**REACT JS**

## **What is React Js?**

React is a JavaScript library that aims to simplify the development of visual interfaces.

Developed at Facebook and released to the world in 2013, it drives some of the most widely used apps, powering Facebook and Instagram among countless other applications.

Its primary goal is to make it easy to reason about an interface and its state at any point in time. It does this by dividing the UI into a collection of components.

You might experience some initial difficulties when learning React. But once it "clicks", I guarantee it's going to be one of the best experiences you ever have. React makes many things easier, and its ecosystem is filled with great libraries and tools.

React in itself has a very small API, and you basically need to understand 4 concepts to get started:

* Components
* JSX
* State
* Props

## 

## 

## **Why should you learn React?**

I highly recommend that any Web developer has at least a basic understanding of React.

That's because of a few reasons.

1. React is very popular. As a developer, it's quite likely that you're going to work on a React project in the future. Perhaps an existing project, or maybe your team will want you to work on a brand new app based on React.
2. A lot of tooling today is built using React at the core. Popular frameworks and tools like Next.js, Gatsby, and many others use React under the hood.
3. As a frontend engineer, React is probably going to come up in a job interview.

Those are all good reasons, but one of the main reasons I want you to learn React is that it's great.

It promotes several good development practices, including code reusability and component-driven development. It is fast, it is lightweight, and the way it makes you think about the data flow in your application perfectly suits a lot of common scenarios.

## 

**Declarative Views in React**

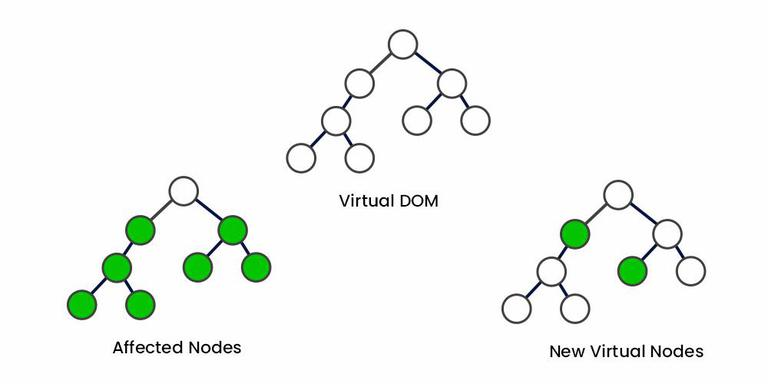
When we update the browser DOM (when not using React), it takes a significant amount of time to lay out the elements and repaint the screen for the user to see changes. This involves a huge section of the DOM tree being repainted.

But, when using React, updates happen in the virtual DOM first. Then, the browser DOM and virtual DOM are compared to see if there are any updates made to virtual DOM that must be reflected or updated in the browser DOM.

If there are any, only then are updates made to the browser DOM to match the virtual DOM. And these updates are only made at places where updates are to be carried out. This means that the entire browser DOM is not updated as previously mentioned. This improves speed and efficiency.

**What is the Virtual DOM?**

The virtual DOM (VDOM) is a programming concept where an ideal, or “virtual”, representation of a UI is kept in memory and synced with the “real” DOM by a library such as ReactDOM. This process is called [reconciliation](https://reactjs.org/docs/reconciliation.html).



**What is Rendering in React Js ?**

Render is the technique that can redirect a page with the help of function render(). Most importantly, render a function we can use to define the [HTML](https://www.simplilearn.com/tutorials/html-tutorial/what-is-html) code within the HTML element. It helps to display certain views in the UI using certain logic defined in the render function and returns the output.

**Purpose of render():**

* React renders HTML to the web page by using a function called render().
* The purpose of the function is to display the specified HTML code inside the specified HTML element.
* In the render() method, we can read props and state and return our JSX code to the root component of our app.
* In the render() method, we cannot change the state, and we cannot cause side effects( such as making an HTTP request to the webserver).

**Lifecycle of Components**

Every React Component has a lifecycle of its own, lifecycle of a component can be defined as the series of methods that are invoked in different stages of the component’s existence.

A React Component can go through four stages of its life as follows.

**Initialization**

In this phase, the developer has to define the props and initial state of the component this is generally done in the constructor of the component.

**Mounting**

This is the next stage in the lifecycle and a critical one for launch. After you have prepared the code with basic requirements, states and props, you need your component to mount in the browser. This is done via browser DOM, and the phase gives you the right ReactJS hooks methods for a before and after fitting. Here are the critical terms that you should be adept in.

**“componentWillMount”**

This is a critical function to remember as it is executed just before the reach component is about the mount. The mounting on the DOM is done after this stage, wherein you can enter all the things that you want to the program to do. It is also executed once in a life cycle of a component and occurs before you render the program for the first time. It is also used for initialising the states or props as well, making it a robust component to leverage.

**“componentDidMount”**

This is the final ReactJS hook method that is executed after the component mounts the DOM. It’s also performed once in the lifecycle and occurs after the first rendering. Engineers can access the DOM via this method and initialise the appropriate [JS libraries](https://www.cuelogic.com/blog/polymer-vs-react-comparison-between-two-front-end-javascript-libraries). You can access the DOM efficiently using this component. You can also initialise using several other libraries that can be incorporated into the final output. You can also make the right API calls under this method so that you can retrieve the data the right way.

**Updating**

Developers can typically update the component in a few main ways. They can either send new props to the command or update the state entirely. Depending on the complexity or the scale of work, they can choose either method and get the program running.

**“shouldComponentUpdate”**

The method tells the program about the state of rendering when it is updated. If new props or rules are being updated, then a rendering can be done or skipped. This is important to code in properly as there are evolving states in the program as well. Updating the method as true/false is the proper approach. The default here is true, which can be changed as per the code.

**“component will update”**

This is executed when the prior method returns the answer of true. It’s then used to prepare the upcoming render, in the case where some previous calculation is necessary before returning a response. For more complex programs, this method can be used as well.

**“componentDidUpdate”**

This is then executed when the updated component has been updated in the DOM as well. You can then initiate new libraries to reload as well so that you can maintain an updated program throughout the process.

**Unmounting**

The final stage of unmounting is essential as it doesn’t require the component and gets unmounted from the DOM. As the final state, it is designed to produce the outcome via unmounting.

**“componentWillUnmount”**

This is the last method in the lifecycle as it pertains to the core unmounting and removal from the DOM. The cleaning up of the component is also performed here. This is also used in the logging out of users when they want to clear out the program from their browser.

## **How to install React**

To start with, It is highly recommended one approach, and that's using the officially recommended tool called create-react-app.

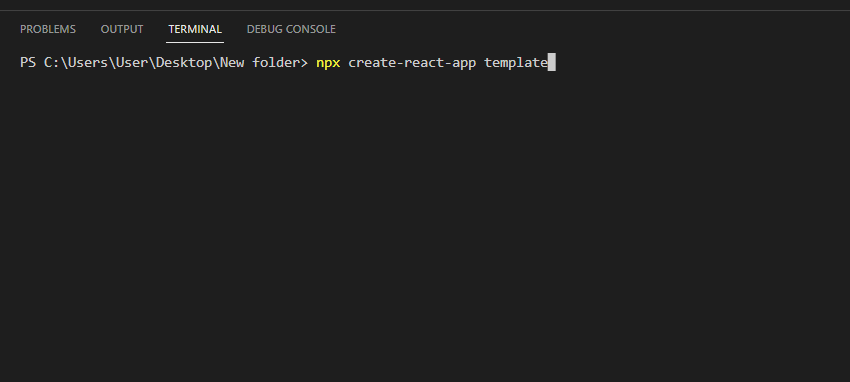
create-react-app is a command line application, aimed at getting you up to speed with React in no time.

You start by using npx, which is an easy way to download and execute Node.js commands without installing them.

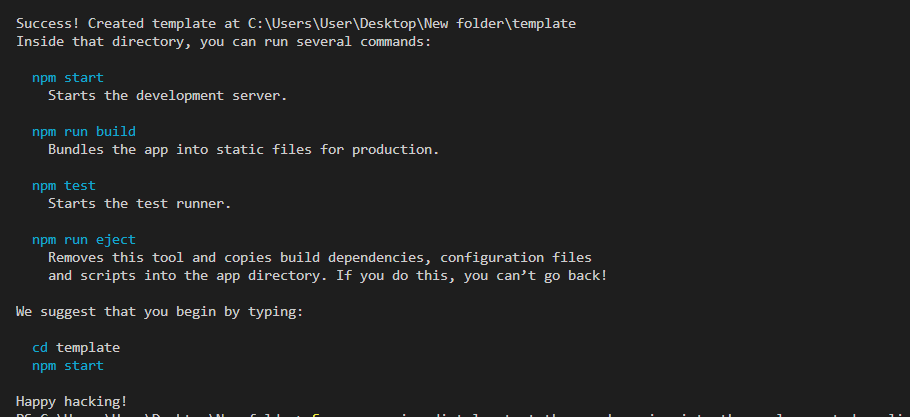
npx comes with npm (since version 5.2).

When you run npx create-react-app <app-name>, npx is going to *download* the most recent create-react-app release, run it, and then remove it from your system.

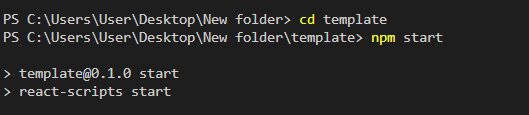
Let's start then:

****

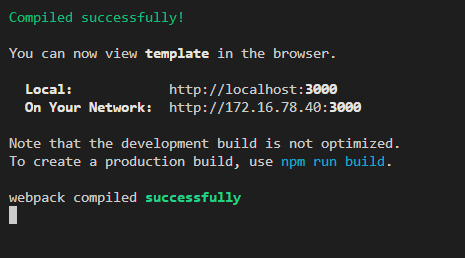
This is when it finished running:

****

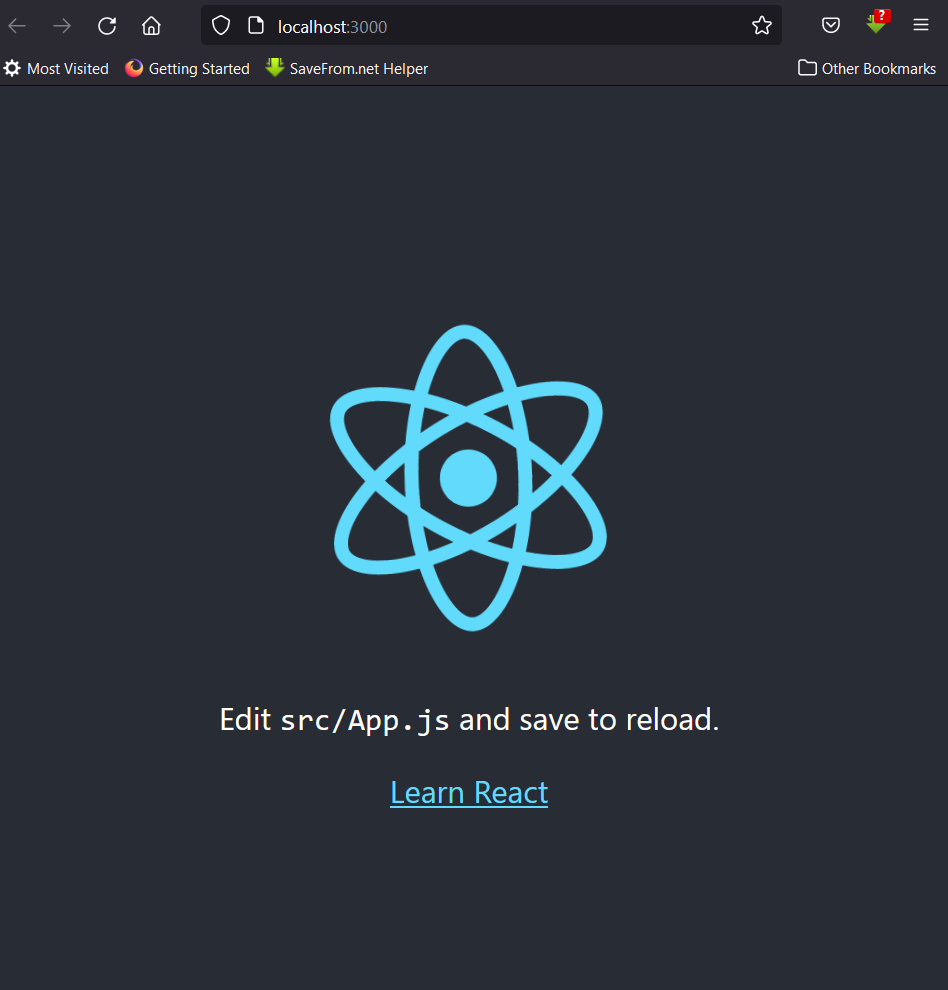
So you can immediately start the app by going into the newly created application folder and running npm start.



You will get following result:



By default this command launches the app on your local port 3000, and it opens your browser showing you the welcome screen:



Now you're ready to work on this application!

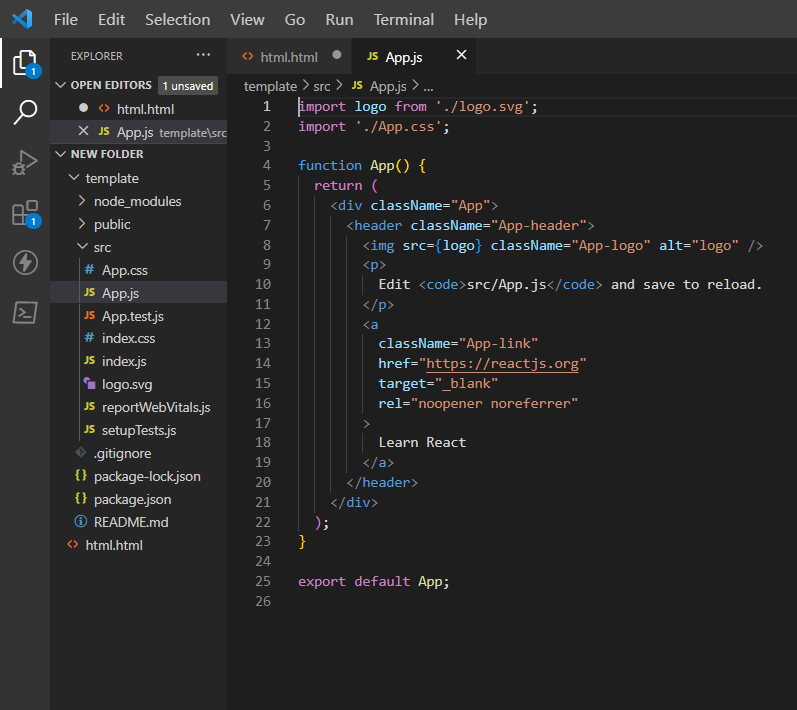
## 

## 

## **React Components**

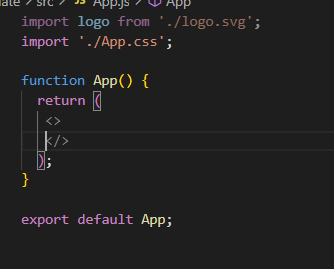
This application comes with a series of files that do various things, mostly related to configuration, but there's one file that stands out: App.js.

App.js is the first React Component you meet.



An application built using React, or one of the other popular frontend frameworks like Vue and Svelte for example, is built using dozens of components.

But let's start by analyzing this first component. I'm going to simplify this component code like this:



You can see a few things here. We *import* some things, and we *export* a function called App.

The things we import in this case are a JavaScript library (the react npm package), an SVG image, and a CSS file.

App is a function that, in the original example, returns something that at first sight looks quite strange.

It looks like **HTML** but it has some JavaScript embedded into it.

That is **JSX**, a special language we use to build a component's output. We'll talk more about JSX in the next section.

In addition to defining some JSX to return, a component has several other characteristics.

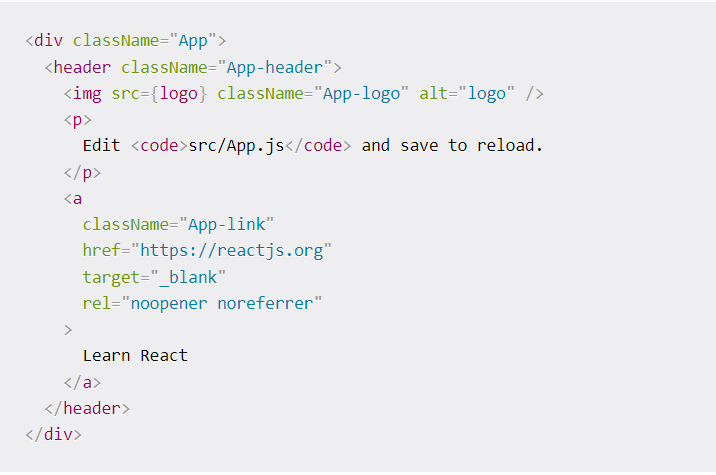
A component can have its own **state**, which means it encapsulates some variables that other components can't access unless this component exposes this state to the rest of the application.

A component can also receive data from other components. In this case we're talking about **props**.

## **Introduction to JSX**

We previously ignored everything that was inside the return statement,

We call JSX everything wrapped inside the parentheses returned by the component:



This looks like HTML, but it's not really HTML. It's a little different.

And it's a bit strange to have this code inside a JavaScript file. This does not look like JavaScript at all!

Under the hood, React will process the JSX and it will transform it into JavaScript that the browser will be able to interpret.

So we're writing JSX, but in the end there's a translation step that makes it digestible to a JavaScript interpreter.

React gives us this interface for one reason: it's easier to build UI interfaces using JSX.

## **Using JSX to compose a UI**

In particular, in a React component you can import other React components, and you can embed them and display them.

A React component is usually created in its own file, because that's how we can easily reuse it (by importing it) in other components.

But a React component can also be created in the same file of another component, if you plan to only use it in that component. There's no "rule" here, you can do what feels best to you.

I generally use separate files when the number of lines in a file grows too much.

To keep things simple let's create a component in the same App.js file.We're going to create a WelcomeMessage component:



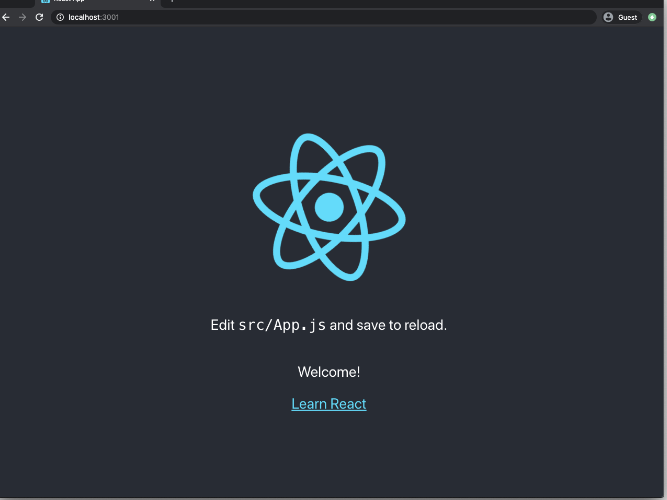
It's a simple function that returns a line of JSX that represents a p HTML element.

We're going to add it to the App.js file.

Now inside the App component JSX we can add <WelcomeMessage /> to show this component in the user interface:



And here's the result. Can you see the "Welcome!"



We say that WelcomeMessage is a **child component** of App, and App is its parent componnet.

We're adding the <WelcomeMessage /> component as if it was part of the HTML language.

That's the beauty of React components and JSX: we can compose an application interface and use it like we're writing HTML.

## **The difference between JSX and HTML**

One of the differences might be quite obvious if you looked at the App component JSX: there's a strange attribute called className.

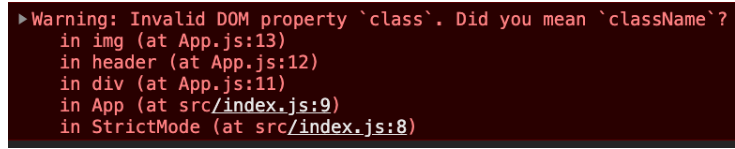
In HTML we use the class attribute. It's probably the most widely used attribute, for various reasons. One of those reasons is CSS. The class attribute allows us to style HTML elements easily, and CSS frameworks like Tailwind put this attribute to the center of the CSS user interface design process.

But there's a problem. We are writing this UI code in a JavaScript file, and class in the JavaScript programming language is a reserved word. This means we can't use this reserved word as we want. It serves a specific purpose (defining JavaScript classes) and the React creators had to choose a different name for it.

That's how we ended up with className instead of class.

You need to remember this especially when you're copy/pasting some existing HTML.

React will try its best to make sure things don't break, but it will raise a lot of warnings in the Developer Tools:

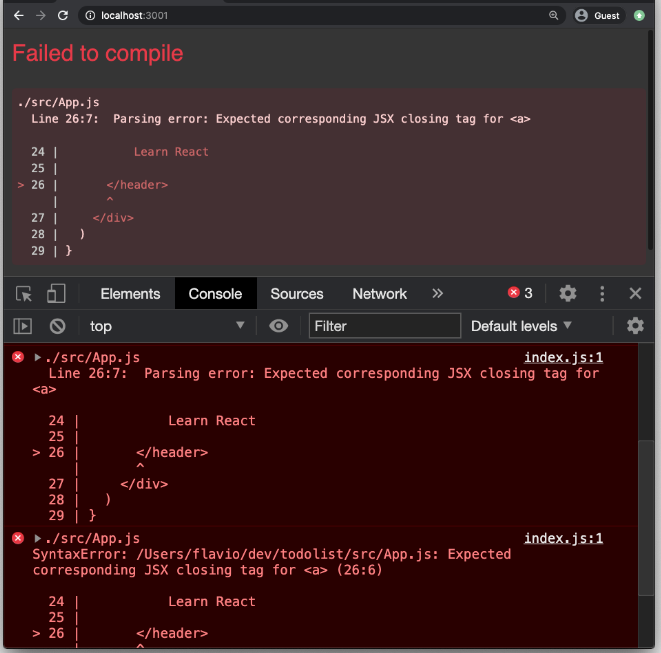


This is not the only HTML feature that suffers from this problem, but it's the most common one.

Another big difference between JSX and HTML is that HTML is very *relaxed*, we can say. Even if you have an error in the syntax, or you close the wrong tag, or you have a mismatch, the browser will try its best to interpret the HTML without breaking.

It's one of the core features of the Web. It is very forgiving.

JSX is not forgiving. If you forget to close a tag, you will have a clear error message:



***React usually gives very good and informative error messages that point you in the right direction to fix the problem.***

## **Embedding JavaScript in JSX**

One of the best features of React is that we can easily embed JavaScript into JSX.

Other frontend frameworks, for example Angular and Vue, have their own specific ways to print JavaScript values in the template, or perform things like loops.

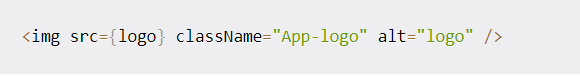
React doesn't add new things. Instead, it lets us use JavaScript in the JSX, by using curly brackets.

The first example of this that I will show you comes directly from the App component we've studied so far.

We import the logo SVG file using



and then in the JSX we assign this SVG file to the src attribute of an img tag:



Let's do another example. Suppose the App component has a variable called message:

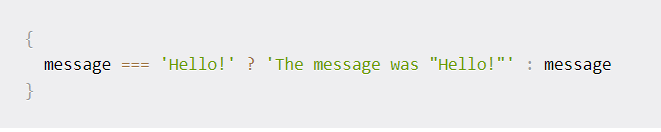


We can print this value in the JSX by adding {message} anywhere in the JSX.

Inside the curly brackets { } we can add any JavaScript statement, but *just one* statement for every curly bracket block.

And the statement must return something.

For example this is a common statement you will find in JSX. We have a ternary operator where we define a condition (message === 'Hello!'), and we print one value if the condition is true, or another value (the content of message in this case) if the condition is false:



## **Managing state in React**

Every React component can have its own **state**.

What do we mean by *state*? The state is the **set of data that is managed by the component**.

Think about a form, for example. Each individual input element of the form is responsible for managing its state: what is written inside it.

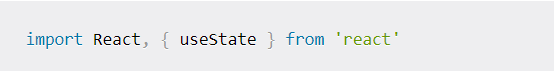
A button is responsible for knowing if it's being clicked, or not. If it's on focus.

A link is responsible for knowing if the mouse is hovering over it.

In React, or in any other components-based framework/library, all our applications are based on and make heavy use of components' state.

We manage state using the useState utility provided by React. It's technically a **hook** (you don't need to know the details of hooks right now, but that's what it is).

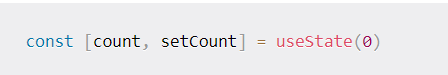
You import useState from React in this way:



Calling useState(), you will get back a new state variable, and a function that we can call to alter its value.

useState() accepts the initial value of the state item and returns an array containing the state variable, and the function you call to alter the state.

Example:

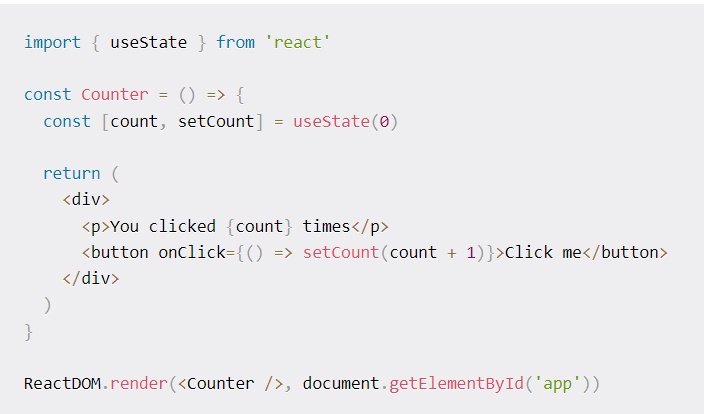


This is important. We can't just alter the value of a state variable directly. We must call its modifier function. Otherwise the React component will not update its UI to reflect the changes of the data.

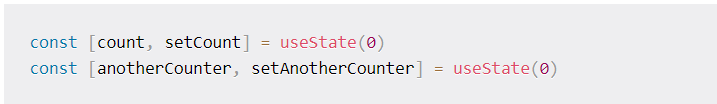
Calling the modifier is the way we can tell React that the component state has changed.

The syntax is a bit weird, right? Since useState() returns an array we use array destructuring to access each individual item, like this: const [count, setCount] = useState(0)

Here's a practical example:



You can add as many useState() calls as you want, to create as many state variables as you want:



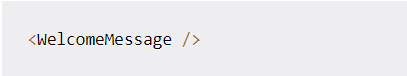
## **Component Props in React**

We call props the initial values passed to a component.

We previously created a WelcomeMessage component



and we used it like this:



This component does not have any initial value. It does not have props.

Props can be passed as attributes to the component in the JSX:



and inside the component we receive the props as arguments:



It's common to use object destructuring to get the props by name:



Now that we have the prop, we can use it inside the component. For example we can print its value in the JSX:



Curly brackets here have various meanings. In the case of the function argument, curly brackets are used as part of the object destructuring syntax.

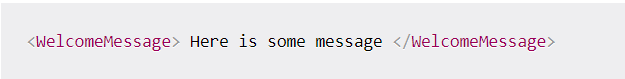
Then we use them to define the function code block, and finally in the JSX to print the JavaScript value.

Passing props to components is a great way to pass values around in your application.

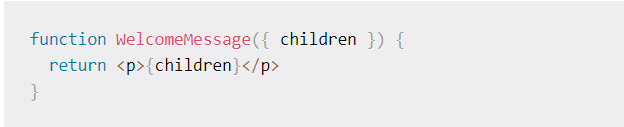
A component either holds data (has state) or receives data through its props.

We can also send functions as props, so a child component can call a function in the parent component.

A special prop is called children. That contains the value of anything that is passed between the opening and closing tags of the component, for example:

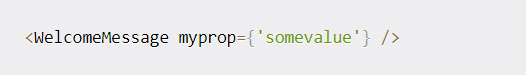


In this case, inside WelcomeMessage we could access the value Here is some message by using the children prop:

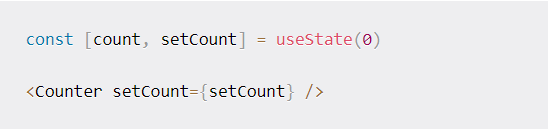


## **Data flow in a React application**

In a React application, data typically flows from a parent component to a child component, using props as we saw in the previous section:



If you pass a function to the child component, you can however change the state of the parent component from a child component:



Inside the Counter component we can now grab the setCount prop and call it to update the count state in the parent component, when something happens:

