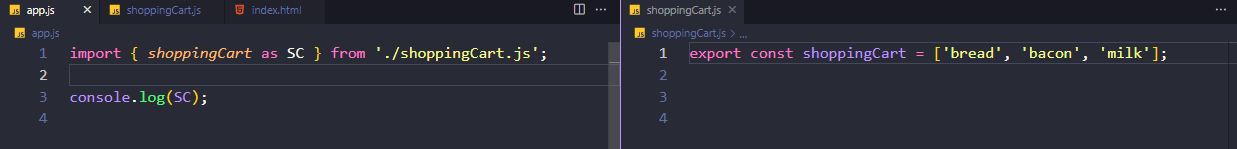
Arrow functions don’t have their own this keyword

## 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  


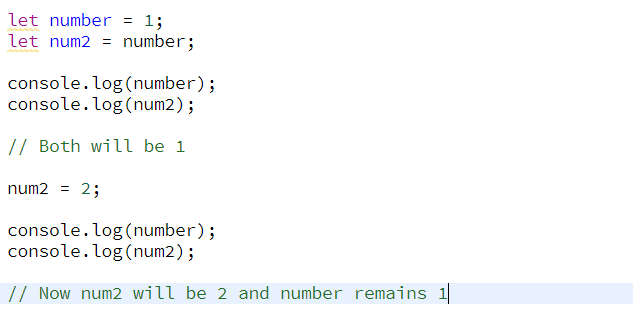
Don’t forget, in order for the modules to work we have to write type=’module’ in the script tag from the html file

## Classes Properties and Methods

Come back to this.

## Reference vs Primitive Types

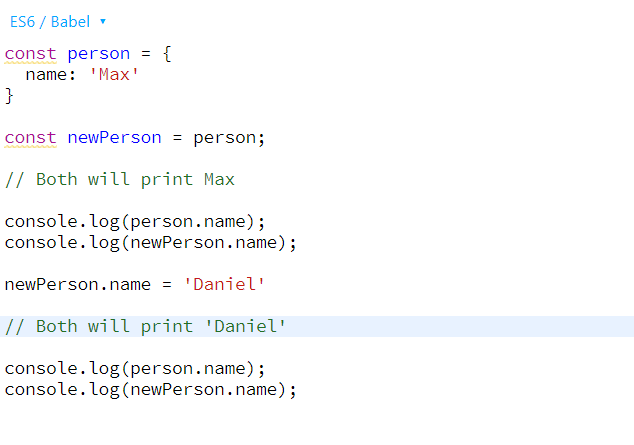
When we copy primitive types it will create a real copy of that primitive.



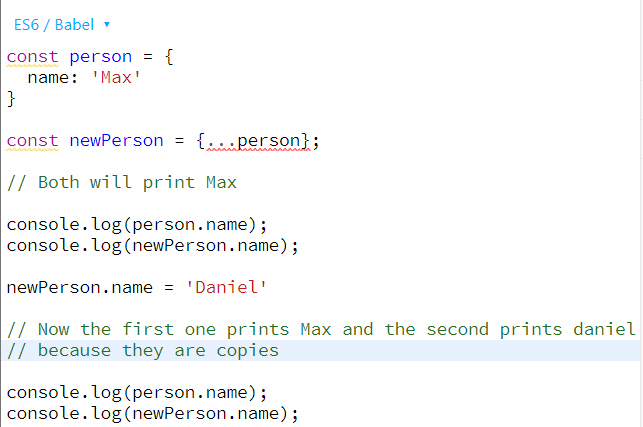
The two will be different because they are primitive types.

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

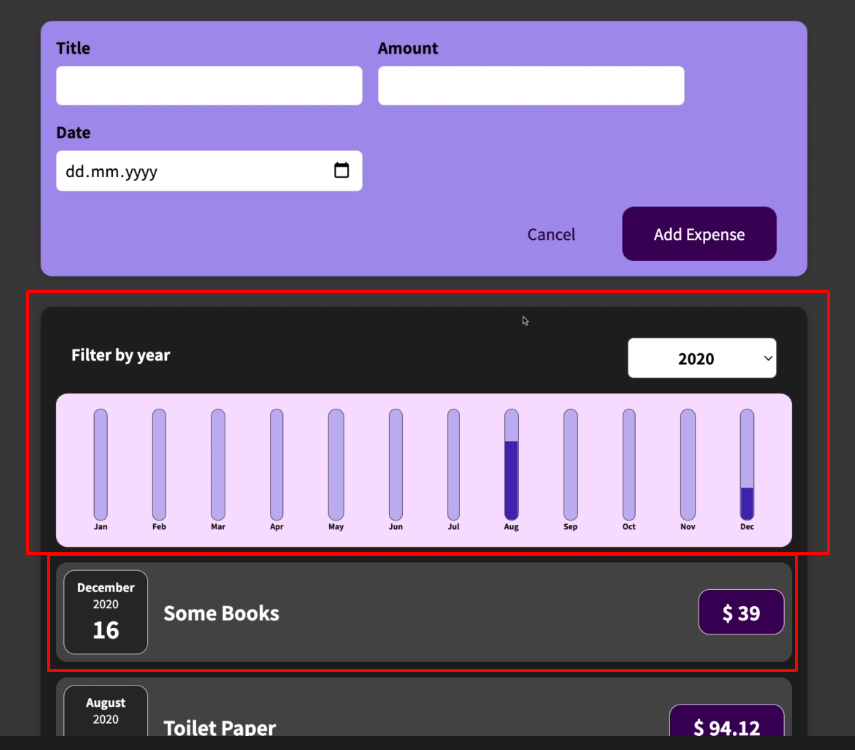
We will learn how about basics and most importantly how to build component driven user interfaces.

## 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?

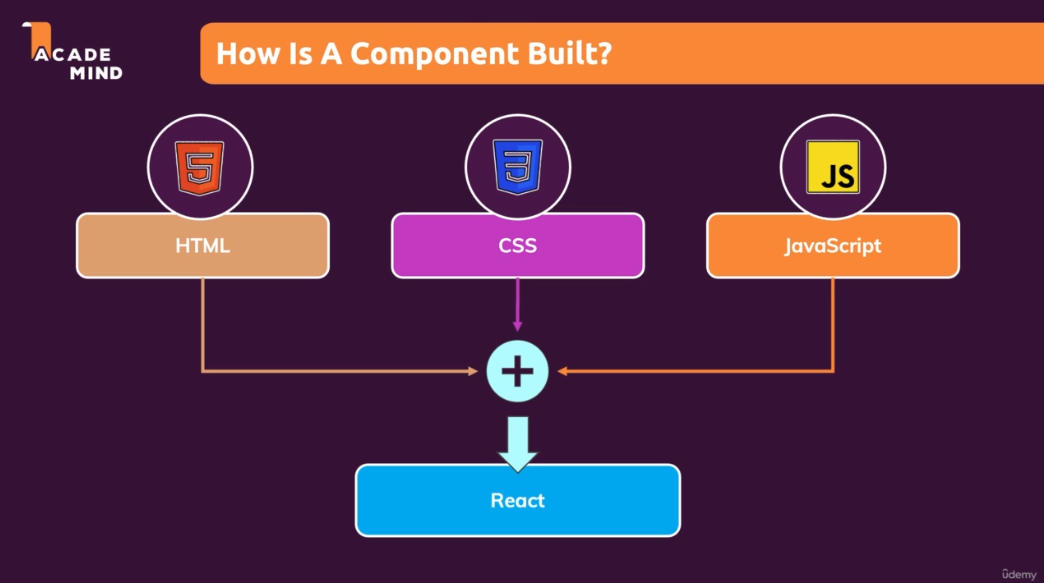
For reusability => for DRY Principle

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.

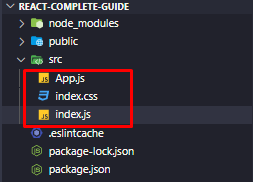


This is all the command line code we need to setup a new react app.

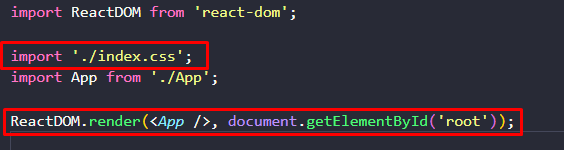
However for the project we used a simplified version of this which removes all the bloat.

## 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.

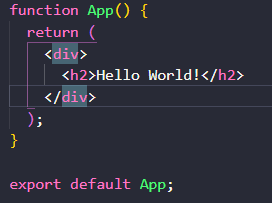


In this index.js file there is some weird code that in normal JS wouldn’t work for example importing a .css file into a JS file or the last line where we run some sort of HTML code in JS.



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.

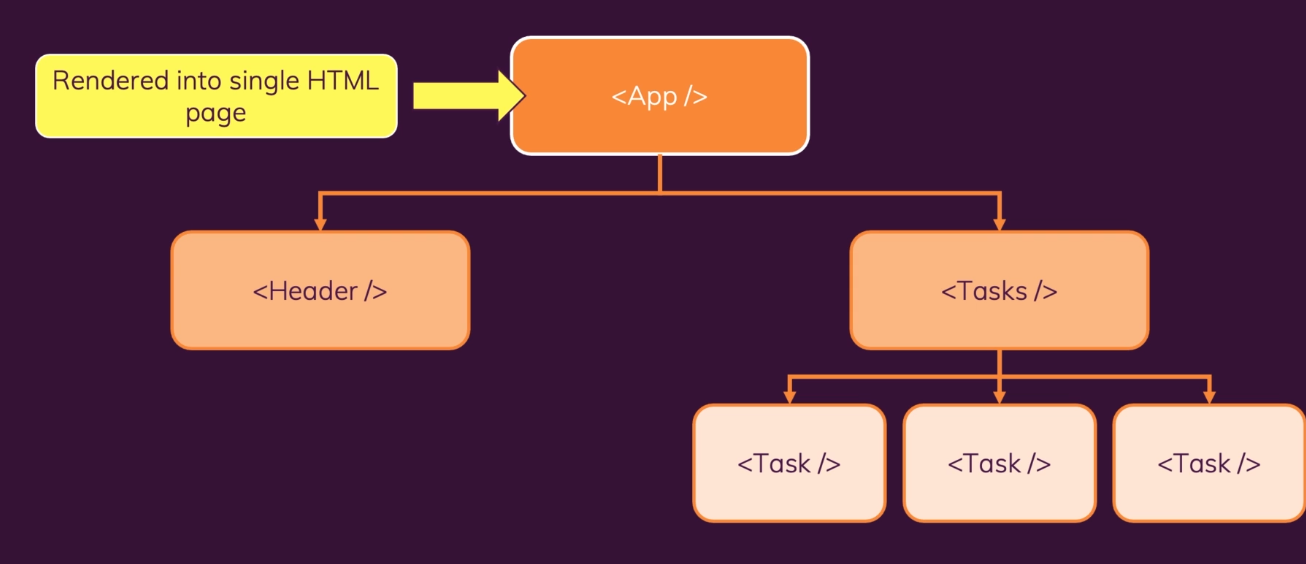
## What is JSX?

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

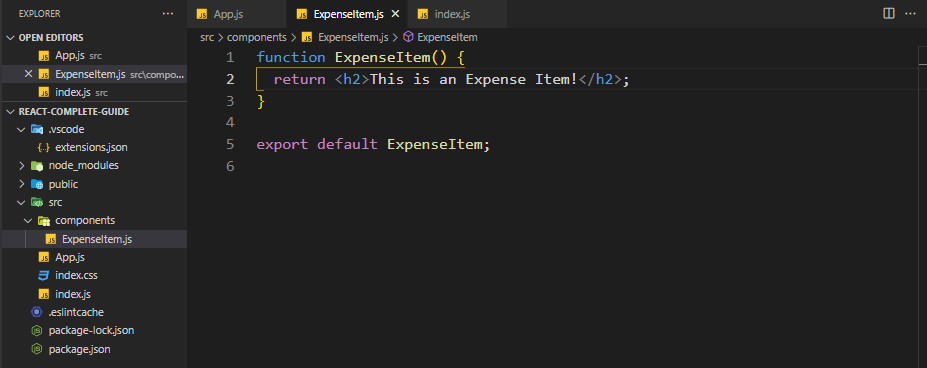
## Building our first custom component

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).

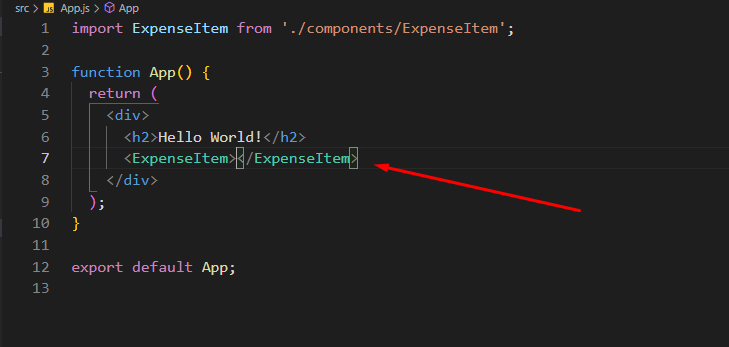
Ultimately with react we create a component tree



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.

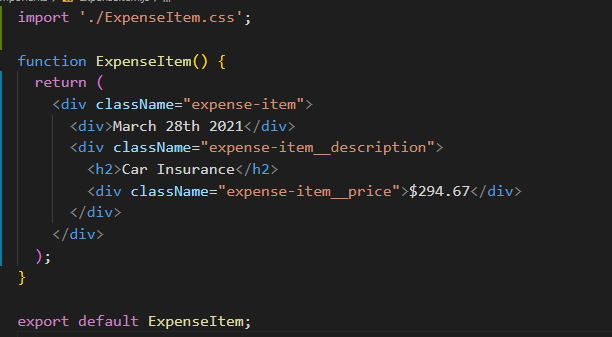


In React we are not allowed to have 2 root divs. Take the example below:



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.

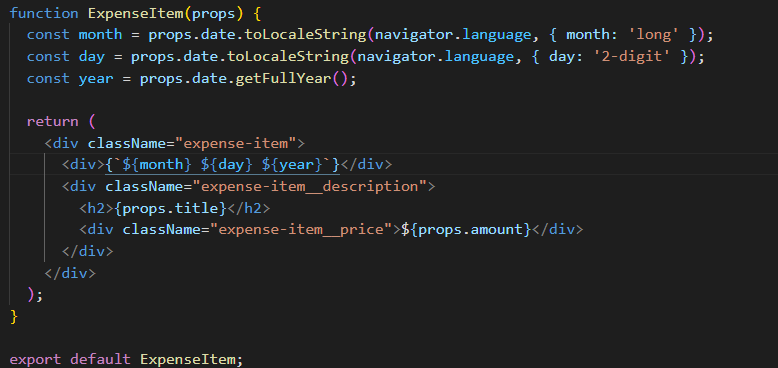
## The key which you access in on your prop object has to have the same name as you picked for your new component attribute.

## 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

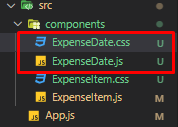
It is a good practice to separate the logic from the return function so we can format the date inside our function and in the return statement just enter what we formatted. I’ve used toLocaleString instead of internationalization API here because that’s what the teacher used in the course.



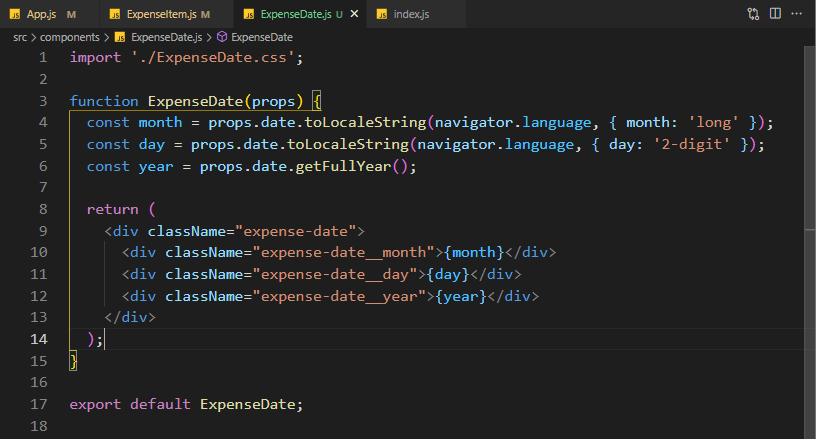
## Splitting Components into Multiple components

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.

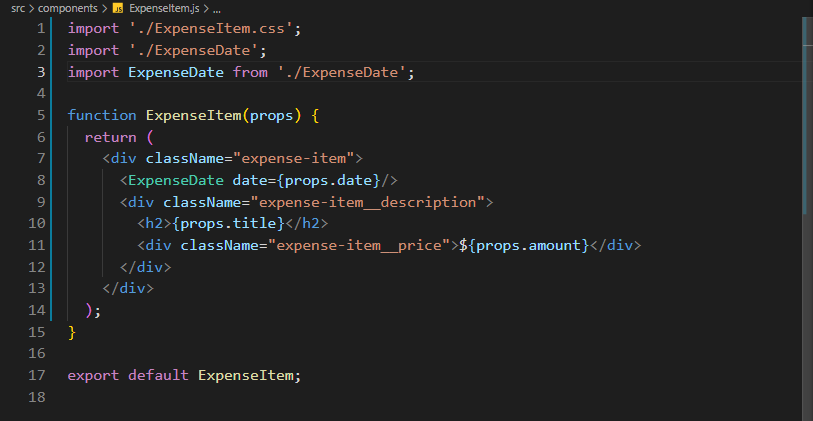
So like we did before we create a new JS and CSS File



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

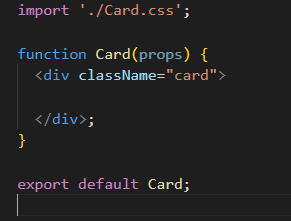
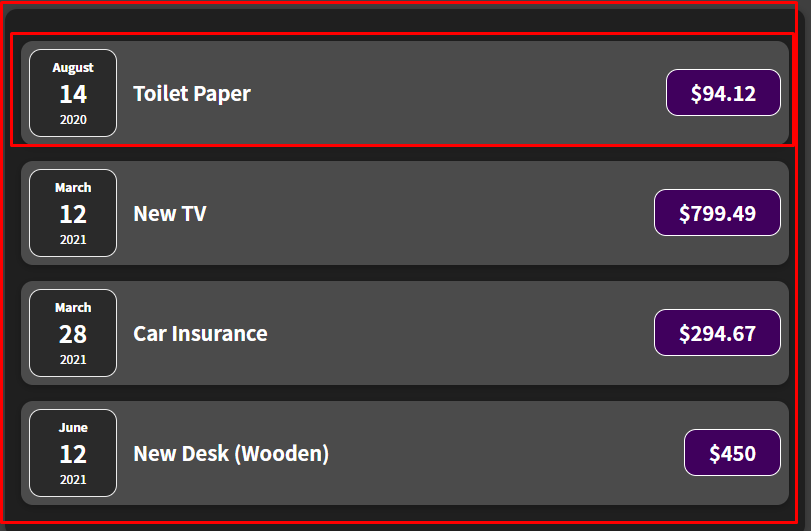
## The concept of “Composition” (“children props”)

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

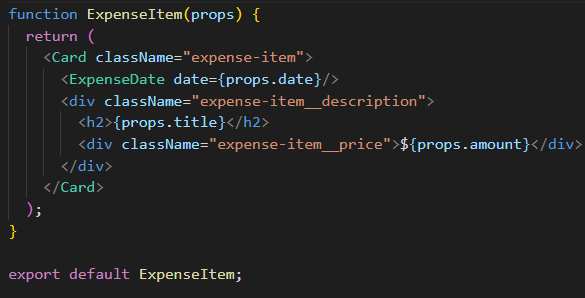
Expenses used the ExpenseItem component which uses the ExpenseDate component

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



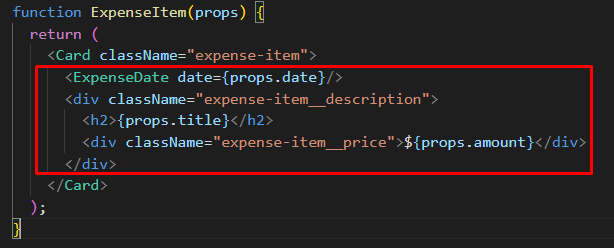
This is our card component which is a shell for now so the next logical step would be to replace the outer divs with the card component in the ExpenseItem and Expenses



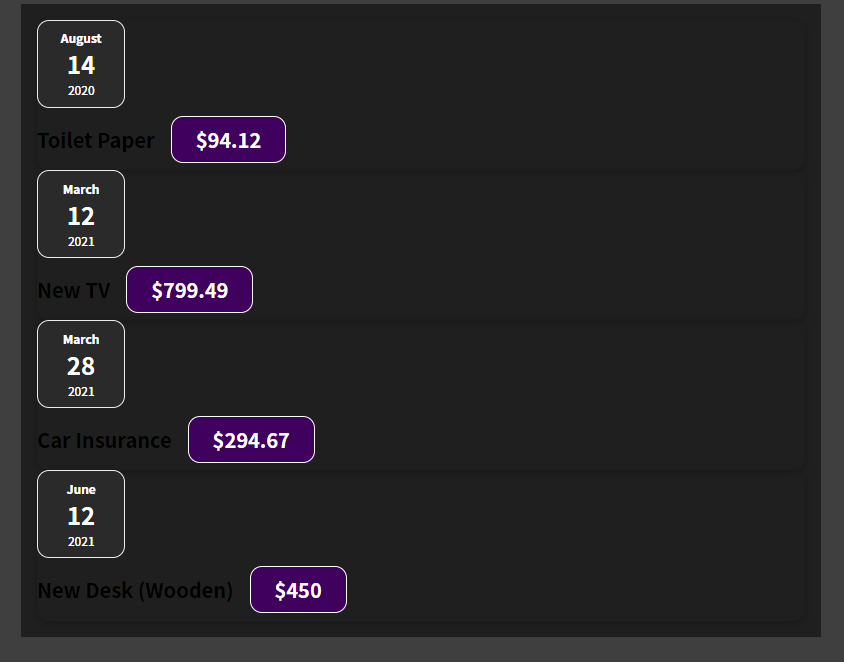
This wouldn’t work like this for custom HTML elements like our card the solution is to use props.

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

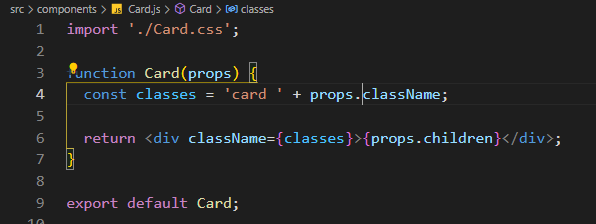


Adding this will somewhat fix our code but it will be still weird



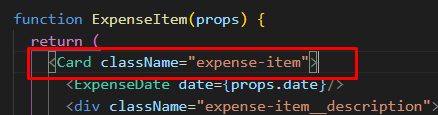
This is because we extracted some CSS but not all, our default HTML elements support classes out of the box but our custom components only support what we tell them to support.

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

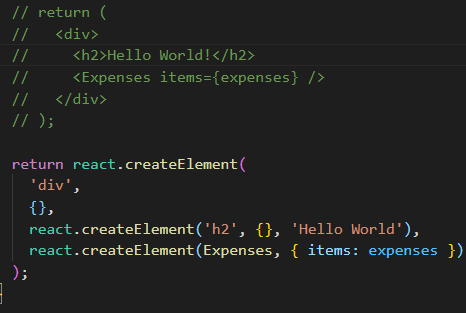
In older react projects we had to use import react from ‘react’ in every file. Now we don’t have to anymore it gets done automatically

Below is what is happening under the hood of JSX

First argument is element/custom element name.

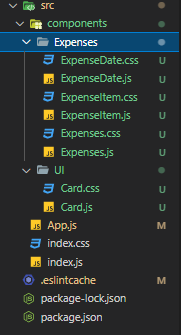
Second argument contains any attributes

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



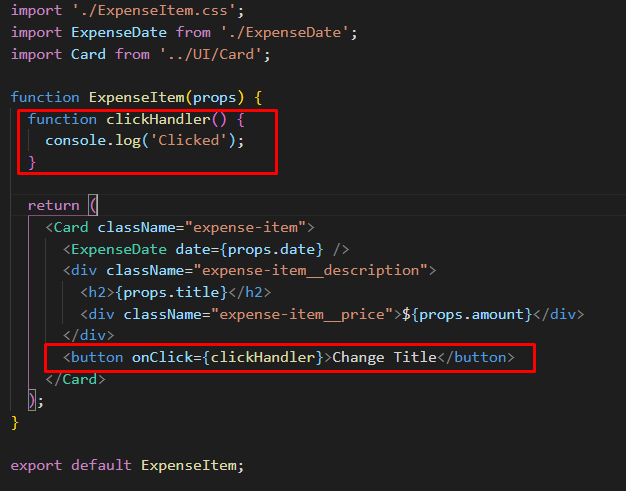
## Organizing Component Files

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

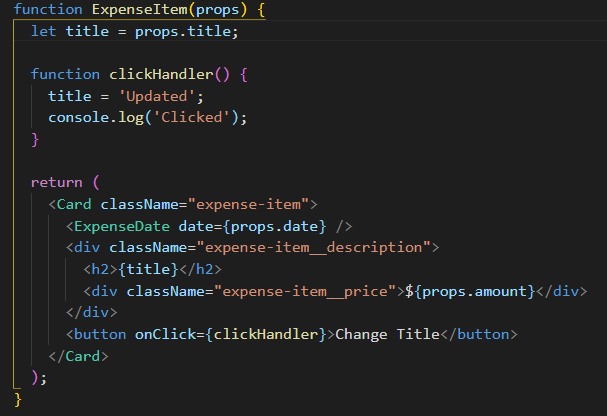


Adding event listeners is fairly simple just add onClick on the button. Since the button element just inherits from HTML Element we can use all sorts of properties and methods on it and all of this are available in react as well

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

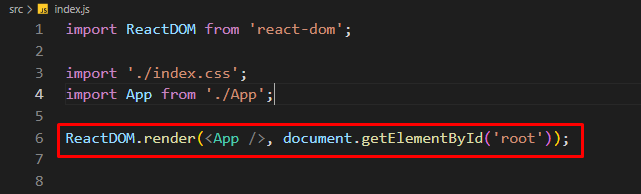


Using this example won’t work because this is not how react works. We have to dive in how react parses the JSX code and how it brings it on to the screen to know this.

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.

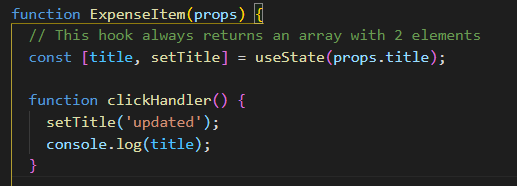
import { useState } from 'react';

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.

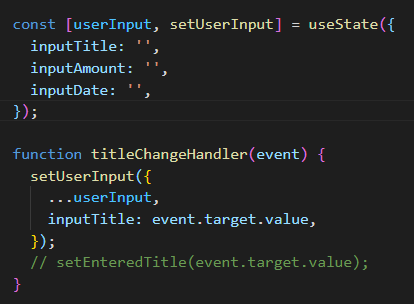
You can have multiple states per component and in this example we are going to use useState(‘’) with an empty string because by default it will be an empty string, in the example before we used useState(prop.title) which was the initial state of the prop



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

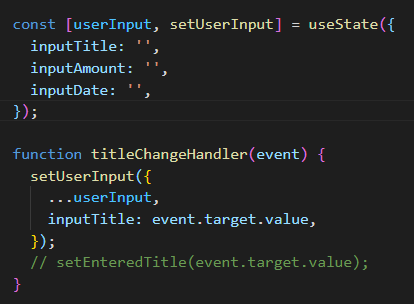


To note that every time we set the user input for one of the inputs we have to copy the other properties with destructuring otherwise it will return an object only with one property

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

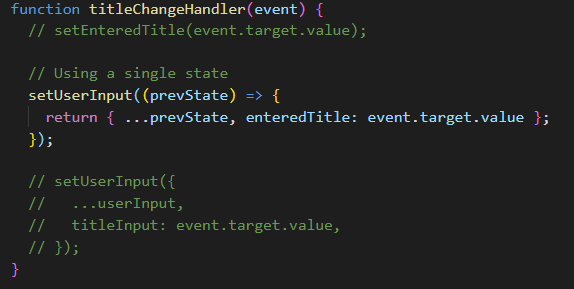
## Updating state that depends on a previous one

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

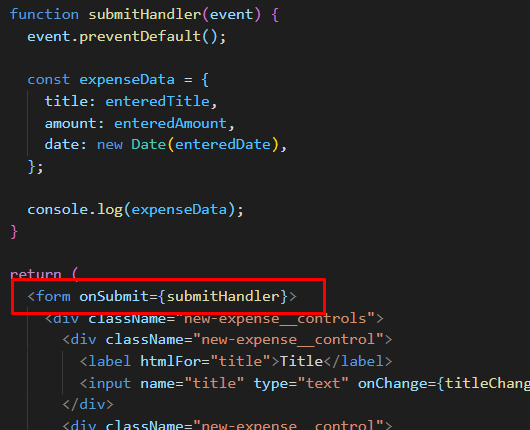


**Now why should we do it like this instead of the previous example? In many cases both will work fine but keep in mind that react schedules state updates and therefore if theoretically if schedule a lot of updates at the same time you could be depending on an outdated state. So by using this method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



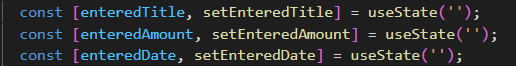
## Adding a Two-Way Binding

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.

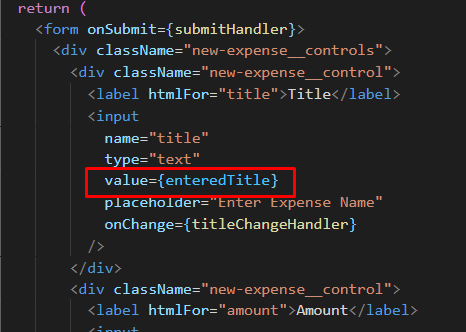
Now we can implement something called a 2 way Binding which simply means for input we don’t just listen to changes but we can pass a new value into the input to reset or change the input programmatically

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.

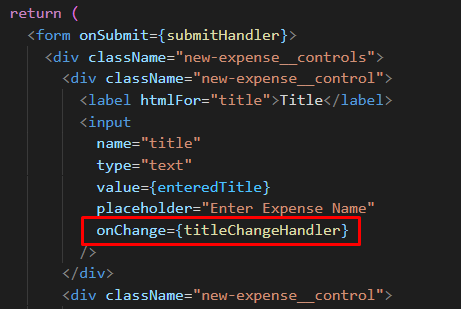


## Child-to-Parent Component Communication (bottom up)

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.

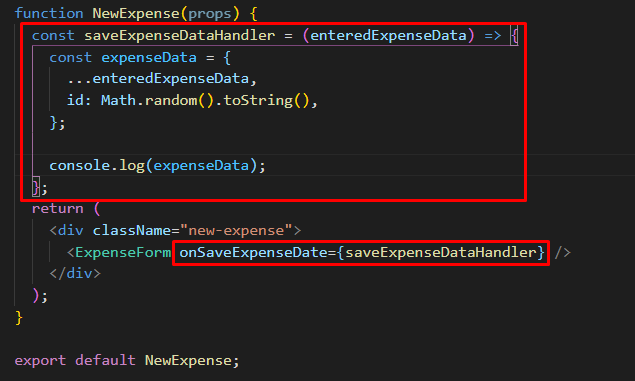
Now we know how to pass data to children by using props but how do we do it in the other way? We already used this but without knowing it.

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



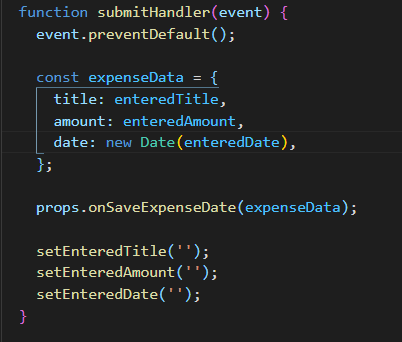
Whenever the user types then titleChangeHandler executes and there we get the default Event object which the browser gives us now we can think about the input element as a component as well, it is not our a component but a prebuilt one provided us by react and translated to the inputDOM element. We also 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.



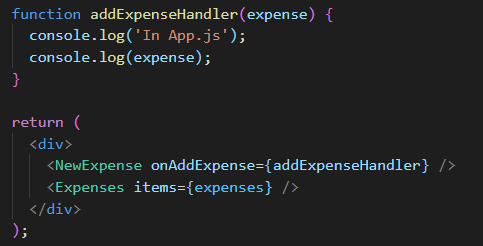
This is one of the parent components that has the prop onSaveExpenseData which passes on the function saveExpenseDataHandler, now this props function expects a parameter called enteredExpenseData. It will copy this enteredExpense data into a new object and add an ID on top of that.

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

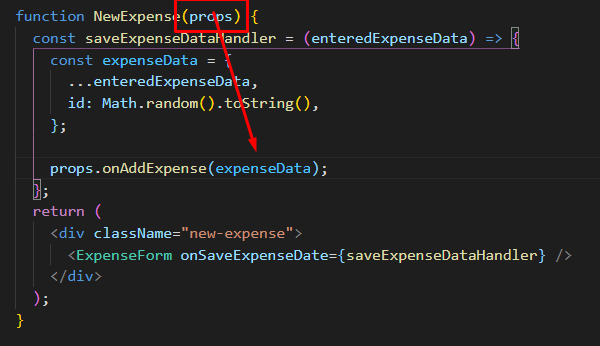


Now the last step is to get the data in the App parent component.

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



And now we use that prop in NewExpense.js

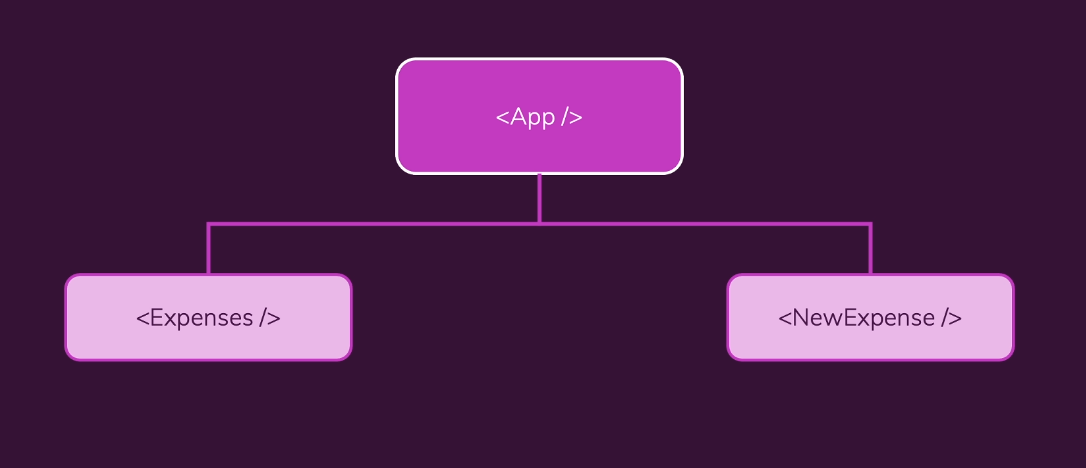


For now we just want to see if the data arrives in the parent app and later we will take care of updating the expenses list

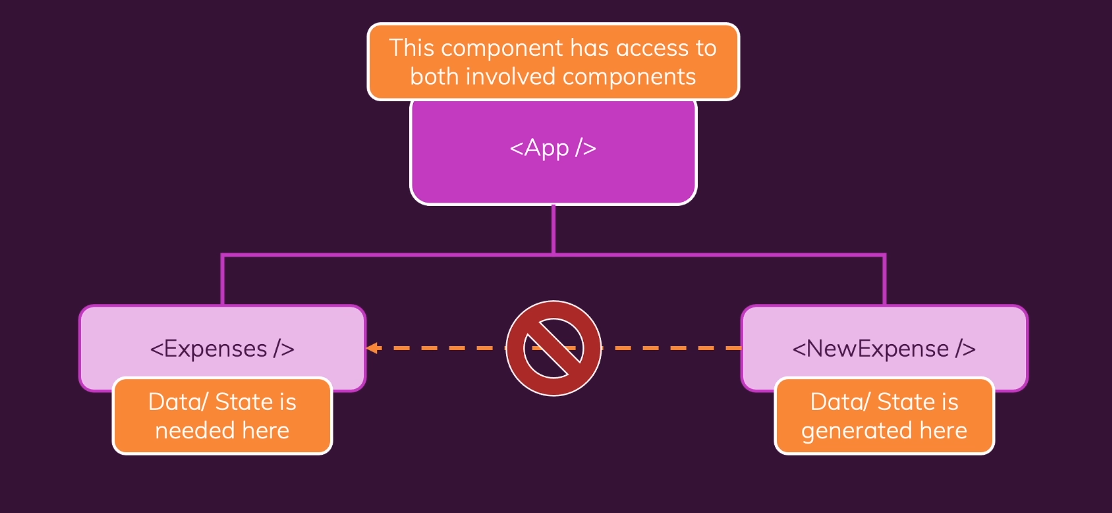
## Lifting the state up

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.

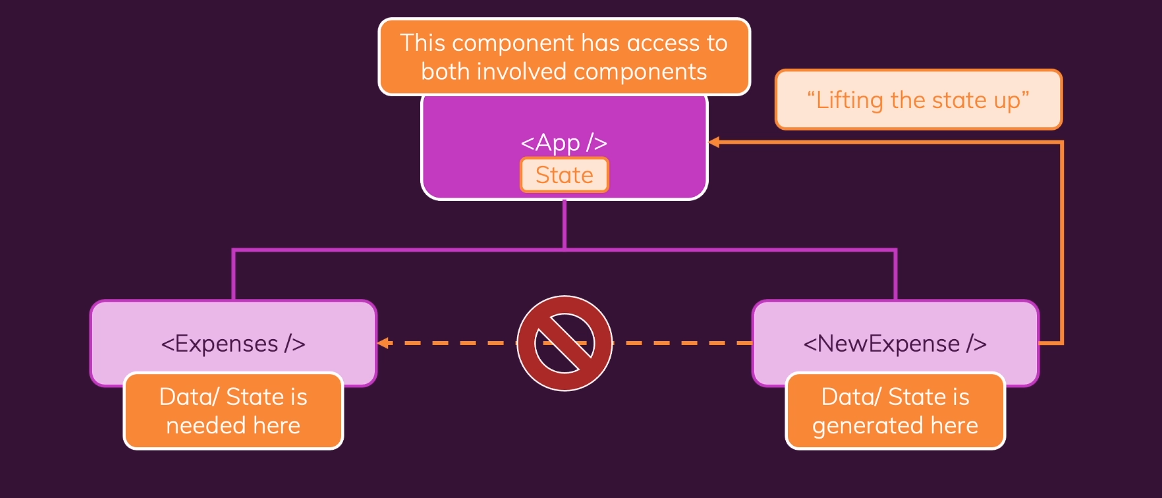
There is a concept called lifting state up. Consider the next tree



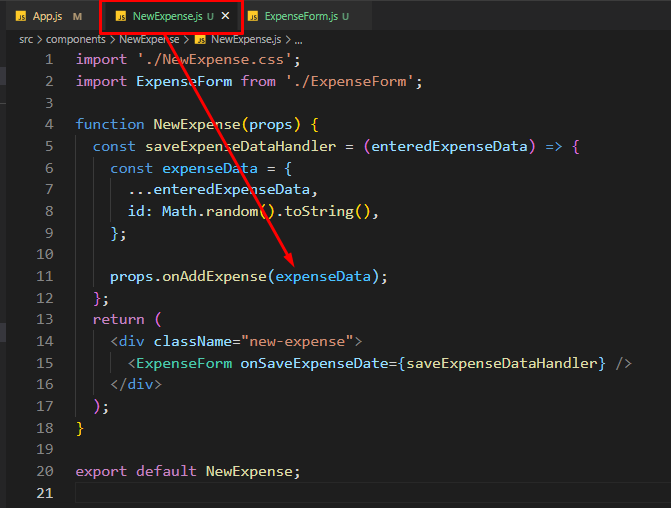
In this example the NewExpense component is a component that generates some data/state it is common that you generate data in a component but you might not need that data in that component (we don’t need the form data in our form, we need it in our expenses)



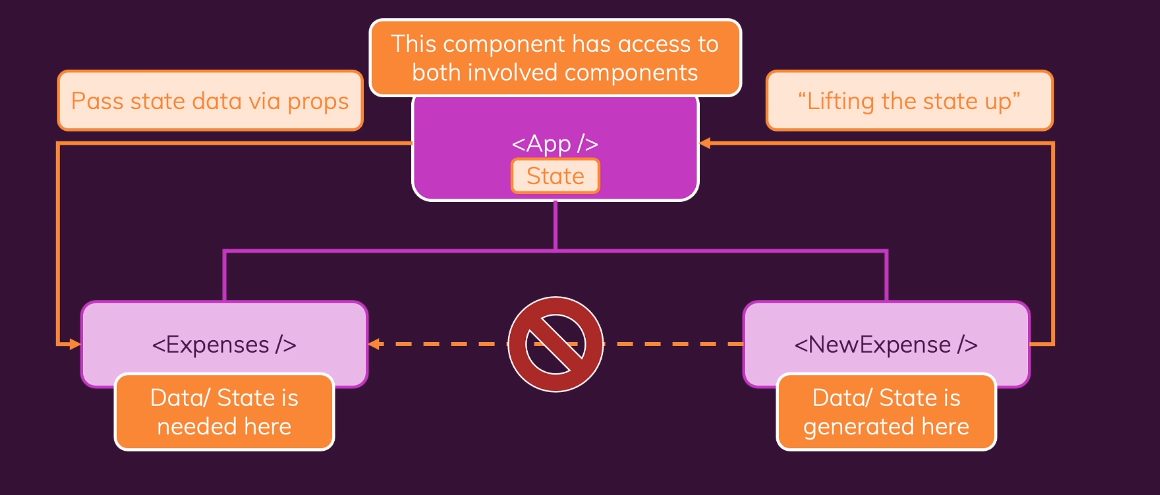
We want to utilize the closest common parent that has access to both components, in our case the App component. Now we can store our state in our App component by lifting our state up



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.