React start-kit for beginners

Islas Carreon Victor Jakxel

3-august-2026

Índice

1. React get started

1.1. Introduction

Why I'm Doing This

When I first started learning React, I realized how hard it was to find good advice or simple guides to just get started with cool projects. Most of the time, I felt overwhelmed, and I wished someone had left a clear, beginner-friendly path to follow.

Sure, nowadays AI can do a lot of things — maybe even faster and better than what I'm doing here... sometimes. But that's not the point. I wanted to build something real on my own, step by step, and constantly improve my skills while doing it.

I'm not an expert yet — I'm literally still learning as I write this. But I'm doing this so that other people (maybe just like me) can have a better experience getting started with React and web development in general. If I learn and you learn too, that's a win-win situation — what more could we ask for?

Also, don't let AI do all the work for you. It might be fast, but often it gets things wrong. You can do better than that. Push yourself. Do the hard stuff. Study. Improve. Don't settle for just "vibing" through your goals — work hard and stand out.

Start doing things by yourself, like reading this or building something on your own. It doesn't have to be perfect. Just start.

1.2. Start a project

I don't know, but on different sites I've checked, they say it's good to have a bit of experience or at least basic knowledge in HTML, CSS, and JavaScript. Maybe before starting with React, you should watch some videos about that.

I recommend the videos from SuperSimpleDev — he has a really good pack of videos for learning those topics. Personally, those were the first videos I watched when I started learning web development, and I totally recommend them.

How to start a project

First, you'll need to set up your coding environment. In my case, I use **Neovim** as my editor, but you can use whatever you're comfortable with. Then, make sure you have **npm** installed. If you don't have it yet... what are you doing? Just install it already! You'll need it to create your project. Once you've got that ready, run this command to create a new project using Vite (I use Vite because it was the first thing I found and it just works):

```
npm create vite@latest name-of-your-project
```

Then, follow the prompts — I usually choose React, but you can pick whatever you want depending on the type of project you're building.

After that, go into your project folder, install dependencies, and start the dev server:

```
cd name-of-your-project
npm install
npm run dev
```

And that's it! Your project should be running locally — now you're ready to start building.

1.3. React environment

What are all this files?!!!

Usually, when I first started using React, my initial reaction was: "What is going on here?" Just a moment ago I was used to working with a maximum of three files — `index.html`, `style.css`, and `script.js`. Suddenly, React throws a whole bunch of files at you and my brain couldn't process it. I almost quit right there.

But once you take the time to look around and understand things step by step, it's actually pretty simple.

Most of the important stuff happens in the `src` folder. That's where your actual project lives. For now, that's really the only folder you need to focus on.

Inside `src`, you'll find two main files: `main.tsx` and `App.tsx`.

- `main.tsx` is where React is connected to the `index.html` file in the `public` folder — it basically tells React where to display your app.
- `App.tsx` is where the main component of your app lives. Think of it like your starting point — the "home" of your React app.

Once you understand that, you'll eventually want to create new folders like `components/` to organize reusable parts of your UI, or `pages/` if your app has multiple pages. You can also add folders like `styles/`, but don't worry about that just yet.

You'll also see folders like `public/` and sometimes `assets/` — these are usually for things like images or static files. For now, they're not super important.

Now that you know what these files are for, you can move around the project more confidently and start building cool stuff!

1.4. React and ES6

While I was looking for good advice or methods to learn the first steps in React development, I realized something: it's surprisingly hard to find clear and helpful study guides. I'm not sure why, but a lot of tutorials out there either overcomplicate things or explain stuff you don't really need when you're just starting out.

That said, I did find some useful information in the W3Schools React tutorials. They cover a lot of topics — maybe too many — and honestly, some of them aren't really necessary when you're getting started. But one thing that really caught my attention was the section on ES6.

If you're new to React, I think it's a good idea to take a look at ES6. It introduces a lot of features that are used all the time in React code, Understanding these will make learning React way easier.

Definición 1.1 (React ES6)**.** *ES6 stands for* ***ECMAScript 6****.*
ECMAScript was created to standardize JavaScript, and ES6 is the 6^{th} *version of it. It was officially published in 2015, and it's also known as* ***ECMAScript 2015****.*

Why should learn ES6? React uses ES6, and you should be familiar with some of the new feautures like:

- Classes
- Arrow Functions
- Variables
- Array methods
- Desctructuring

- Modules
- Temary operator
- Spread Operator

1.4.1. ES6 Classes

Definición 1.2 (Class in react)**.** *A class is a type of function, but instead of using teh keyword function to initialize it, we used the keyword class, and the properties are assigned inside a constructor method.*

Ejemplo 1.1. *A simple class constructor*

```
class Car {
  constructor(name) {
    this.brand = name;
  }
}
```

notice the case of the class name, We have begun the name Car, with an uppercase character. This is a standar naming convention for classes. **Now we can create objects using teh car class:**

Ejemplo 1.2. *Create an object called "mycar"bases on the car class*

```
class Car {
  constructor(name) {
    this.brand = name;
  }
}
```

the constructor function is called automatically when the object is initialized.

1.4.2. Arrow functions

Arrow functions allow us to write shorter function sintax

Ejemplo 1.3. *before:*

```
hello = function () {
  return "hello world!";
}
```

Ejemplo 1.4. *with arrow function:*

```
hello = () => {
  return "hello World!";
}
```

It gets shorter! if the function has only one statement, and the statement return a value, you can remove the brackets and the return keyworkd

Note: this works only if the function has only one statement. If you have parameters, pass them inside the parentheses.

Ejemplo 1.5. *Arrow function with parameters*

```
hello = (val) => "hello" + val;
```

What about *this***?** The handling of *this* is also different in arrow functions compared to regular functions. In short, with arrow functions there is no binding of *This*. In regular functions the *this* keyword represented the object that called the function, which could be the window, the document, a button or whatever. With arrow functions, the *this* keyword always represents the object that defined the arrow function.

1.4.3. Variables

Now with ES6, there are three ways of definiing your variables: *var*, *let*, and *const*.

Ejemplo 1.6. *var*

```
var x = 5.6;
```

if you use *var* outside a function, it belongs to the global scope. if youe use *var* inside of a function, it belong to that function. if you use *var* inside of a block, i.e. a for lool, the variable is still available outside of that block. *var* has a fuction scope, not a block scope.

Ejemplo 1.7. *let*

```
let x = 5.6;
```

let is the block scoped version of *var*, and is limited block (or expression) where it is defined. if you use *let* inside a block, i.e. a for loop, the variable is only available inside that for loop. *let* has a block scope.

Ejemplo 1.8. *const*

```
const x = 5.6;
```

const is a variable that once it has been created, its value can never change. *const* has a block scope. It does not define a constants value. It defines a constants reference to a value. Because of this you can NOT:

- Reassing a constant value
- Reassing a constant Array
- Reassing a constant object

but you CAN:

- Change the elements of a constant Array
- Change the properties of a constant object

1.4.4. Array Methods

There a many JavasCript array methods. One of the most useful in react is the *.map*() array method. The *.map*() method allows you to run a function on each item in the array, returning a new array as the result. In react, *.map*() can be used to generate list.

Ejemplo 1.9. *generate a list of items from an array*

```
const myArray = ['apple', 'banana', 'orange'];
const myList = myArray.map((item) => <p>{item}</p>)
```

1.4.5. Desctructuring

To illustrate destructuring, we'll make a sandwich. Do you take everything out of the refrigerator to make your sandwich? No, you only take out the items you would like to use on your sandwich. Destructuring is exactly the same. We may have an array or object that we are working with, but we only need some of the items contained in these. Destructuring makes it easy to extract only what is needed.

Destructuring Arrays

Ejemplo 1.10. *destructuring*

```
const vehicles = ['mustang', 'f-150', 'expedition'];

const [car, truck, suv] = vehicles;
```

When destructuring arrays, the order that variables are declared is important. If we only want the car and suv we can simply leave out the truck but keep the comma:

Ejemplo 1.11. *Only what we want*

```
const vehicles = ['mustang', 'f-150', 'expedition'];

const [car,, suv] = vehicles;
```

Destructuring objects

Ejemplo 1.12. *Only what we want*

```
const vehicleOne = {
  brand: 'Ford',
  model: 'Mustang',
  type: 'car',
  year: 2021,
  color: 'red',
  registration: {
    city: 'Houston',
    state: 'Texas',
    country: 'USA'
  }
}

myVehicle(vehicleOne)

function myVehicle({ model, registration: { state } }) {
  const message = 'My ' + model + ' is registered in ' + state + '.';
}
```

1.4.6. Spread Operator

The JavaScript spread operator (...) allows us to quickly copy all or part of an existing array or object into another array or object.

Ejemplo 1.13. *spread operator*

```
const numbersOne = [1,2,3];
const numbersTwo = [4,5,6];
const numbersCombined = [...numbersOne, ...numbersTwo];
```

the spread operator is often used in combination with destructuring

Ejemplo 1.14. *Assing the first and second items from numbers to variables and put the rest in an array*

```
const numbers = [1,2,3,4,5,6];
const [one, two, ...rest] = numbers;
```

We can use the spread operator with objects too

Ejemplo 1.15. *Combine thse two objects*

```
const myVehicle = {
  brand: 'Ford',
  model: 'Mustang',
  color: 'red'
}

const updateMyVehicle = {
  type: 'car',
  year: 2021,
  color: 'yellow'
}

const myUpdatedVehicle = {...myVehicle, ...updateMyVehicle}
```

1.4.7. Modules

JavaScript modules allow you to break up your code into separate files. This makes it easier to maintain the code-base. ES Modules rely on the *import* and *export* statements.

Export You can export a function or variable from any file.

Import You can import modules into a file in two ways, based on if they are named exports or default exports. Named exports must be destructured using curly braces. Default exports do not.

Ejemplo 1.16. *import named exports from the file person.js*

```
export { name , age } from 'person.js';
```

Ejemplo 1.17. *import a default export from the file message.js*

```
import message from './message.js';
```

1.4.8. Ternary Operator

The ternary operator is simplified conditional operator like if / else. Syntax: *condition ? <expression if true>: <expression if false>*

Ejemplo 1.18. *Before:*

```
if (authenticated) {
  renderApp();
} else {
  renderLogin();
}
```

Example using ternary operator:

```
authenticated ? renderApp() : renderLogin();
minted
```

1.5. React Components

One of the most important features in React — and probably the main reason many beginner programmers start using it — is the concept of **components**.

But... why components?

Definición 1.3 (React Components)**.** *A React component is an independent and reusable piece of code. They serve the same purpose as JavaScript functions, but they work in isolation and return HTML (via JSX).*

Why Are They So Important?

In my experience, components help a lot with keeping your code clean, organized, and easier to scale. Instead of having a huge file with all your HTML and logic, you can break your UI into smaller parts — each one focused on a specific task or section of the page.

That makes your project easier to understand and maintain.

How to Create a Component

There are two main ways to create a component: function components and class components. Nowadays, function components are more common because they are simpler and work perfectly with React Hooks.

Here's a basic example of a functional component:

```
function Greeting() {
  return <h1>Hello from a component!</h1>;
}
```

To use it, just write:

```
<Greeting />
```

Components Inside Components

You can place one component inside another — that's actually a common and powerful pattern in React.

```
function App() {
  return (
    <div>
      <Header />
      <MainContent />
      <Footer />
    </div>
  );
}
```

Each of those components (‘Header‘, ‘MainContent‘, ‘Footer‘) would be defined separately — either in the same file or, ideally, in their own files.

Components in Files

As your app grows, it's a good idea to create a folder called `components/` and organize each component into its own file.

For example:

```
src/
  App.tsx
  components/
    Header.tsx
    MainContent.tsx
    Footer.tsx
```

Then, in your `App.tsx`, you can import them like this:

```
import Header from './components/Header';
import MainContent from './components/MainContent';
import Footer from './components/Footer';
```

This way, your code stays clean and modular — which is one of the biggest advantages of using React.

1.6. React JSX

Apart from the previous section, there's something else I found interesting and a bit helpful when trying to understand the basics of React — and that's JSX.

JSX might look confusing at first, especially if you're coming from regular HTML and JavaScript, but once you get the idea behind it, things start making a lot more sense. So let's take a quick look at what JSX is and why it's important.

Definición 1.4 (JSX)**.** *JSX stand for JavaScript XML, this allow us to write HTML in React. and makes it easier to write and add HTML in react.*

Coding JSX

JSX allows us to write HTML elements in JavaScript and place them in the DOM wihout any *createElement*() and.or *appendChild*() methods. *You are not required to use JSX, but JSX makes it easier to write React Applications.*

Expressions in JSX

With JSX you can write expressions inside curly braces . The expression can be a React variable, or property, or any other valid JavaScript expression. JSX will execute the expression and return the result:

Ejemplo 1.19. *Execute the expresion* $5 + 5$*:*

```
const myElement = <h1>React is {5 + 5} times better with JSX</h1>;
```

Inserting a Large block of HTML

To write HTML on multiple lines, put the HTML inside parentheses:

Ejemplo 1.20. *Create a list with three list items:*

```
const myElement = (
  <ul>
    <li>Apples</li>
    <li>Bananas</li>
    <li>Cherries</li>
  </ul>
);
```

One Top-Level Element

In React, the HTML must be wrapped in **one** top-level element. So, if you want to write two paragraphs, you need to place them inside a parent element, like a `<div>`.

Alternatively, you can use a **fragment** to wrap multiple elements. This helps avoid unnecessarily adding extra nodes to the DOM. A fragment looks like an empty HTML tag: `<></>`.

Ejemplo 1.21 (One Top-Level Element)**.** *Wrap two paragraphs inside a* `div` *element:*

```
const myElement = (
  <div>
    <p>I am a paragraph.</p>
    <p>I am a paragraph too.</p>
  </div>
);
```

Wrap two paragraphs inside a fragment:

```
const myElement = (
  <>
    <p>I am a paragraph.</p>
    <p>I am a paragraph too.</p>
  </>
);
```

Attribute class = classname

The *class* attribute is a much used attribute in HTML, but since JSX is rendered as JavaScript, and the *class* keyword is a reserved word in JavaScript, you are not allowed to use it in JSX. *Use attribute className instead.* JSX solved this by using *className* instead. When JSX is rendered, it translates *className* attributes into *class* attributes.

Ejemplo 1.22. *Using the attribute ClassName instead of class in JSX*

```
const myElement = <h1 className="myclass">Hello World</h1>;
```

Conditions - if statements

React supports if statements, but not inside JSX.

To be able to use conditional statements in JSX, you should put the if statements outside of the JSX, or you could use a ternary expression instead, wich one we alrady talk in ES6 section.

1.7. React Props

Another important topic in React is the use of **props**.

Props are arguments passed into React components. They are passed via HTML-like attributes and allow you to send data from a parent component to a child component.

The word **props** stands for "properties"— basically, values that you can use inside a component to make it more dynamic and reusable.

Using Props in React

Here's a simple example to help you understand how props work:

Ejemplo 1.23 (React Props). *A component that receives props:*

```
function Welcome(props) {
  return <h1>Hello, {props.name}!</h1>;
}
```

Using the component and passing a prop:

```
<Welcome name="Jakxel" />
```

In this case, the `Welcome` component receives a prop called `name`, and it uses that value to display a custom message. You can pass as many props as you want, and even pass objects, arrays, or functions. Props are read-only, which means a component cannot change the props it receives — this helps keep your code predictable and easier to debug. Props are one of the most essential tools in React, especially when you're working with multiple components that need to share data.

1.8. React Events

In React, just like in the HTML DOM, you can perform actions based on user events. Events are one of the main ways to create interaction and dynamic functionality in your project.

React supports the same types of events as plain HTML — like `onClick`, `onChange`, `onMouseOver`, and many others.

Adding Events

You can add an event to an element by using a camelCase syntax and passing a function:

```
function MyButton() {
  function handleClick() {
    alert('Button clicked!');
  }

  return <button onClick={handleClick}>Click me</button>;
}
```

Notice how `onClick` is written in camelCase (not lowercase like in HTML), and you pass the function reference without calling it (no parentheses).

Passing Arguments to Event Handlers

If you want to pass arguments to the function when the event happens, you can use an arrow function:

```
<button onClick={() => handleClick('Jakxel')}>Greet</button>

function handleClick(name) {
  alert(`Hello, \${name}!`);
}
```

React Event Objects

React wraps the native browser event in its own synthetic event system for better performance and cross-browser compatibility. You can access the event object like this:

```
function handleInput(event) {
  console.log(event.target.value);
}

<input type="text" onChange={handleInput} />
```

This `event` object has all the properties you'd expect from a regular DOM event — like `target`, `type`, and more.

—

Using events in React is a fundamental part of building interactive interfaces. Whether it's clicking buttons, typing in inputs, or hovering over elements, React gives you powerful and simple tools to handle it all.

1.9. Conditionals in React

As we've seen before, you can use conditional statements in JSX to control what gets rendered. React supports different ways to handle conditions — like `if` statements, the ternary operator, and even the powerful `&&` operator.

Each of these tools helps you decide what should be shown on the screen based on certain conditions (like a boolean state, user input, or data).

Using `if` Statements

You can use a regular `if` statement outside of JSX (before the return) to decide what to render:

```
function Greeting(props) {
  if (props.isLoggedIn) {
    return <h1>Welcome back!</h1>;
  }
  return <h1>Please log in.</h1>;
}
```

Using the Ternary Operator

This is useful for writing inline conditionals inside JSX:

```
<h1>{isLoggedIn ? 'Welcome back!' : 'Please log in.'}</h1>
```

Using the `&&` Operator

The `&&` operator is great for rendering something only if a condition is true. It's like saying: Ïf this is true, then do that."

```
{isLoggedIn && <p>You are logged in.</p>}
```

This works because if the condition before `&&` is false, React skips rendering the element on the right.

Summary

- Use `if` for more complex logic outside JSX. - Use the `ternary operator` for inline "if-else" inside JSX. - Use `&&` when you only want to show something if a condition is true.

Understanding these will help you make your components more dynamic and responsive to different states.

1.10. Routing in React

Routing is one of the fundamental topics when building a web project. It allows you to navigate between different pages or views in your application — which is super useful when you're building things like a navbar, a dashboard, or any multi-page structure.

In React, we usually handle routing using a library called `react-router-dom`. This tool helps us create and manage routes easily, so we can display different components depending on the URL.

Why is Routing Important?

Without routing, your entire app would just be one long page. Routing helps you split your app into separate "pages"(even though technically they're still part of a single-page app). This keeps everything organized and makes it feel like a real website.

How to Set Up Routing

First, you need to install the routing package:

```
npm install react-router-dom
```

Then, in your main file (usually `main.tsx` or `main.jsx`), wrap your app with the `BrowserRouter`:

```
import { BrowserRouter } from 'react-router-dom';
import App from './App';

ReactDOM.createRoot(document.getElementById('root')).render(
  <BrowserRouter>
    <App />
  </BrowserRouter>
);
```

Creating Routes

Inside your `App.tsx` (or `App.jsx`), define your routes like this:

```
import { Routes, Route } from 'react-router-dom';
import Home from './pages/Home';
import About from './pages/About';

function App() {
  return (
    <Routes>
      <Route path="/" element={<Home />} />
      <Route path="/about" element={<About />} />
    </Routes>
  );
}
```

Now, when the user visits `/`, they'll see the `Home` component, and when they go to `/about`, they'll see the `About` page.

Using a Navbar

With routing set up, you can create a simple navbar like this:

```
import { Link } from 'react-router-dom';

function Navbar() {
  return (
    <nav>
      <Link to="/">Home</Link>
      <Link to="/about">About</Link>
    </nav>
  );
}
```

This lets users click links and navigate between pages without reloading the page — a key feature of single-page applications (SPAs).

1.11. React Hooks

What is a Hook?

Hooks allow function components to have acces to state and other react features, Because of this, class componentsare generally no longer needed Althought Hooks generally replace class components, there are no plants to remove classes from React. Hooks allows us to "*Hook*" into react features such as state and lifecycle methods we must *import* Hooks fro *react*.

Hook rules

There are 3 rules for hooks:

- Hooks can only be called inside react function components
- Hoos can only be called at the top level of a component
- Hooks cannot be a conditional

Note: Hooks will not work in React class components.

2. useState

2.1. State fundamentals

Definición 2.1 (UseState Hook)**.** *The useState hook is a fnctio9n that allows you to add state to a functional component.*

UseState is an alternative to the usereducer hook that is preferred when we require the basic uptade. Is used to add the state variables in the components. FOr using the useState hook we have to inported it into he component.

```
const [state, setState] =
useState(initialState)
```

- **state:** It is he value of the current state
- **setState:**
- **initialState:** It is the initial value of the state.

2.2. How does useState() works?

1. **initialize State:** When you call usestate(initialvalue), it creates a state variable and an updater function.

```
const [count, setCount ] = useState(0);
```

2. **State is preserved across renders:** react remembers the state value between re-renders of the component. Each time the component renders, react keeps the latest value of count.
3. **State uptades with the updater function:** When you call setCount(newValue) react updates the state and it re-renders the component to reflect the new state value.

```
<button onCLick={() => setCount(count+1)}>Increment</button>
```

4. **Triggers Re-render:** react will re-render **only the component** where useState was used-ensuring you UI updates automatically when the state changes.

2.3. When to use useState

Whe should use the state management solution.

- We need a simple state management solution.
- We component has state that changes over time.
- The state does not require complex updates or dependencies.

3. useEffect

The useEffect is one of the most commonly used hooks in react used to handle side effects in functional components.

3.1. What is useEffect?

Definición 3.1 (useEffect)**.** *The useEffect is used to handle the side effects such as fetching data and updating DOM.*

This hooks runs on every render but there is aslo a way of using a dependecy array using which we can control the effect of rendering. It can be used ti perform actions such as:

- Fetching data from an API.
- Setting up event listeners of subscriptions.
- Manipulating the DOM directly(Althought react generally handles DOM manipulation for you)
- Cleaning up resources when a component unmounts.

Syntax

```
useEffect(() => {
  // code to run on each render
  return () => {
    // cleanup function (optional)
  };
}, [dependencies]);
```

- **Effect function:** this is wehre your side effect code runs.
- **Cleanup functions:** This optional return function cleans up side effects like subscriptions or timers when the component unmounts.
- **dependeces array:** React re-runs the effect if any of the values in this array change.

3.2. how does it works?

- **Initial render happens:** react renders the component and updates the DOM.
- **useEffect after render:** It runs after the paint, not during render.
- **Dependecies are checked:** If there is no dependecy array, the effect runs after every render; if the array is empty([]), it runs once on mount; if dependencies are provided, it run sonly when those values change.
- **Cleanup function runs:** Before teh effect re-runs or the components unmountsm the clenaup function (returned from useEffect) is executed.
- **Effect re-runs:** If dependencies changed, the effect runs again - after cleanup.

3.3. Best practices for useEffect

Always provide a dependecy array: This helps React know when the effect shoul run. If you dont specify dependencies, it will run on every render.

- **Use multiple useEffect hooks for different concerns:** Split logic into separate effects to keep your code organized and readable.
- **Cleanup effects:** If your effect involves timers, subscriptions, or external resources, return a cleanup function to prevent memory leaks.
- **Avoid heavy logic inside useEffect:** Keep the logic inside useEffect simple. if needed, move complex logic into separate functions.

Implement useEffect hook

```
import { useState, useEffect } from "react";

function HookCounterOne() {
    const [count, setCount] = useState(0);

    useEffect(() => {
        document.title = `You clicked \${count} times`;
    }, [count]);

    return (
```

```
        <div>
            <button onClick={() => setCount((prevCount) => prevCount + 1)}>
                Click {count} times{" "}
            </button>
        </div>
    );
}
export default HookCounterOne;
```

In this example

- useEffect triggers a function on every component render, using react to execute specified task efficiently.
- Positioned within the component, it grants easy access to state and props without additional coding.
- for replicating lifecycle methods in functional components, copy and customize the provided code snippet according to your needs.

3.4. useEffect conclusion

The useEffect hook is a powerful tool in react for handling side effects in function components. By using useEffect, you can easily manage tasks like data fetching, subscribing to events, and cleaning up resources when a component unmounts. Its flexibility allows you to run effects after the component render, only when certain dependencies change, or once when the component mounts.

4. useContext

In react applications, sometimes managing state across deeply nested components can become very difficult. The useContext hook offers a simple and efficient solution to share state between components without the need for prop drilling.

4.1. What is useContext hook

The useContext hook in React allows components to consume values from the React context. React's context API is primarily designed to pass data down the component tree without manually passing props at every level. useContext is a part of React's hooks system. It help avoid the problem of "prop drilling"where props are passed down multiple levels for parent to child components.

- Simplies accessing shared state aracross components.
- Avoids prop drilling by eliminating the need to pass props down multiple levels.
- Works seamlessly with react context API to provide global state.
- Ideal for managing themes, authentication, or user preferences across the app.

Syntax

```
const contextValue = userContext(myContext);
```

- **MyContext:** The context object is created using react.createContext().
- **contextValue:** The current context value that we can use in our component.