React Tutorial

Setting up a React Environment

First, make sure you have Node.js installed. You can check by running this in your terminal:

node -v

If Node.js is installed, you will get a result with the version number:

v22.15.0

If not, you will need to install Node.js.

Install a Build Tool (Vite)

When you have Node.js installed, you can start creating a React application by choosing a build tool.

We will use the [Vite](https://vite.dev/" \t "_blank) build tool in this tutorial.

Run this command to install Vite:

npm install -g create-vite

If the installation was a success, you will get a result like this:

added 1 package in 649ms

Create a React Application

Run this command to create a React application named my-react-app:

npm create vite@latest my-react-app -- --template react

If you get this message, just press y and press Enter to continue:

Need to install the following packages:  
create-vite@6.5.0  
Ok to proceed? (y)

If the creation was a success, you will get a result like this:

> npx  
> create-vite my-react-app --template react  
  
|  
o  Scaffolding project in C:\Users\stale\my-react-app...  
|  
—  Done. Now run:  
  
  cd my-react-app  
  npm install  
  npm run dev

Install Dependencies

As the result above suggests, navigate to your new react application directory:

cd my-react-app

And run this command to install dependencies:

npm install

Which will result in this:

added 154 packages, and audited 155 packages in 8s  
  
33 packages are looking for funding  
  run `npm fund` for details  
  
found 0 vulnerabilities

Run the React Application

Now you are ready to run your first *real* React application!

Run this command to run the React application my-react-app:

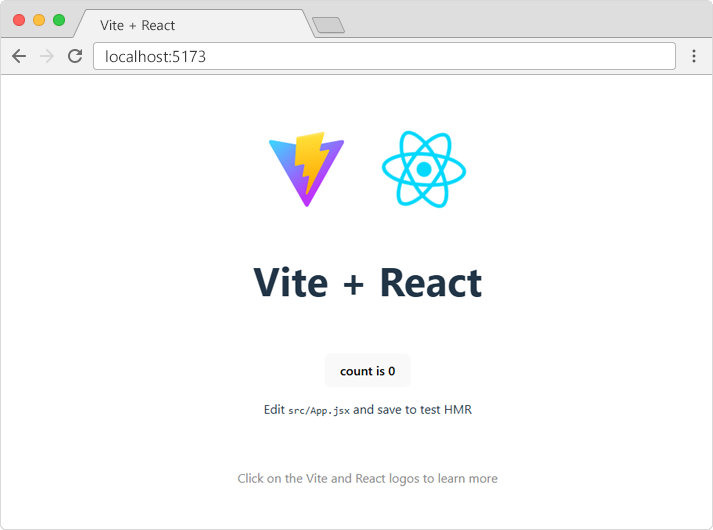
npm run dev

Which will result in this:

VITE v6.3.5  ready in 217 ms  
  
➜ Local: <http://localhost:5173/>  
➜ Network: use --host to expose  
➜ press h + enter to show help

A new browser window will pop up with your newly created React App! If not, open your browser and type localhost:5173 in the address bar.

The result:



## Modify the React App

Look in the my-react-app directory, and you will find a src folder. Inside the src folder there is a file called App.js, open it and it will look like this:

### **Example**

This is the default content of the App.jsx file in the src folder:

App.jsx

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://vitejs.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.jsx</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

Try replacing the **entire file content** with the code below and save the file.

### **Example**

Replace all the content of the App.jsx file with the code below:

App.jsx

function App() {

return (

<div className="App">

<h1>Hello World!</h1>

</div>

);

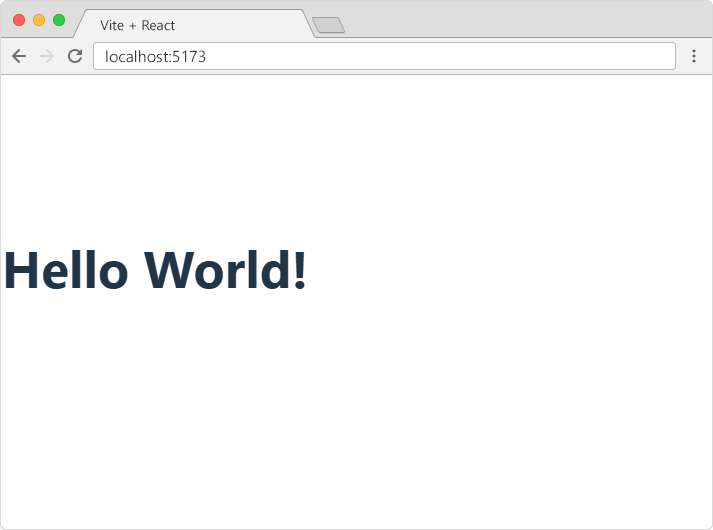
}

export default App;

See the changes in the browser when you click Save.

Notice that the changes are visible immediately after you save the file, you do not have to reload the browser!

The result:



## The Container

React uses a container to render HTML in a web page.

Typically, this container is a <div id="root"></div> element in the index.html file.

If you have followed the steps in the previous chapter, you should have a file called index.html in the root directory of your project:

### **Example**

The default content of the index.html file:

index.html

<!doctype html>

<html lang="en">

<head>

<meta charset="UTF-8" />

<link rel="icon" type="image/svg+xml" href="/vite.svg" />

<meta name="viewport" content="width=device-width, initial-scale=1.0" />

<title>Vite + React</title>

</head>

<body>

<div id="root"></div>

<script type="module" src="/src/main.jsx"></script>

</body>

</html>

To better understand the content of the index.html file, let's remove all the code we don't need.

### **Example**

The index.html file should now look like this:

index.html

<!doctype html>

<html lang="en">

<body>

<div id="root"></div>

<script type="module" src="/src/main.jsx"></script>

</body>

</html>

The file is now stripped from unnecessary code, and we can concentrate on learning React without any disturbing elements.

## The createRoot Function

The createRoot function is located in the main.jsx file in the src folder, and is a built-in function that is used to create a root node for a React application.

### **Example**

The default content of the src/main.jsx file:

main.jsx

import { StrictMode } from 'react'

import { createRoot } from 'react-dom/client'

import './index.css'

import App from './App.jsx'

createRoot(document.getElementById('root')).render(

<StrictMode>

<App />

</StrictMode>

)

The createRoot() function takes one argument, an HTML element.

The purpose of the function is to define the HTML element where a React component should be displayed.

To better understand the createRoot function, let's remove unnecessary code and write our own "Hello React!" example:

### **Example**

The src/main.jsx file should now look like this:

main.jsx

import { createRoot } from 'react-dom/client'

createRoot(document.getElementById('root')).render(

<h1>Hello React!</h1>

)

If you save the file, the result in the browser will look like this:

# https://www.w3schools.com/react/react_helloreact.png

## The render Method

Did you notice the render method?

The render method defines what to render in the HTML container.

The result is displayed in the <div id="root"> element.

### **Example**

Display a paragraph inside the "root" element:

main.jsx

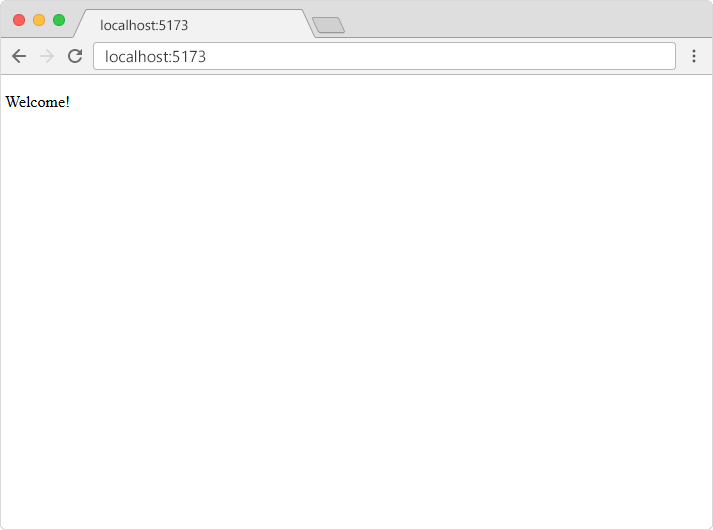
import { createRoot } from 'react-dom/client'

createRoot(document.getElementById('root')).render(

<p>Welcome!</p>

)

The result will look like this:



**Note:** the element id does not have to be "root", but this is the standard convention.

## Show React

W3Schools has its own "Show React" tool where we will show the result of the code we explain in the tutorial.

Click the "Run Example" button to see the result:

### **Example**

The same example shown in our "Show React" tool:

main.jsx

import { createRoot } from 'react-dom/client'

createRoot(document.getElementById('root')).render(

<p>Welcome!</p>

)

## The HTML Code

The HTML code in this tutorial uses JSX which allows you to write HTML tags inside the JavaScript code:

Don't worry if the syntax is unfamiliar, you will learn more about JSX later in this tutorial.

### **Example**

Create a variable that contains HTML code and display it in the "root" node:

main.jsx

import { createRoot } from 'react-dom/client'

const myelement = (

<table>

<tr>

<th>Name</th>

</tr>

<tr>

<td>John</td>

</tr>

<tr>

<td>Elsa</td>

</tr>

</table>

);

createRoot(document.getElementById('root')).render(

myelement

)

## The Root Node

The root node is the HTML element where you want to display the result.

It is like a container for content, managed by React.

It does NOT have to be a <div> element and it does NOT have to have the id='root':

### **Example**

The root node can be called whatever you like.

Display the result in the <header id="sandy"> element:

index.html

main.jsx

<!doctype html>

<html lang="en">

<body>

<header id="sandy"></header>

<script type="module" src="/src/main.jsx"></script>

</body>

</html>

## What is ES6?

ES6 stands for ECMAScript 6.

ECMAScript was created to standardize JavaScript, and ES6 is the 6th version of ECMAScript, it was published in 2015, and is also known as ECMAScript 2015.

## Why Should I Learn ES6?

React uses ES6, and you should be familiar with some of the new features like:

* [Classes](https://www.w3schools.com/react/react_es6_classes.asp)
* [Arrow Functions](https://www.w3schools.com/react/react_es6_arrow.asp)
* [Variables](https://www.w3schools.com/react/react_es6_variables.asp) (let, const, var)
* [Array Methods](https://www.w3schools.com/react/react_es6_array_methods.asp) like .map()
* [Destructuring](https://www.w3schools.com/react/react_es6_destructuring.asp)
* [Modules](https://www.w3schools.com/react/react_es6_modules.asp)
* [Ternary Operator](https://www.w3schools.com/react/react_es6_ternary.asp)
* [Spread Operator](https://www.w3schools.com/react/react_es6_spread.asp)

## Classes

ES6 introduced classes.

A class is a type of function, but instead of using the keyword function to initiate it, we use the keyword class, and the properties are assigned inside a constructor() method.

### **Example**

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 standard naming convention for classes.

Now you can create objects using the Car class:

### **Example**

Create an object called "mycar" based on the Car class:

class Car {

constructor(name) {

this.brand = name;

}

}

const mycar = new Car("Ford");

**Note:** The constructor function is called automatically when the object is initialized.

## Method in Classes

You can add your own methods in a class:

### **Example**

Create a method named "present":

class Car {

constructor(name) {

this.brand = name;

}

present() {

return 'I have a ' + this.brand;

}

}

const mycar = new Car("Ford");

mycar.present();

As you can see in the example above, you call the method by referring to the object's method name followed by parentheses (parameters would go inside the parentheses).

## Class Inheritance

To create a class inheritance, use the extends keyword.

A class created with a class inheritance inherits all the methods from another class:

### **Example**

Create a class named "Model" which will inherit the methods from the "Car" class:

class Car {

constructor(name) {

this.brand = name;

}

present() {

return 'I have a ' + this.brand;

}

}

class Model extends Car {

constructor(name, mod) {

super(name);

this.model = mod;

}

show() {

return this.present() + ', it is a ' + this.model

}

}

const mycar = new Model("Ford", "Mustang");

mycar.show();

The super() method refers to the parent class.

By calling the super() method in the constructor method, we call the parent's constructor method and get access to the parent's properties and methods.

## Arrow Functions

Arrow functions allow us to write shorter function syntax:

### **Example**

Before:

hello = function() {

return "Hello World!";

}

### **Example**

With Arrow Function:

hello = () => {

return "Hello World!";

}

It gets shorter! If the function has only one statement, and the statement returns a value, you can remove the brackets and the return keyword:

### **Example**

Arrow Functions Return Value by Default:

hello = () => "Hello World!";

**Note:** This works only if the function has only one statement.

If you have parameters, you pass them inside the parentheses:

### **Example**

Arrow Function With Parameters:

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

In fact, if you have only one parameter, you can skip the parentheses as well:

### **Example**

Arrow Function Without Parentheses:

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.

Let us take a look at two examples to understand the difference.

Both examples call a method twice, first when the page loads, and once again when the user clicks a button.

The first example uses a regular function, and the second example uses an arrow function.

The result shows that the first example returns two different objects (window and button), and the second example returns the Header object twice.

### **Example**

With a regular function, this represents the object that called the function:

class Header {

constructor() {

this.color = "Red";

}

//Regular function:

changeColor = function() {

document.getElementById("demo").innerHTML += this;

}

}

const myheader = new Header();

//The window object calls the function:

window.addEventListener("load", myheader.changeColor);

//A button object calls the function:

document.getElementById("btn").addEventListener("click", myheader.changeColor);

### **Example**

With an arrow function, this represents the Header object no matter who called the function:

class Header {

constructor() {

this.color = "Red";

}

//Arrow function:

changeColor = () => {

document.getElementById("demo").innerHTML += this;

}

}

const myheader = new Header();

//The window object calls the function:

window.addEventListener("load", myheader.changeColor);

//A button object calls the function:

document.getElementById("btn").addEventListener("click", myheader.changeColor);

## Variables

Before ES6 there was only one way of defining your variables: with the var keyword. If you did not define them, they would be assigned to the global object. Unless you were in strict mode, then you would get an error if your variables were undefined.

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

### **Example**

var

var x = 5.6;

If you use var outside of a function, it belongs to the global scope.

If you use var inside of a function, it belongs to that function.

If you use var inside of a block, i.e. a for loop, the variable is still available outside of that block.

var has a function scope, not a block scope.

### **Example**

let

let x = 5.6;

let is the block scoped version of var, and is limited to the block (or expression) where it is defined.

If you use let inside of a block, i.e. a for loop, the variable is only available inside of that loop.

let has a block scope.

### **Example**

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.

The keyword const is a bit misleading.

It does not define a constant value. It defines a constant reference to a value.

Because of this you can NOT:

* Reassign a constant value
* Reassign a constant array
* Reassign a constant object

But you CAN:

* Change the elements of constant array
* Change the properties of constant object

# **JavaScript Array map()**

## The map() Method

The map() method creates a new array with the results of calling a function for every array element.

### **Example**

Multiply each number by 2:

const numbers = [1, 2, 3, 4];

const doubled = numbers.map(x => x \* 2);

## map() in React

The map() method is commonly used in React to render lists of elements:

### **Example**

const fruitlist = ['apple', 'banana', 'cherry'];

function MyList() {

return (

<ul>

{fruitlist.map(fruit =>

<li key={fruit}>{fruit}</li>

)}

</ul>

);

}

**Note:** When using map() in React to create list items, each item needs a unique key prop.

## map() with Objects

You can also use map() with arrays of objects:

### **Example**

const users = [

{ id: 1, name: 'John', age: 30 },

{ id: 2, name: 'Jane', age: 25 },

{ id: 3, name: 'Bob', age: 35 }

];

function UserList() {

return (

<ul>

{users.map(user =>

<li key={user.id}>

{user.name} is {user.age} years old

</li>

)}

</ul>

);

}

## map() Parameters

The map() method takes three parameters:

* currentValue - The current element being processed
* index - The index of the current element (optional)
* array - The array that map was called upon (optional)

### **Example**

const fruitlist = ['apple', 'banana', 'cherry'];

function App() {

return (

<ul>

{fruitlist.map((fruit, index, array) => {

return (

<li key={fruit}>

Number: {fruit}, Index: {index}, Array: {array}

</li>

);

})}

</ul>

);

}

**Note:** The map() method always returns a new array. It does not modify the original array.