REACT HANDBOOK

FLAVIO COPES

Preface

This book aims to introduce you to React, the most popular JavaScript/TypeScript UI library.

It’s a gentle introduction to the basic concepts of React, and assumes no prior React knowledge.

You’ll learn the basic concepts in a very straightforward way, without too much jargon or complicated explanations, using an 80/20 approach that give you the mental models and knowledge to start using React for your projects.

Examples are written using TypeScript. If you need, check out my JavaScript Handbook, and the TypeScript Handbook.

The book is up to date with React 19 and the latest best practices for client-side React.

For use within Next.js, check out my Next.js Handbook.

This book was published in late 2024.

Legal

Introduction to React

The goal of this handbook is to provide a starter guide to learning React.

At the end of the book, you'll have a basic understanding of:

- What is React and why it's so popular
- How to install React using Vite
- Components, JSX, State, Props
- Handling user events in React
- Effects

React is a JavaScript library that aims to simplify the development of highly interactive 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, by dividing the UI into a collection of components.

You might find some initial difficulties learning React, depending on your existing experience. It’s normal. Once it "clicks", I guarantee it's going to be one of the best experiences you will have, because React makes many things easier than ever, and its ecosystem is filled with great libraries and tools.

How much JavaScript do you need to know to use React?

Before jumping straight into React, you should have a good understanding of some core JavaScript concepts.

You don't have to be an expert, but I think you need a good overview of:

- Variables
- Functions, Arrow functions
- Objects and arrays
- Work with objects and arrays using rest and spread
- Object and array destructuring
- Template literals
- Callbacks
- ES Modules

If those concepts sound unfamiliar, I recommend reading my JavaScript Beginner’s Handbook first. I also provided you with some links to find out more about those subjects.

Why should you learn React?

I highly recommend any Web developer to have at least a basic understanding of React, because of a few reasons.

1. React is very popular. A lot of companies use it. A lot of jobs require knowing React. 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. The ecosystem of React is huge. Many, many libraries and interesting projects are built around it, for example libraries like React Router, TanStack Query, shadcn/ui
3. A lot of tooling today is built using React at the core. Popular frameworks and tools like Next.js, Remix, TanStack Start and many others use React under the hood.

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

It allows you to build UIs in ways that you can't do with plain DOM APIs.

It promotes several good development practices, including code reusability and components-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.

Of course, similar libraries exist, like Vue or Angular or Svelte, but in this book we'll focus on React only.

How to install React

There are many ways to create a project with React.

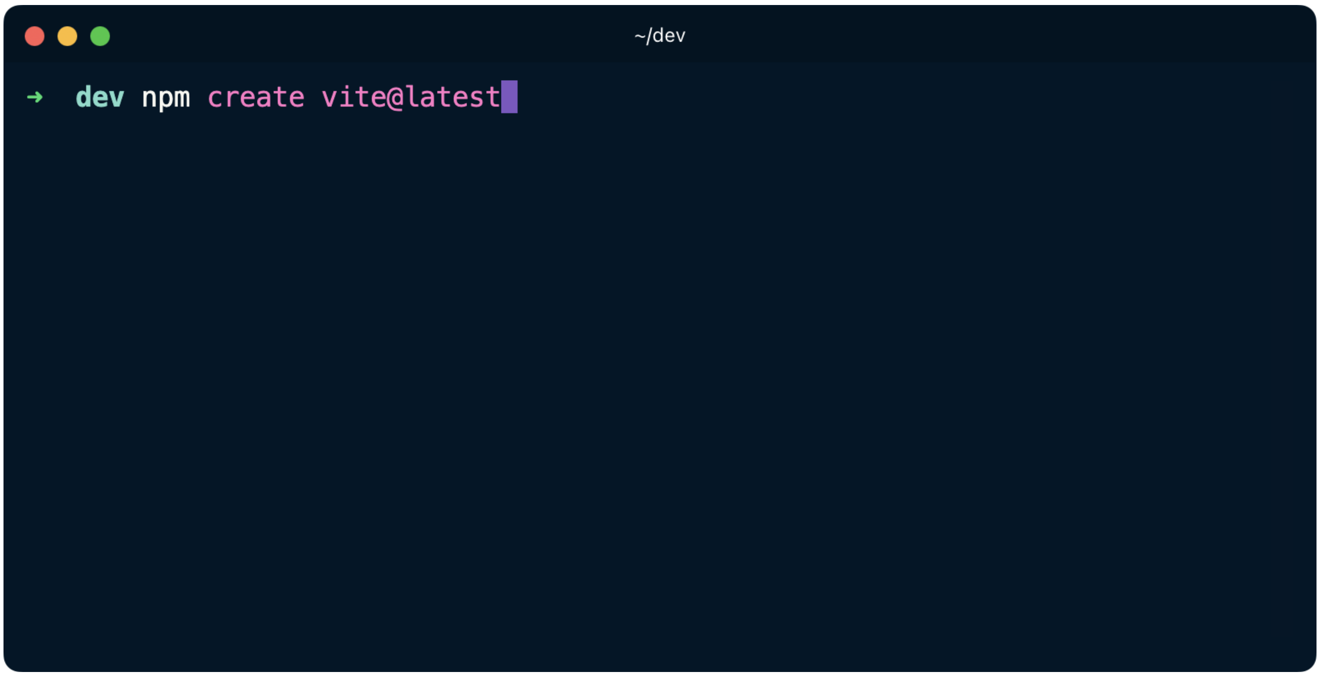
The way I suggest is by using Vite.

Vite is a modern tool that provides a development server, is very fast, and many people in the JS community consider it optimal.

💁‍♂️ Note: Vite can be used as a replacement for `create-react-app`, another tool that's popular but also slower. You can use that instead if you prefer, but I found Vite just great and it can also be used with other libraries, not just React.

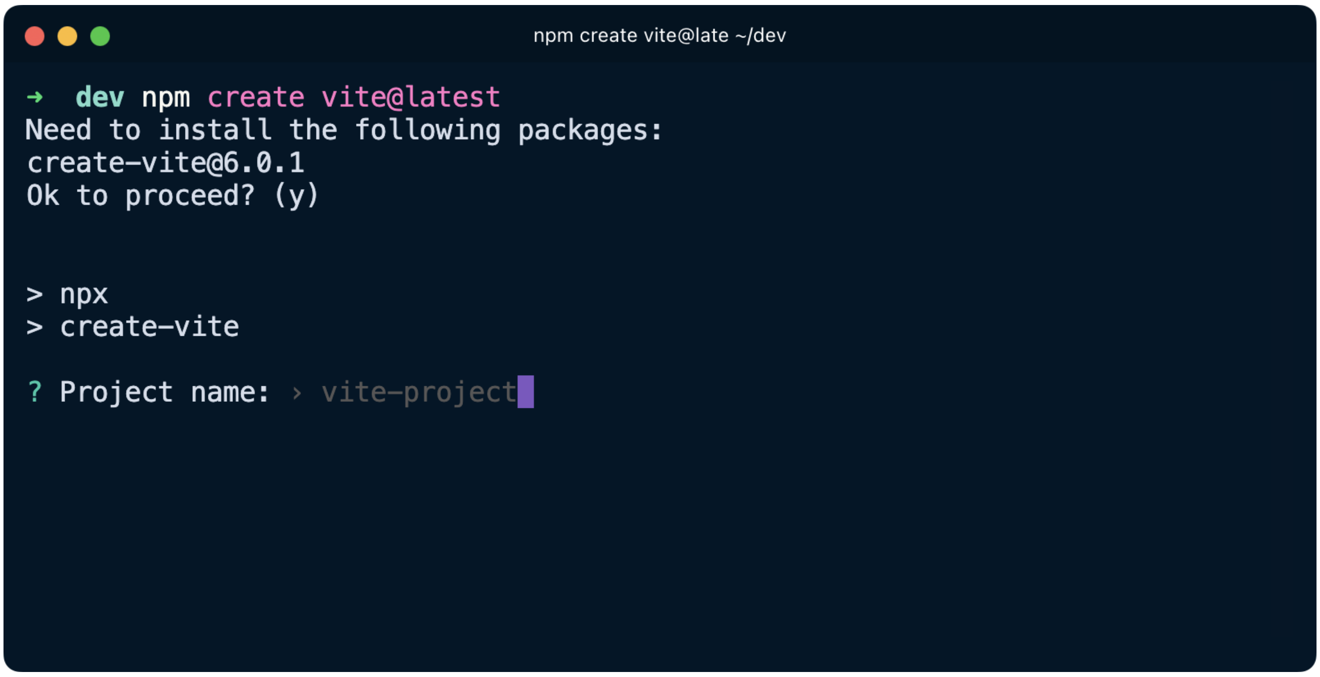
To create a project using Vite you first go into the folder where you host all your projects, in my case a `dev` folder in my user's home folder.

Then run `npm create vite@latest` (you don't have to install anything, it's all automatic thanks to `npm create`):



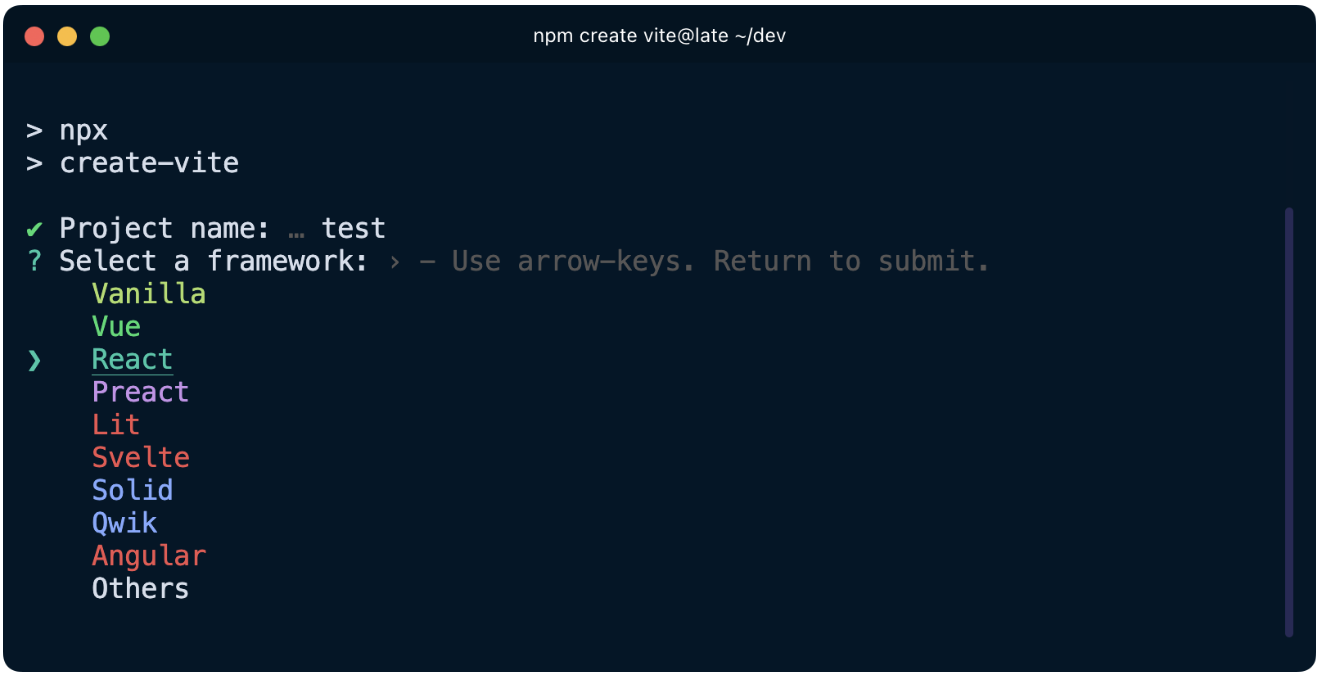
```
➜  dev npm create vite@latest
```

Choose a name for the project. That will also be the project's folder name. In this case “test”:



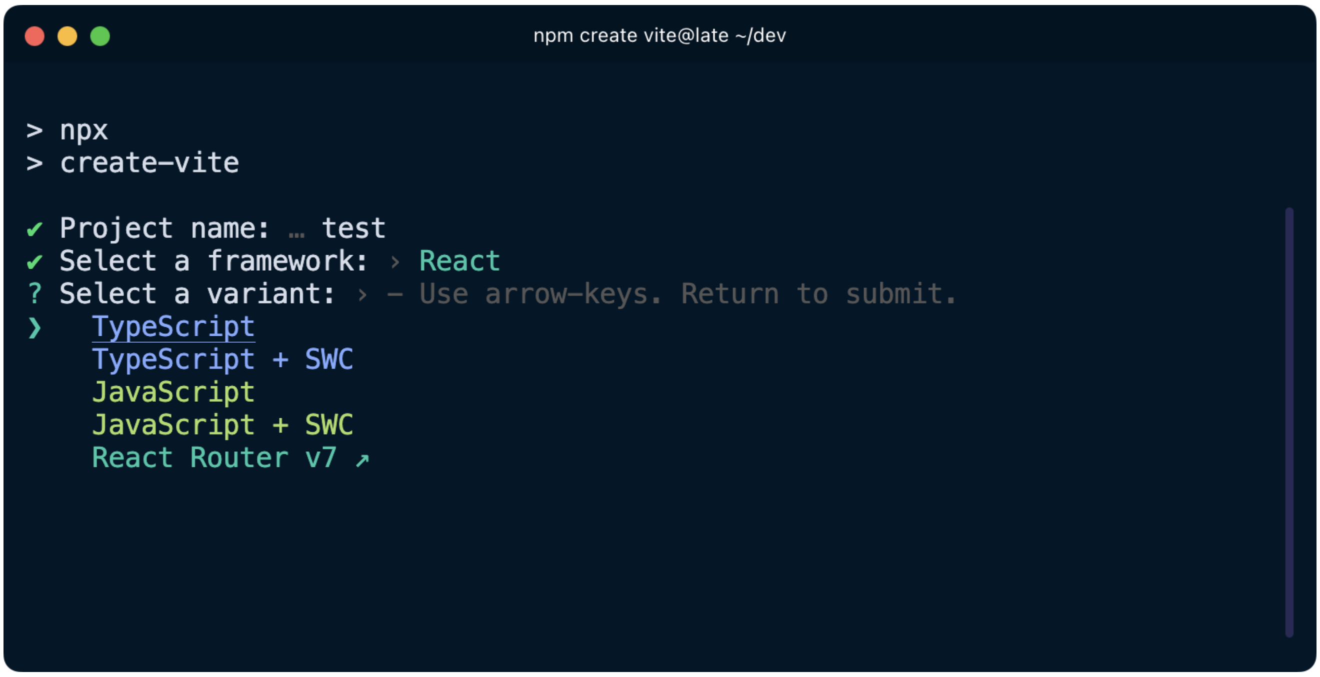
```
➜  dev npm create vite@latest
Need to install the following packages:
create-vite@6.0.1
Ok to proceed? (y)


> npx
> create-vite

? Project name: › vite-project
```

Now you can choose a framework. Pick “React”.



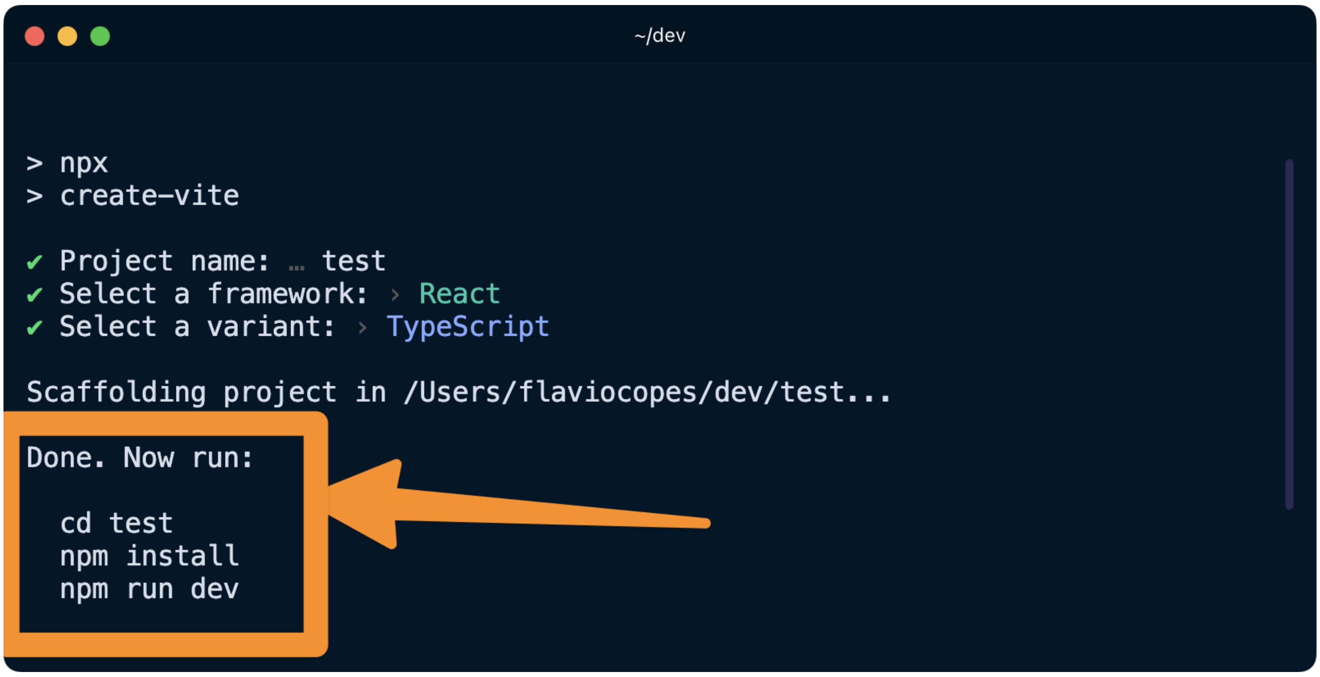
```
> npx
> create-vite

✔ Project name: … test
? Select a framework: › - Use arrow-keys. Return to submit.
    Vanilla
    Vue
❯   React
    Preact
    Lit
    Svelte
    Solid
    Qwik
    Angular
    Others
```

Pick **TypeScript** from the list (note that we won't use any TypeScript-specific feature, but it's always good to enable it due to the nice help it can provide to the editor and the code authoring experience):



```
> npx
> create-vite

✔ Project name: … test
✔ Select a framework: › React
? Select a variant: › - Use arrow-keys. Return to submit.
❯   TypeScript
    TypeScript + SWC
    JavaScript
    JavaScript + SWC
    React Router v7 ↗
```

Done!



```
> npx
> create-vite

✔ Project name: … test
✔ Select a framework: › React
✔ Select a variant: › TypeScript

Scaffolding project in /Users/flaviocopes/dev/test...

Done. Now run:

  cd test
  npm install
  npm run dev
```

Now go to the newly created project folder:

```
cd test
```

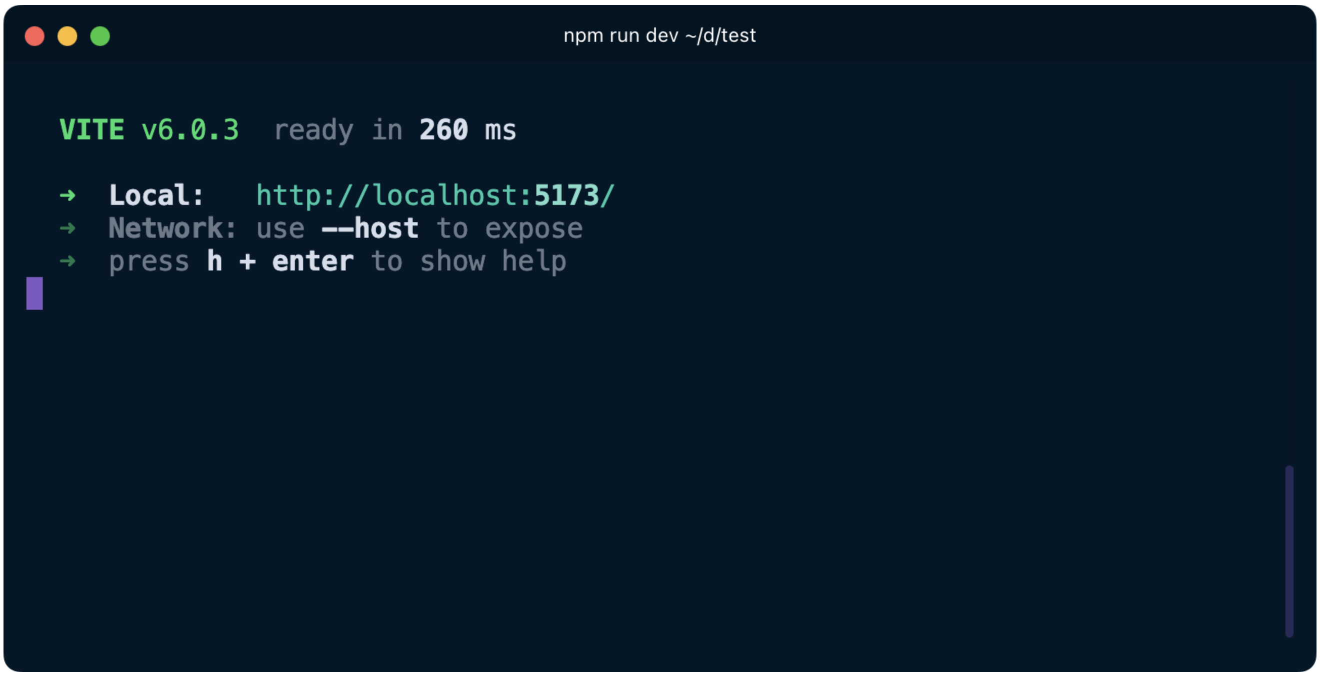
and run

```
npm install
```

to install the dependencies, followed by

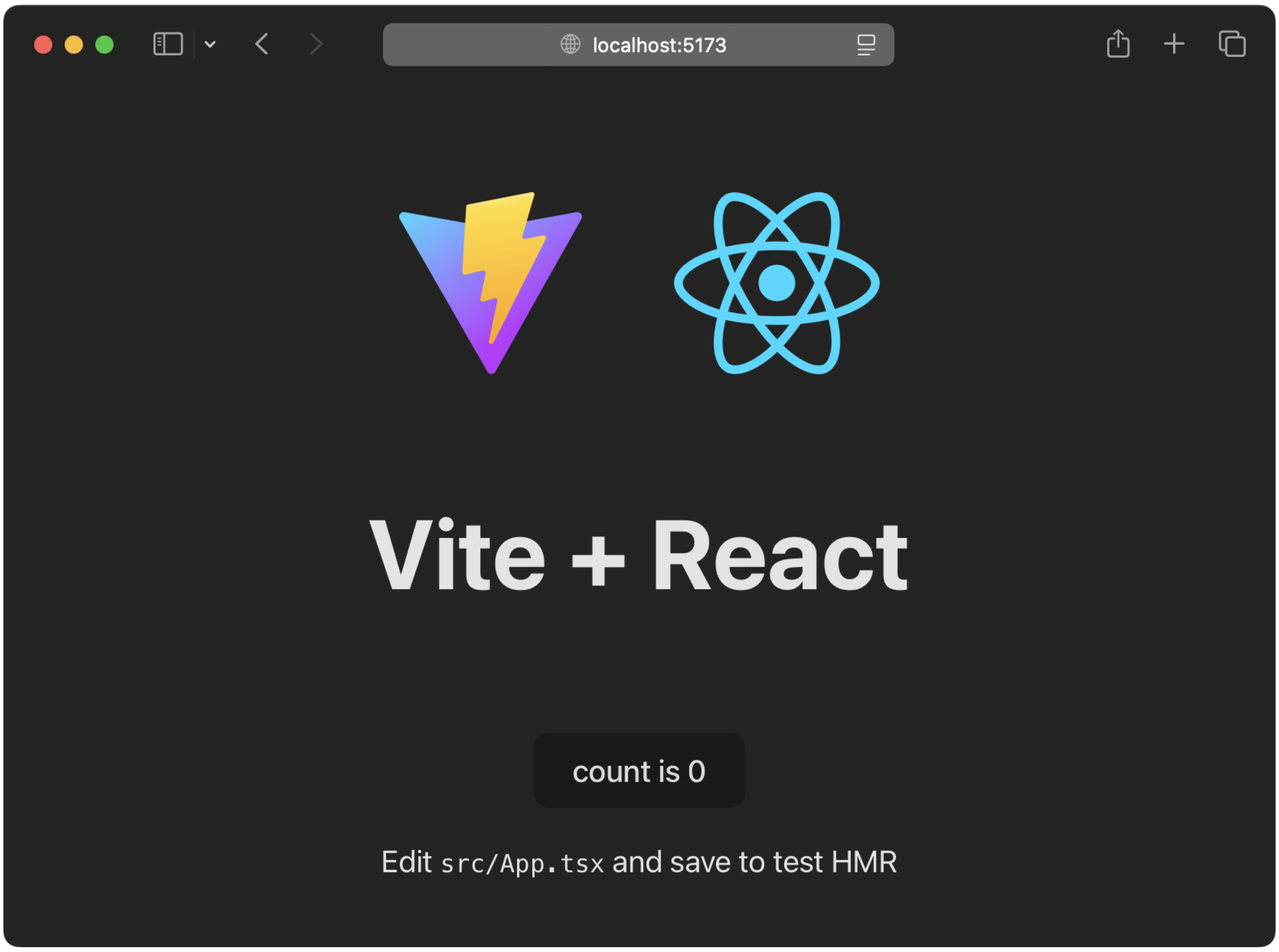
```
npm run dev
```

to start the application:



```
  VITE v6.0.3  ready in 260 ms

  ➜  Local:   http://localhost:5173/
  ➜  Network: use --host to expose
  ➜  press h + enter to show help
```

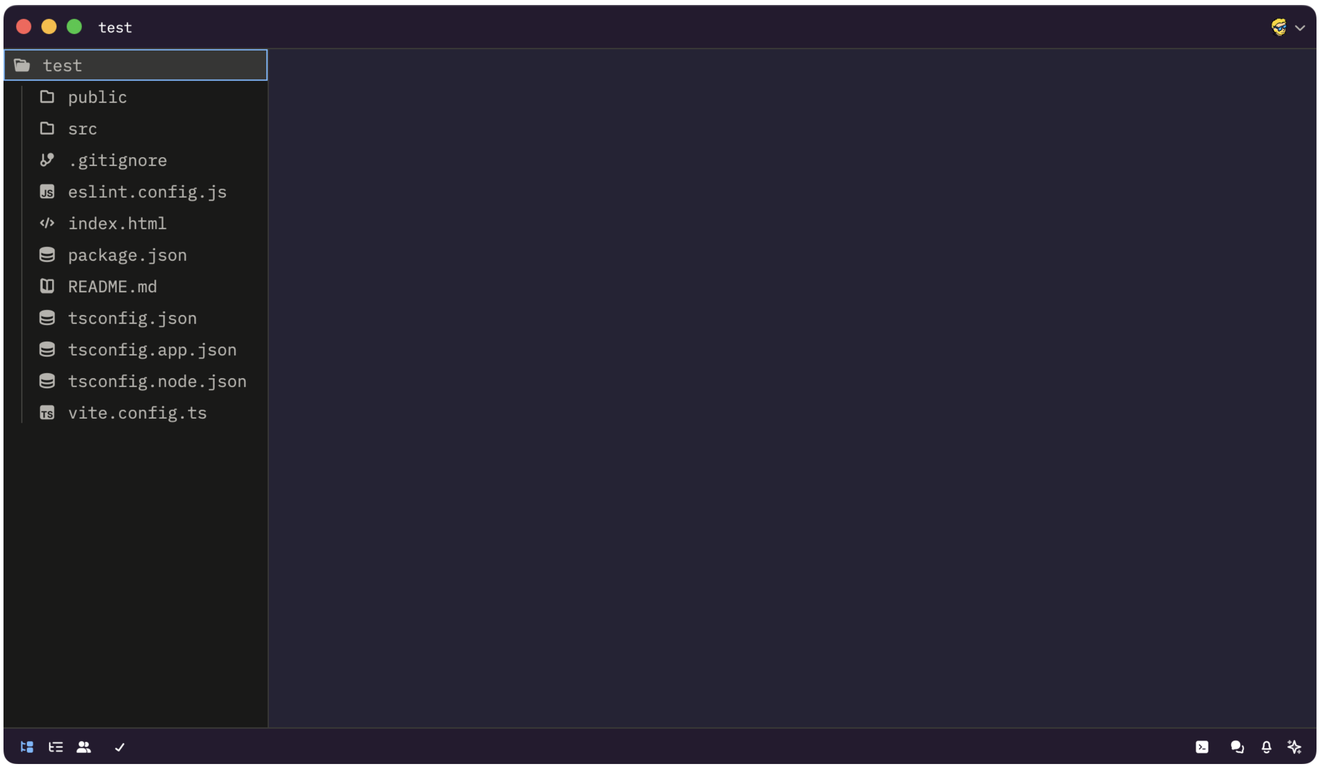
The application should be running at http://localhost:5173 (the port might be different if it's already used)



Now you're ready to work on this application!

Here's the application folder opened in a code editor.

As you can see, Vite created a basic application with some placeholder files:



React Components

You just saw how to create your first React application, using Vite.

One of the main parts of a React applications are **components**.

Let's start by talking about what is a component.

Let’s create an `App` component, a super simple one:

```
function App() {
  return /* something */
}
```

You can see we define a function called `App`.

`App` is a function.

Inside the function we can do *something*, typically we fetch data, or we inspect the current state of the application.

Then, we can return something from a function, and that will be *rendered* by React on the page.

Like this:

```
function App() {
  return <p>test</p>
}
```

Basically when the component is rendered, in this case in the browser we'll see the HTML code `<p>test</p>`.

This looks like **HTML,** but it is returned from a JavaScript function.

Quite strange, if you're not used to 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.

First, it must have a name that starts with a capital letter, like `App` in our case.

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 talk about **props**.

Don't worry, we're going to look in detail at all those terms (JSX, State, and Props) right now.

JSX

We can't talk about React without first explaining JSX (technically, TSX as we use TypeScript).

Let's look at a real component. You have it in the app you created, in `src/App.tsx`:

```
import { useState } from 'react'
import reactLogo from './assets/react.svg'
import viteLogo from '/vite.svg'
import './App.css'

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

  return (
    <>
      <div>
        <a href="https://vite.dev" target="_blank">
          <img src={viteLogo} className="logo" alt="Vite logo" />
        </a>
        <a href="https://react.dev" target="_blank">
          <img src={reactLogo} className="logo react" alt="React logo"
/>
        </a>
      </div>
      <h1>Vite + React</h1>
      <div className="card">
        <button onClick={() => setCount((count) => count + 1)}>
          count is {count}
        </button>
        <p>
          Edit <code>src/App.tsx</code> and save to test HMR
        </p>
      </div>
      <p className="read-the-docs">
        Click on the Vite and React logos to learn more
      </p>
    </>
  )
}

export default App
```

Feel free to ignore the imports, and that weird `useState()` function call. Let's look at what is returned from the component.

This is all JSX:

```
 <>
  <div>
    <a href="https://vite.dev" target="_blank">
      <img src={viteLogo} className="logo" alt="Vite logo" />
    </a>
    <a href="https://react.dev" target="_blank">
      <img src={reactLogo} className="logo react" alt="React logo" />
    </a>
  </div>
  <h1>Vite + React</h1>
  <div className="card">
    <button onClick={() => setCount((count) => count + 1)}>
      count is {count}
    </button>
    <p>
      Edit <code>src/App.tsx</code> and save to test HMR
    </p>
  </div>
  <p className="read-the-docs">
    Click on the Vite and React logos to learn more
  </p>
</>
```

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

We have `className` instead of `class` (plot twist, it’s because `class` is a reserved word in JavaScript and can’t be used in code).

And we have functions embedded in the code with curly braces `{}`.

It looks like a weird mix of HTML and JavaScript.

We have an empty tag `<></>` wrapping everything, too. That is basically an empty HTML tag, if you want, and it’s needed because we can only return a single tag from a component.

Under the hood, React will process the JSX and 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: JSX is a fantastic way to compose interfaces.

Once you'll get more familiar with it, of course.

In the next section, we'll talk about how JSX lets you easily compose a UI, then we'll look at the differences with "normal HTML" that you need to know.

Using JSX to compose UI

As introduced in the last section, one of the main benefits of JSX is to make it very easy to build a UI.

In particular, in a React component, you can import other React components, 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.tsx` file.

We're going to create a `WelcomeMessage` component:

```
function WelcomeMessage() {
  return <h1>Welcome!</h1>
}
```

> We define a **component** as *a function that returns some JSX,* with a capitalized name

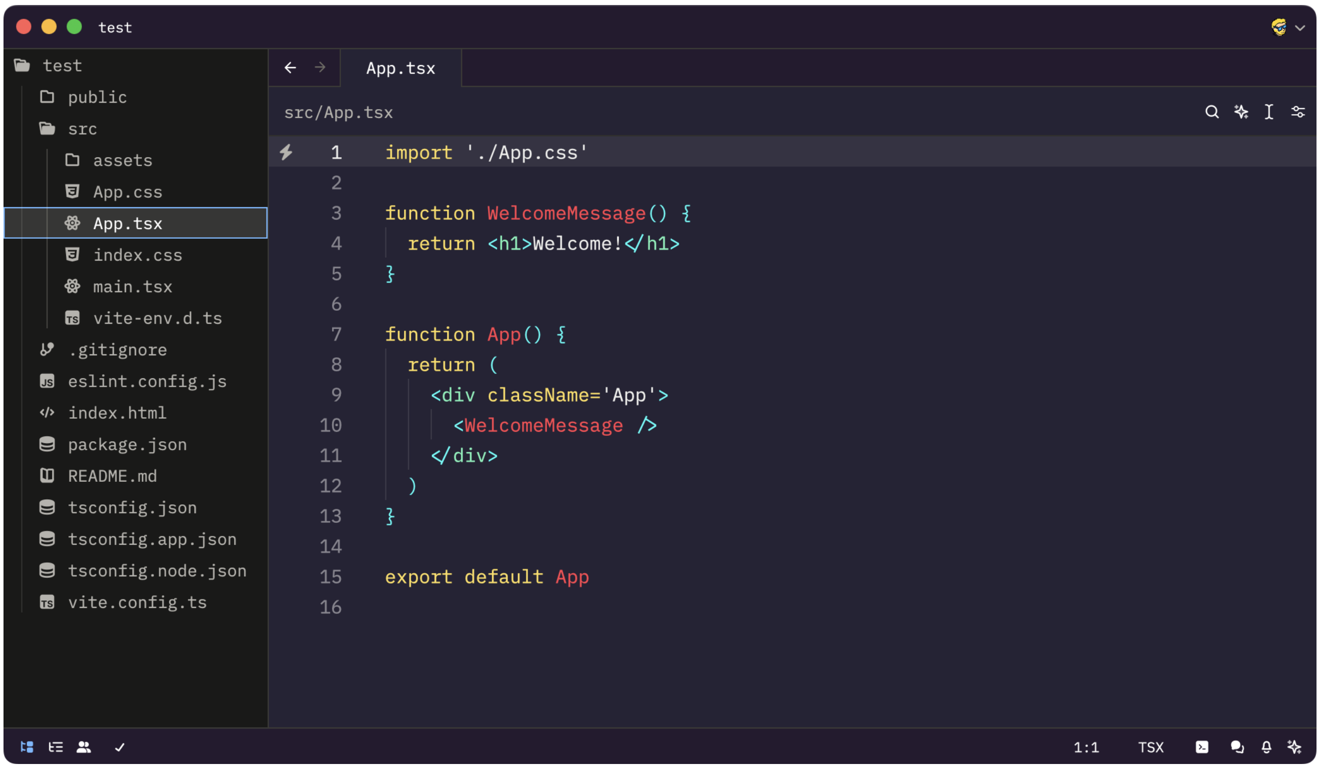
See? `WelcomeMessage` is a simple function that returns a line of JSX that represents an `h1` HTML element.

We're going to add it to the `App.tsx` file.

> NOTE: as your app grows you typically put each component in its own file, and import it. But for simple cases, like this, we can define multiple components in a single file.

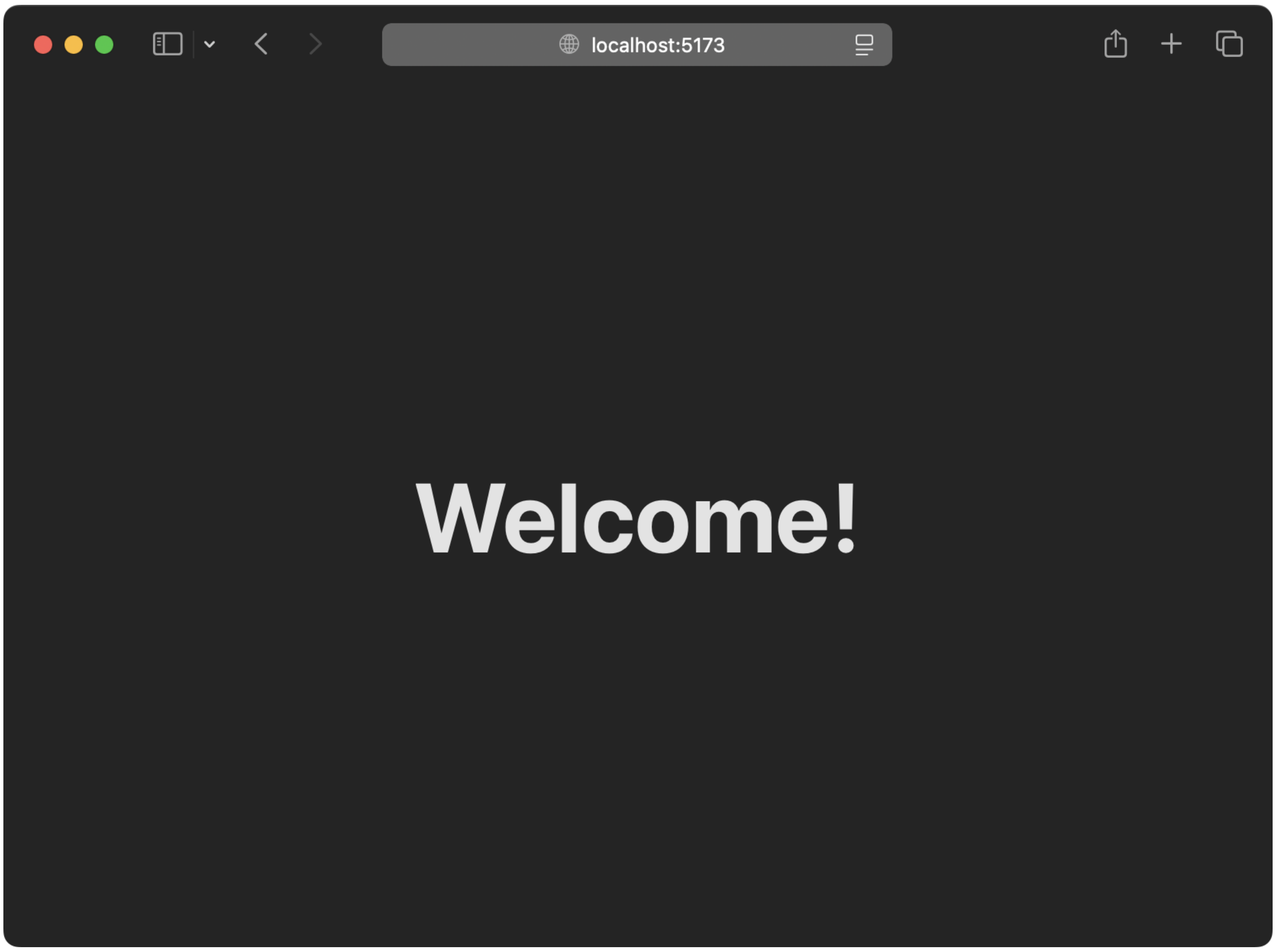
We can delete all we have in that file, and we can write this code that adds `<WelcomeMessage />` to the `App` component to show that component in the user interface:

```
import './App.css'

function WelcomeMessage() {
  return <h1>Welcome!</h1>
}

function App() {
  return (
    <div className='App'>
      <WelcomeMessage />
    </div>
  )
}

export default App
```



```tsx
import './App.css'

function WelcomeMessage() {
  return <h1>Welcome!</h1>
}

function App() {
  return (
    <div className='App'>
      <WelcomeMessage />
    </div>
  )
}

export default App
```

And here's the result. Can you see the "Welcome!" message on the screen?



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

We're including the `<WelcomeMessage />` component as a tag, as if it was part of the HTML language, like `<p>` or `<input>`.

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

With some differences, as we'll see in the next lesson.

The difference between JSX and HTML

JSX kind of looks like HTML, but it's not.

In this section, I want to introduce you to some of the most important things you need to keep in mind when using JSX.

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 at 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 copying/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. But JSX is not forgiving. If you forget to close a tag in JSX, 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.

Another big difference between JSX and HTML is that in JSX we can *embed JavaScript*.

Let's talk about this in the next section.

Embedding JavaScript in JSX

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

React lets us use JavaScript in the JSX, by wrapping it into curly brackets.

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

We import the `react.svg` SVG file using

```
import reactLogo from './assets/react.svg'
```

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

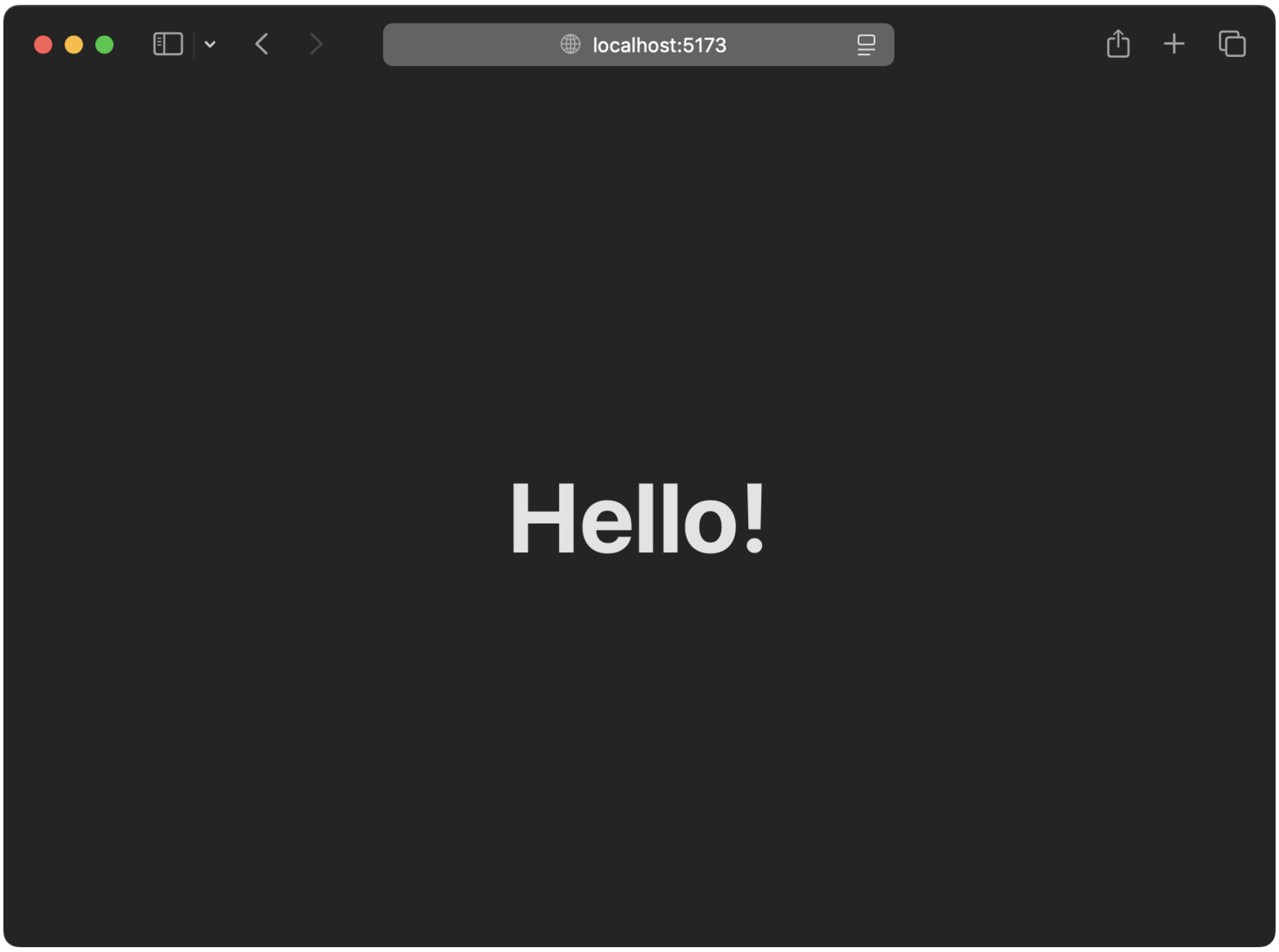
```
<img src={reactLogo} className='logo react' alt='React logo' />
```

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.

```
import './App.css'

function App() {
  const message = 'Hello!'

  return (
    <div className='App'>
      <h1>{message}</h1>
    </div>
  )
}

export default App
```

Try it! You should see the `Hello!` message in the browser. We successfully embedded a JavaScript variable in the output of the component (the JSX):



Inside the curly brackets `{ }` we can add any JavaScript statement.

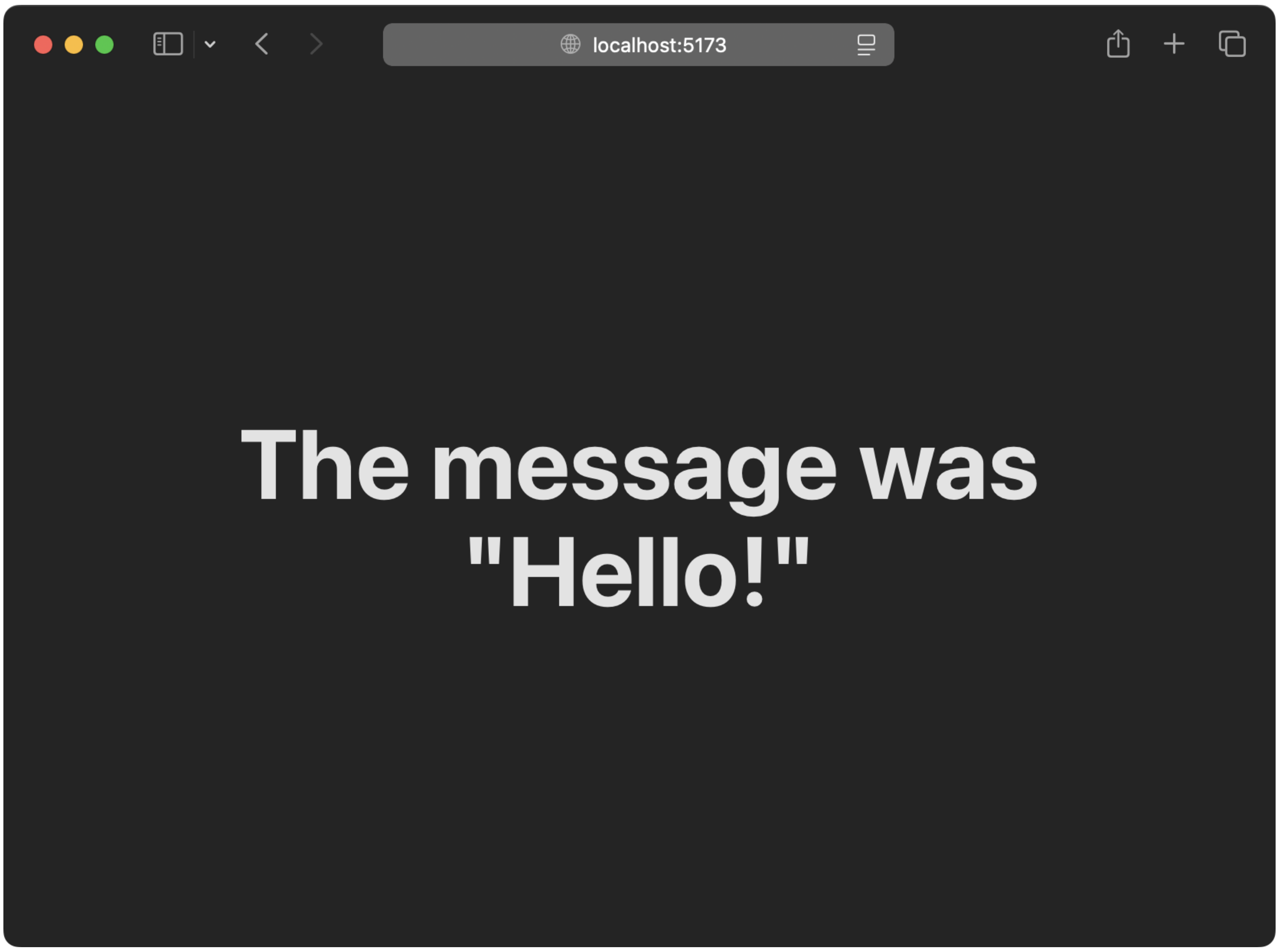
For example, this is a common statement you will find in JSX. We have a ternary operator where we define a condition (`message === 'Hello!'`), and we print one value if the condition is true, or another value (the content of `message` in this case) if the condition is false:

```
{
  message === 'Hello!' ?
    'The message was "Hello!"' : message
}
```

Like this:

```
import './App.css'

function App() {
  const message = 'Hello!'

  return (
    <div className='App'>
      <h1>{message === 'Hello!' ? 'The message was "Hello!"' : message}
</h1>
    </div>
  )
}

export default App
```

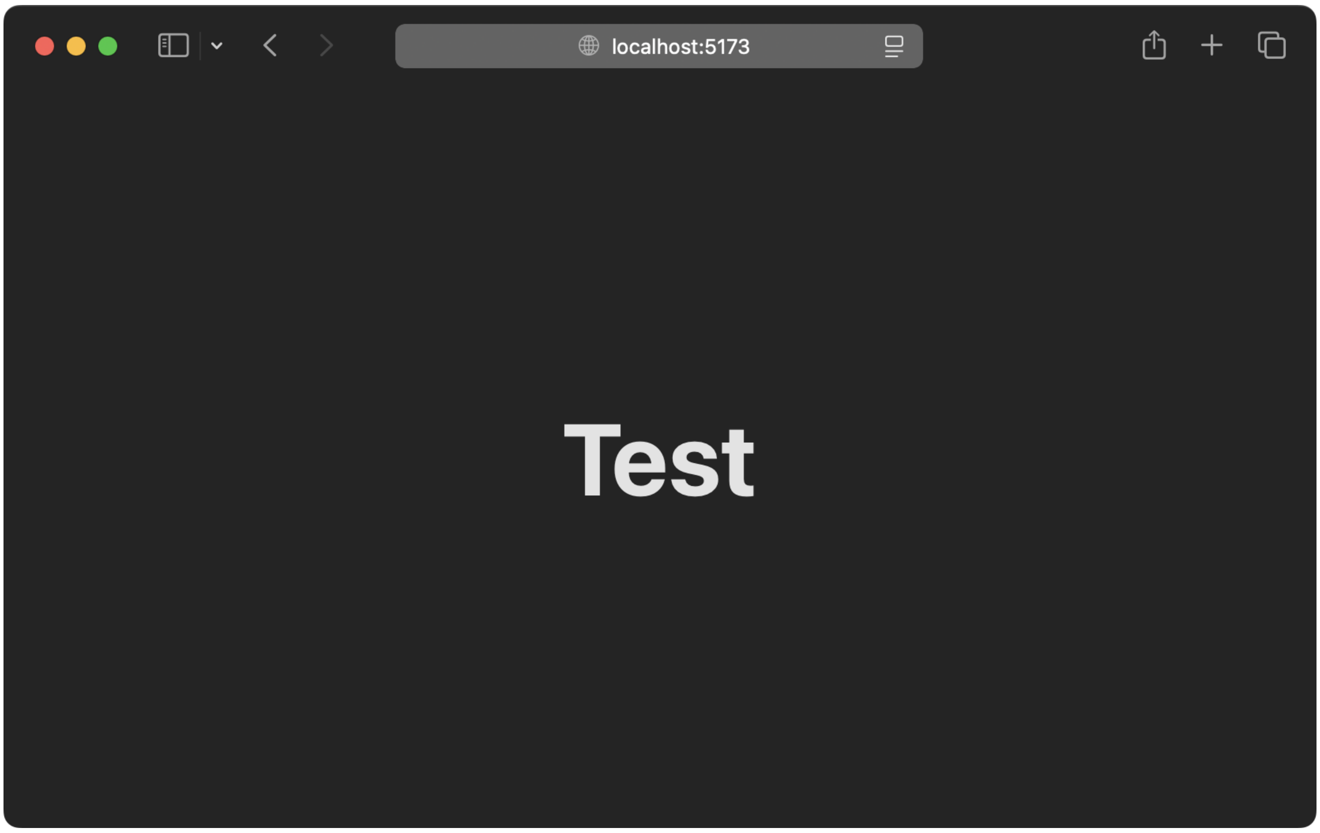
Here's the result:



The message was "Hello!"

If you change the content of the `message` variable, JSX will print something else:

```
import './App.css'

function App() {
  const message = 'Test'

  return (
    <div className='App'>
      <h1>{message === 'Hello!' ? 'The message was "Hello!"' : message}
</h1>
    </div>
  )
}

export default App
```



Test

How to loop inside React JSX

Suppose you have a React component and an `items` array you want to loop over, to print all the “items” you have.

Here’s how you can do it.

In the returned JSX, add a `<ul>` tag to create a list of items:

```
return (
  <ul></ul>
)
```

Inside this list, we add a JavaScript snippet using curly brackets `{}` and inside it we call `items.map()` to iterate over the items.

We pass to the `map()` method a *callback function* that is called for every item of the list.

Inside this function we return a `<li>` (list item) with the value contained in the array, and with a `key` prop that is set to the index of the item in the array. This is needed by React.

```
return (
  <ul>
    {items.map((value, index) => {
      return <li key={index}>{value}</li>
    })}
  </ul>
)
```

You can also use the shorthand form with implicit return:

```
return (
  <ul>
    {items.map((value, index) =>
      <li key={index}>{value}</li>
    )}
  </ul>
)
```

Handling user events in React

React provides an easy way to manage events fired from DOM events like clicks, form events and more.

Let's talk about **click events**, which are pretty simple to digest.

> NOTE: if you’re new to DOM events, check out my **Browser APIs Handbook**.

In the React component’s JSX you can use the `onClick` attribute on any JSX element:

```
<button
  onClick={(event) => {
    /* handle the event */
    console.log('click', event)
  }}
>
  Click here
</button>
```

When the element is clicked, the function passed to the `onClick` attribute is fired.

You can also define this function outside of the JSX, which is handy if the function code is long:

```
const handleClickEvent = (event) => {
  /* handle the event */
  console.log('click', event)
}

function App() {
  return <button onClick={handleClickEvent}>Click here</button>
}
```

When the `click` event is fired on the button, React calls the event handler function.

React supports a vast amount of types of events, like `onKeyUp`, `onFocus`, `onChange`, `onMouseDown`, `onSubmit`, and many more.

Managing state in a React component

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**.

For example if we have a table of addresses we need to display, one state variable could be an array containing all the rows in our table.

Another example is a form.

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 and make heavy use of components state.

The simplest way manage state is by using the `useState` utility provided by React.

It's technically called a **hook**.

You import `useState` from React in this way:

```
import { useState } from 'react'
```

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.

Here’s an example of how to use `useState()`:

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

> `useState()` returns an **array**. The above construct uses a special syntax called array destructuring which we’ll use all the time to extract from the array the first value in the

> `count` variable, and the second value in the `setCount` variable.

This is important: we can't just alter the value of a state variable directly, doing `count++` or `count = count + 1`. ****

We must call its modifier function `setCount()`.

Otherwise, the React component will not update its UI to reflect the changes in 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:

```
import { useState } from 'react'

const Counter = () => {
  const [count, setCount] = useState(0)

  return (
    <div>
      <p>You clicked {count} times</p>
      <button onClick={() => setCount(count + 1)}>Click me</button>
    </div>
  )
}
```

You can add as many `useState()` calls as you want, to create as many state variables as you want, which can hold any value, not just numbers (also objects and arrays are valid):

```
const [count, setCount] = useState(0)
const [name, setName] = useState('John')
```

Component Props in React

When we include a component in another component, we can pass a set of initial values to it.

We call `props` the initial values passed to a component.

We previously created a `WelcomeMessage` component

```
function WelcomeMessage() {
  return <p>Welcome!</p>
}
```

and we used it like this:

```
<WelcomeMessage />
```

This component does not have any initial value passed to it, because it does not have props.

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

```
<WelcomeMessage myprop={'test'} />
```

and inside the component we receive the props as argument:

```
function WelcomeMessage(props) {
  return <p>Welcome!</p>
}
```

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

```
function WelcomeMessage({ myprop }) {
  return <p>Welcome!</p>
}
```

> `myprop` is one of the props contained in the `props` object, like this: `{ myprop: 'test' }`. Using this syntax we only *extract* this prop from the `props` object.

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

```
function WelcomeMessage({ myprop }) {
  return <p>{myprop}</p>
}
```

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 kind of prop is automatically available inside the component with the name `children`.

That variable contains the value of anything that is passed between the opening and closing tags of the component, for example:

```
<WelcomeMessage>Here is some message</WelcomeMessage>
```

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

```
function WelcomeMessage({ children }) {
  return <p>{children}</p>
}
```

Data flow in a React application

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

```
<WelcomeMessage name={'Flavio'} />
```

If you pass a function to the child component, like we do below with an imaginary `setCount()` function, you can also change a state variable defined in the parent component from a child component:

```
<Counter setCount={setCount} count={count} />
```

This is helpful when the parent component is “in charge” of the state variable.

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

```
const Counter = ({ count, setCount }) => {
  return (
    <div>
      <p>You clicked {count} times</p>
      <button onClick={() => **setCount(count + 1)**}>Click me</button>
    </div>
  )
}
```

Here's a full example:

```
import { useState } from 'react'

const Counter = ({ count, setCount }) => {
  return (
    <div>
      <button onClick={() => setCount(count + 1)}>Click me</button>
    </div>
  )
}

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

  return (
    <div>
      <p>You clicked {count} times</p>
      <Counter setCount={setCount} count={count} />
    </div>
  )
}

export default App
```

There are more advanced ways to manage data, which include the built-in *Context API* and libraries like Zustand, MobX or *Redux.*

You should learn one of them too, later, for more complex scenarios, but those introduce more complexity, and many times using the `useState` hook is a good solution.

The useEffect hook

So far we've seen how to manage the state with the `useState` hook.

There's another hook I want to introduce: `useEffect`.

The `useEffect` hook allows components to have access to the lifecycle events of a component and do *something* when the component is first rendered as it's mounted in the DOM, and when it's re-rendered.

> Note that many tutorials use this hook to load data needed by the component. For simple use cases it's fine, but in many cases you'll want to use a state management library, like TanStack Query, SWR, or others.

When you call the `useEffect` hook, you pass it a function.

The function will be run by React when the component is first rendered, and on every subsequent re-render/update.

Remember, a component will re-render when its state or props change.

React first updates the DOM, then calls any function passed to `useEffect()`.

Here is an example:

```
import { useEffect, useState } from 'react'

const CounterWithNameAndSideEffect = () => {
  const [count, setCount] = useState(0)

  useEffect(() => {
    console.log(`You clicked ${count} times`)
  })

  return (
    <div>
      <p>You clicked {count} times</p>
      <button onClick={() => setCount(count + 1)}>Click me</button>
    </div>
  )
}
```

The `useEffect()` function is run on every subsequent re-render/update of the component.

We can tell React to skip a re-render, for performance purposes, by adding a second parameter, an **array** that contains a list of state variables to *watch for*.

React will only re-run the function if one of the items in this array changes:

```
useEffect(() => {
  console.log(`Hi ${name} you clicked ${count} times`)
}, [name, count])
```

Using this syntax you can tell React to only execute the function when the component is first loaded by passing an empty array:

```
useEffect(() => {
  console.log(`Component mounted`)
}, [])
```

> NOTE: In development mode (not in production mode) the function is called 2 times when the component is first rendered, to help you solve possible bugs.

If during the effect you perform an action like opening a network connection for example, you'll need to close that connection when the component unmounts. To do that, you return a function from the function passed to `useEffect()`:

```
useEffect(() => {
  console.log(`connect...`)

  return () => {
    console.log('cleanup connection')
  }
}, [])
```

Install the React Developer Tools

One very useful tool we absolutely need to install (if you haven't already) is the React Developer Tools.

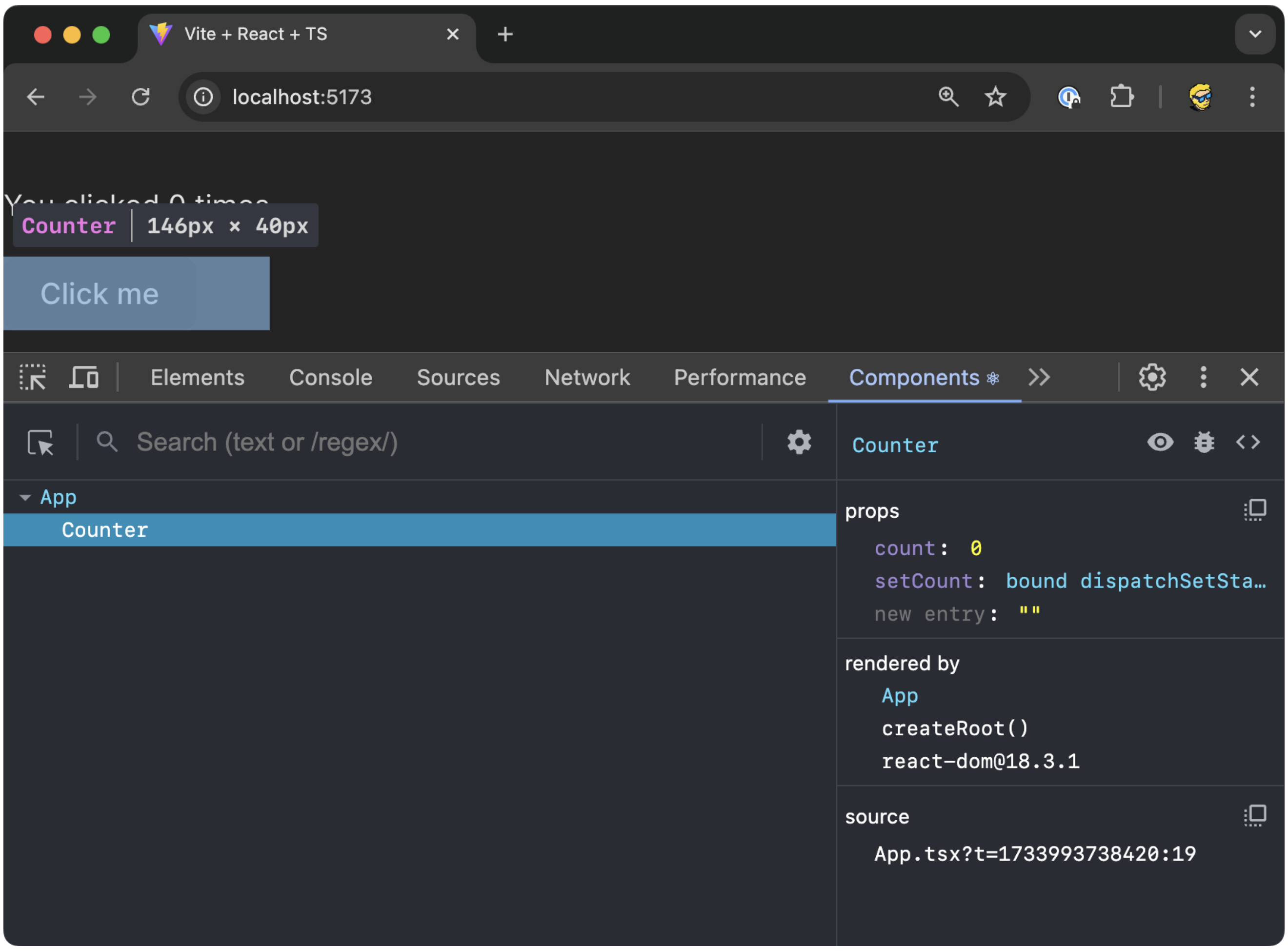
Available for both Chrome and Firefox, the React Developer Tools are an essential instrument you can use to inspect a React application.

They provide an inspector that reveals the React components tree that builds your page, and for each component, you can go and check the props, the state, hooks, and lots more.

Once you have installed the React Developer Tools, you can open the regular browser dev tools (in Chrome, right-click on the page, then click `Inspect`) and you'll find 2 new panels: **Components** and **Profiler**.

Once you're in the Components tab, you'll see the application components list:

In this case, we only have 1 component.

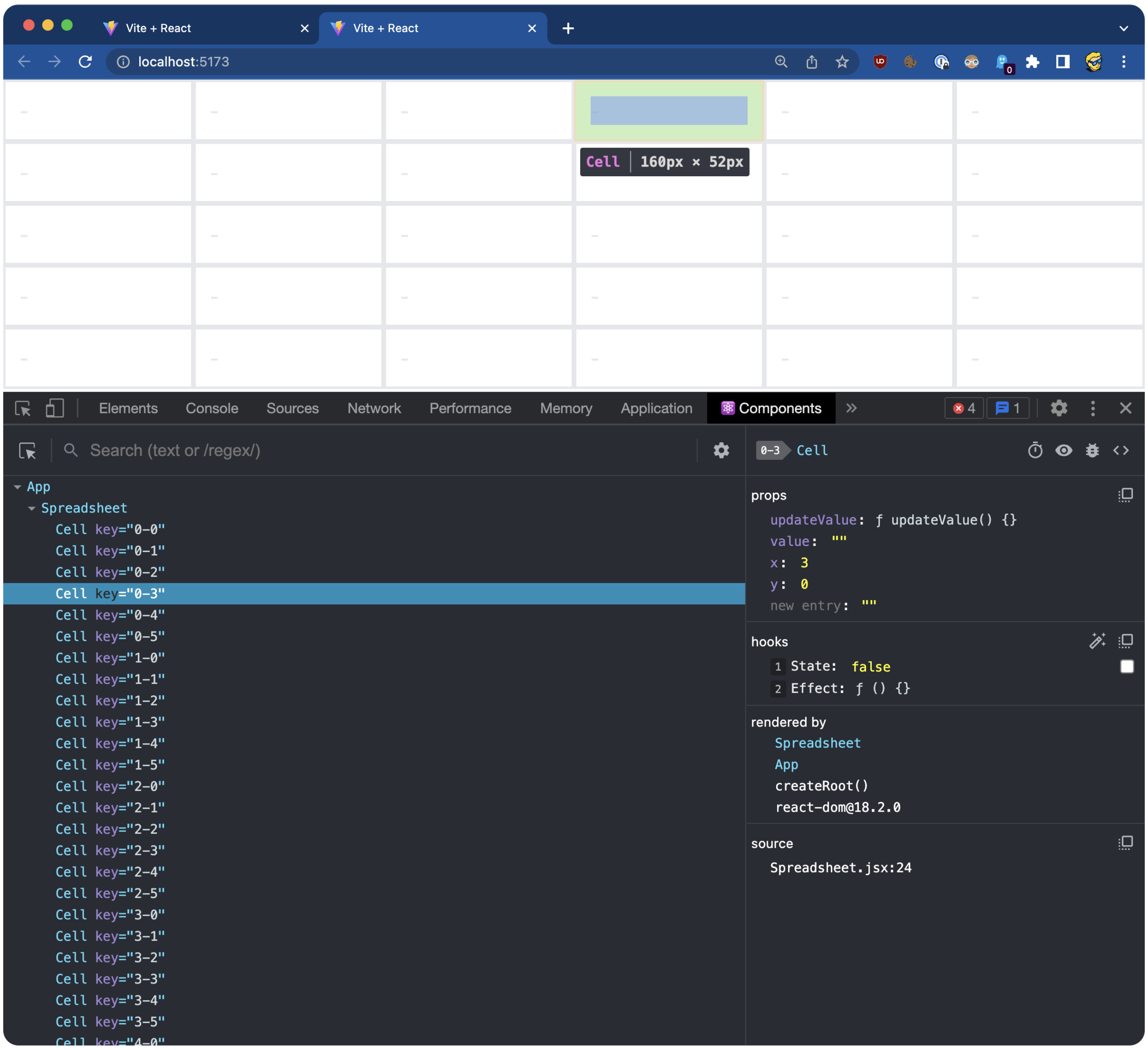


```
App
  Counter
```

```
Counter
props
  count: 0
  setCount: bound dispatchSetSta…
  new entry: ""
rendered by
  App
  createRoot()
  react-dom@18.3.1
source
  App.tsx?t=1733993738420:19
```

As your app grows in size and components, you'll find this panel very useful.

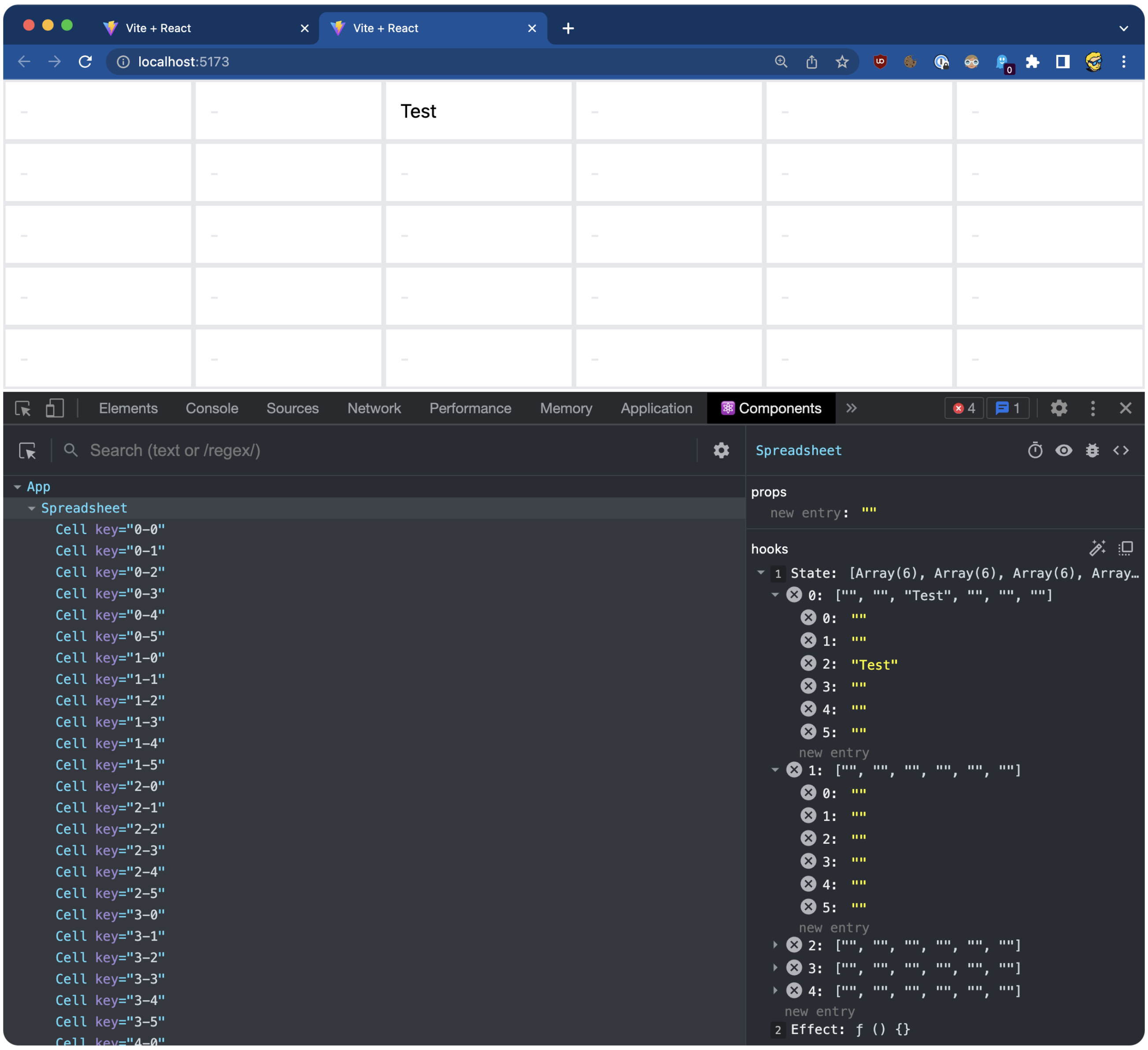
Here's an example of a spreadsheet implemented in React. You can see the `Spreadsheet` component, and inside it a series of `Cell` components, each responsible for a single cell of the spreadsheet:



```
▾ App
  ▾ Spreadsheet
      Cell key="0-0"
      Cell key="0-1"
      Cell key="0-2"
      Cell key="0-3"
      Cell key="0-4"
      Cell key="0-5"
      Cell key="1-0"
      Cell key="1-1"
      Cell key="1-2"
      Cell key="1-3"
      Cell key="1-4"
      Cell key="1-5"
      Cell key="2-0"
      Cell key="2-1"
      Cell key="2-2"
      Cell key="2-3"
      Cell key="2-4"
      Cell key="2-5"
      Cell key="3-0"
      Cell key="3-1"
      Cell key="3-2"
      Cell key="3-3"
      Cell key="3-4"
      Cell key="3-5"
      Cell key="4-0"
```

```
0-3 Cell

props
  updateValue: ƒ updateValue() {}
  value: ""
  x: 3
  y: 0
  new entry: ""

hooks
  1 State: false
  2 Effect: ƒ () {}

rendered by
  Spreadsheet
  App
  createRoot()
  react-dom@18.2.0

source
  Spreadsheet.jsx:24
```

Through the devtools you will be able to inspect the component's data:



```
▾ App
  ▾ Spreadsheet
      Cell key="0-0"
      Cell key="0-1"
      Cell key="0-2"
      Cell key="0-3"
      Cell key="0-4"
      Cell key="0-5"
      Cell key="1-0"
      Cell key="1-1"
      Cell key="1-2"
      Cell key="1-3"
      Cell key="1-4"
      Cell key="1-5"
      Cell key="2-0"
      Cell key="2-1"
      Cell key="2-2"
      Cell key="2-3"
      Cell key="2-4"
      Cell key="2-5"
      Cell key="3-0"
      Cell key="3-1"
      Cell key="3-2"
      Cell key="3-3"
      Cell key="3-4"
      Cell key="3-5"
      Cell key="4-0"
```

```
Spreadsheet
props
  new entry: ""
hooks
▾ 1 State: [Array(6), Array(6), Array(6), Array…
  ▾ 0: ["", "", "Test", "", "", ""]
      0: ""
      1: ""
      2: "Test"
      3: ""
      4: ""
      5: ""
      new entry
  ▾ 1: ["", "", "", "", "", ""]
      0: ""
      1: ""
      2: ""
      3: ""
      4: ""
      5: ""
      new entry
  ▸ 2: ["", "", "", "", "", ""]
  ▸ 3: ["", "", "", "", "", ""]
  ▸ 4: ["", "", "", "", "", ""]
    new entry
  2 Effect: ƒ () {}
```

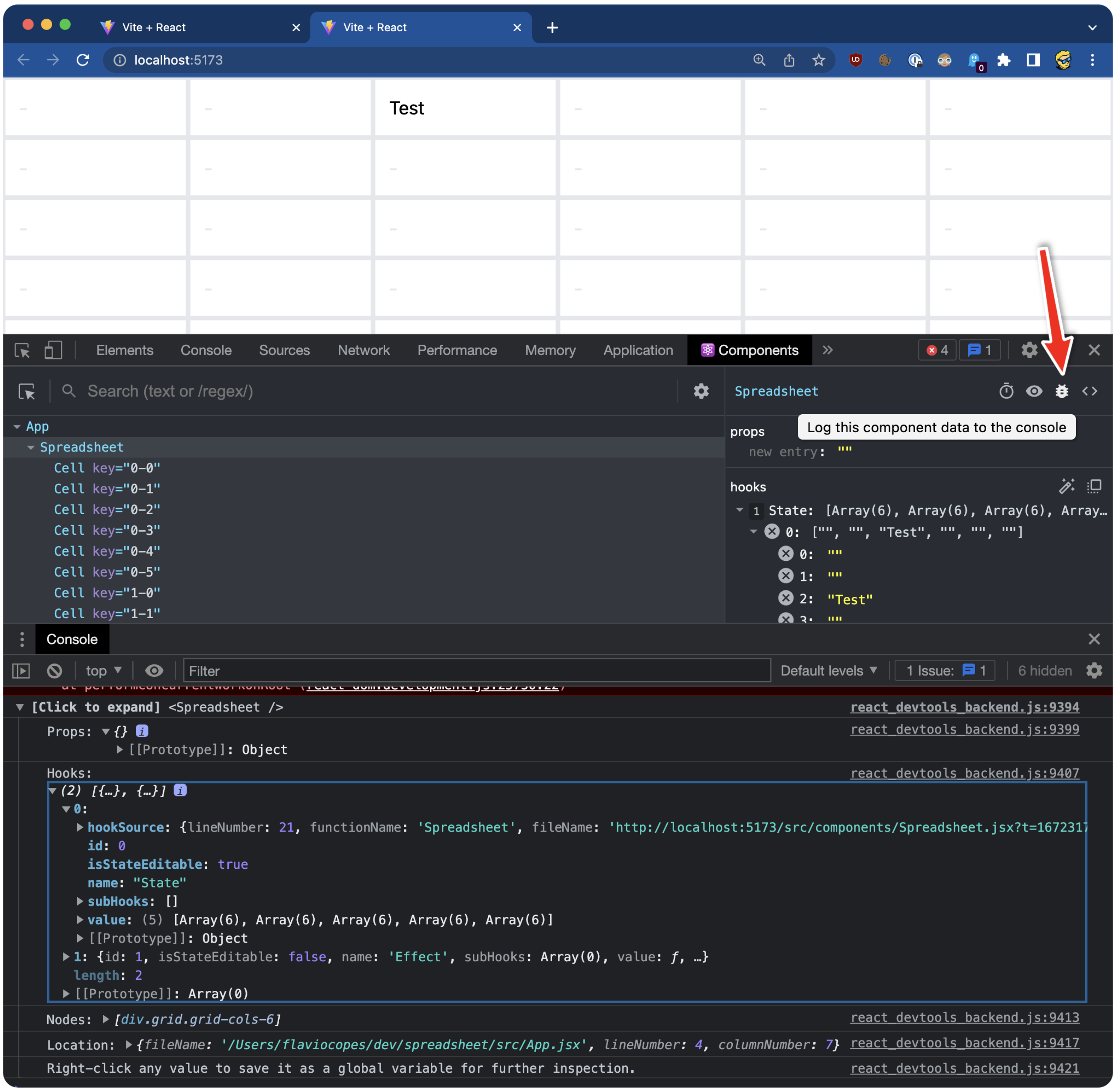
If you move the mouse over the components, you'll see that on the page, the browser will select the parts that are rendered by that component.

If you select any component in the tree, the right panel will show you a reference to **the parent component**, and the props passed to it.

You can easily navigate by clicking around the component names.

You can click the eye icon in the Developer Tools toolbar to inspect the DOM element, and also if you use the first icon, the one with the mouse icon (which conveniently sits under the similar regular DevTools icon), you can hover an element in the browser UI to directly select the React component that renders it.

You can use the bug-shaped 🐞 icon at the top right of the panel to log component data to the console.



```
App
  Spreadsheet
    Cell key="0-0"
    Cell key="0-1"
    Cell key="0-2"
    Cell key="0-3"
    Cell key="0-4"
    Cell key="0-5"
    Cell key="1-0"
    Cell key="1-1"
```

```
props
  new entry: ""
hooks
  1 State: [Array(6), Array(6), Array(6), Array…
    0: ["", "", "Test", "", "", ""]
      0: ""
      1: ""
      2: "Test"
```

```
[Click to expand] <Spreadsheet />                                  react_devtools_backend.js:9394
  Props: {}                                                        react_devtools_backend.js:9399
        [[Prototype]]: Object
  Hooks:                                                           react_devtools_backend.js:9407
  (2) [{…}, {…}]
    0:
      hookSource: {lineNumber: 21, functionName: 'Spreadsheet', fileName: 'http://localhost:5173/src/components/Spreadsheet.jsx?t=1672317
      id: 0
      isStateEditable: true
      name: "State"
      subHooks: []
      value: (5) [Array(6), Array(6), Array(6), Array(6), Array(6)]
      [[Prototype]]: Object
    1: {id: 1, isStateEditable: false, name: 'Effect', subHooks: Array(0), value: ƒ, …}
    length: 2
    [[Prototype]]: Array(0)
  Nodes: [div.grid.grid-cols-6]                                    react_devtools_backend.js:9413
  Location: {fileName: '/Users/flaviocopes/dev/spreadsheet/src/App.jsx', lineNumber: 4, columnNumber: 7} react_devtools_backend.js:9417
  Right-click any value to save it as a global variable for further inspection. react_devtools_backend.js:9421
```

Those are the basics.

This is a super powerful tool that will help you debug your React applications going forward.

Managing forms in React

Forms are one of the few HTML elements that are interactive by default.

They were designed to allow the user to interact with a page.

Common uses of forms?

- Search
- Contact forms
- Shopping carts checkout
- Login and registration
- and more!

Using React we can make our forms much more interactive and less static.

In the simplest form (for complex needs you can use form libraries made for React like react-hook-form or TanStack Form, but in this book we focus on “core React”), we build a form by creating JSX elements.

The difference with React is how we manage the value entered in the form: we associate it to a state variable defined with `useState`, and once it’s time to send the form data to the server, we use this state variable instead of looking up the value from the DOM.

```
import { useState } from 'react'

export default function Form() {
  const [username, setUsername] = useState('')

  return (
    <form>
      Username:
      <input
        type='text'
        value={username}
      />
    </form>
  )
}
```

When an element state changes in a form field managed by a component, we track it using the `onChange` attribute.

```
import { useState } from 'react'

export default function Form() {
  const [username, setUsername] = useState('')

  return (
    <form>
      Username:
      <input
        type='text'
        value={username}
        **onChange={(event) => {
          setUsername(event.target.value)
        }}**
      />
    </form>
  )
}
```

We use the `onSubmit` attribute on the form to call the `handleSubmit` method when the form is submitted:

```
import { useState } from 'react'

export default function Form() {
  const [username, setUsername] = useState('')

  **const handleSubmit = async (event) => {
    event.preventDefault()

  }**

  return (
    <form **onSubmit={handleSubmit}**>
      Username:
      <input
        type='text'
        value={username}
        onChange={(event) => {
          setUsername(event.target.value)
        }}
      />
    </form>
  )
}
```

`event.preventDefault()` in the form submit handler is needed so the browser does not perform the default form submit behavior which is to do a GET request to the same URL the page is on (ultimately this means just reloading the page).

Validation in a form can be handled in the `onChange` event handler, where you have access to the old value of the state and the new one. You can check the new value and if not valid reject the updated value (and communicate it in some way to the user). And you can also do validation in the `onSubmit` event handler.

For example, you can check the fields were filled with data you interpret to be valid.

Once you're ready to send the data, you can use a `fetch` request to a POST API endpoint:

```
import { useState } from 'react'

export default function Form() {
  const [username, setUsername] = useState('')

  const handleSubmit = async (event) => {
    event.preventDefault()
```

```
      const response = await fetch('/api/form', {
        body: JSON.stringify({
          username,
        }),
        headers: {
          'Content-Type': 'application/json',
        },
        method: 'POST',
      })
    }

    return (
      <form onSubmit={handleSubmit}>
        Username:
        <input
          type='text'
          value={username}
          onChange={(event) => {
            setUsername(event.target.value)
          }}
        />
      </form>
    )
  }
```

Where to go from here

Mastering the topics explained in this book is a great step towards your goal of learning React.

What should you learn next?

Start building some simple React applications. For example, build a simple counter or interact with a public API.

Learn more theory about the Virtual DOM, writing declarative code, unidirectional data flow, immutability, and composition.

Learn how to manage state using the Context API, useContext, Zustand.

Learn React Server Components.

Learn an application framework built on top of React, like Next.js or TanStack Start.

The most important part: just use React in your projects, and learn along the way.