Definition

React is, in our opinion, the premier way to build big, fast Web apps with JavaScript. It has scaled very well for us at Facebook and Instagram. – Official Website

React is a declarative, efficient, and flexible JavaScript library for building user interfaces. It lets you compose complex UIs from small and isolated pieces of code called “components”.

Installation – Local setup

Pre-requisite

[Node >= 8.10 and npm >= 5.6](https://nodejs.org/en/)

Editor – VS Code

Setup 🡪

npx create-react-app my-app

cd my-app

npm start

Introducing JSX 🡪

const element = <h1>Hello, world!</h1>;

It is called JSX, and it is a syntax extension to JavaScript. We recommend using it with React to describe what the UI should look like. JSX may remind you of a template language, but it comes with the full power of JavaScript.

React embraces the fact that rendering logic is inherently coupled with other UI logic: how events are handled, how the state changes over time, and how the data is prepared for display.

Instead of artificially separating technologies by putting markup and logic in separate files, React [separates concerns](https://en.wikipedia.org/wiki/Separation_of_concerns) with loosely coupled units called “components” that contain both. (Like Angular does)

Embedding Expressions in JSX

const name = 'React';

const element = <h1>Hello, {name}</h1>;

ReactDOM.render(

element,

document.getElementById('root')

);

You can put any valid [JavaScript expression](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Expressions_and_Operators#Expressions) inside the curly braces in JSX.

function formatName(user) {

return user.firstName + ' ' + user.lastName;

}

const user = {

firstName: 'Harper',

lastName: 'Perez'

};

const element = (

<h1> Hello, {formatName(user)}!</h1>

);

ReactDOM.render(

element,

document.getElementById('root')

);

**JSX is an Expression Too**

After compilation, JSX expressions become regular JavaScript function calls and evaluate to JavaScript objects.

This means that you can use JSX inside of if statements and for loops, assign it to variables, accept it as arguments, and return it from functions:

function getGreeting(user) {

if (user) {

return <h1>Hello, {formatName(user)}!</h1>;}

return <h1>Hellow, Stranger.</h1>;

}

### Specifying Attributes with JSX

### You may use quotes to specify string literals as attributes:

const element = <div tabIndex="0"></div>;

You may also use curly braces to embed a JavaScript expression in an attribute:

const element = <img src={user.avatarUrl}></img>;

Don’t put quotes around curly braces when embedding a JavaScript expression in an attribute. You should either use quotes (for string values) or curly braces (for expressions), but not both in the same attribute.

### Specifying Children with JSX

const element = (

<div>

<h1>Hello!</h1>

<h2>Good to see you here.</h2>

</div>

);

### JSX Represents Objects

Babel compiles JSX down to React.createElement() calls.

These two examples are identical:

const element = (

<h1 className="greeting">

Hello, world!

</h1>

);

const element = React.createElement(

'h1',

{className: 'greeting'},

'Hello, world!'

);

React.createElement() performs a few checks to help you write bug-free code but essentially it creates an object like this:

const element = {

type: 'h1',

props: {

className: 'greeting',

children: 'Hello, world!'

}

};

These objects are called “React elements”. You can think of them as descriptions of what you want to see on the screen. React reads these objects and uses them to construct the DOM and keep it up to date.

**React Rendering Element**

Elements are the smallest building blocks of React apps.

An element describes what you want to see on the screen:

const element = <h1>Hello, world</h1>;

## Rendering an Element into the DOM

Let’s say there is a <div> somewhere in your HTML file:

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

We call this a “root” DOM node because everything inside it will be managed by React DOM.

Applications built with just React usually have a single root DOM node. If you are integrating React into an existing app, you may have as many isolated root DOM nodes as you like.

To render a React element into a root DOM node, pass both to [ReactDOM.render()](https://reactjs.org/docs/react-dom.html#render):

const element = <h1>Hello, world</h1>;

ReactDOM.render(element, document.getElementById('root'));

It displays “Hello, world” on the page.

## Updating the Rendered Element

React elements are [immutable](https://en.wikipedia.org/wiki/Immutable_object). Once you create an element, you can’t change its children or attributes. An element is like a single frame in a movie: it represents the UI at a certain point in time.

The only way to update the UI is to create a new element, and pass it to [ReactDOM.render()](https://reactjs.org/docs/react-dom.html#render).

Consider this ticking clock example:

function tick() {

const element = (

<div>

<h1>Hello, world!</h1>

<h2>It is {new Date().toLocaleTimeString()}.</h2>

</div>

);

ReactDOM.render(element, document.getElementById('root'));

}

setInterval(tick, 1000);

It calls [ReactDOM.render()](https://reactjs.org/docs/react-dom.html#render) every second from a [setInterval()](https://developer.mozilla.org/en-US/docs/Web/API/WindowTimers/setInterval) callback.

## React Only Updates What’s Necessary

React DOM compares the element and its children to the previous one, and only applies the DOM updates necessary to bring the DOM to the desired state.

Lets verify last example.

Even though we create an element describing the whole UI tree on every tick, only the text node whose contents have changed gets updated by React DOM.

# Components and Props

Components let you split the UI into independent, reusable pieces, and think about each piece in isolation

Conceptually, components are like JavaScript functions. They accept arbitrary inputs (called “props”) and return React elements describing what should appear on the screen.

React lets you define components as classes or functions.

The simplest way to define a component is to write a JavaScript function:

function Welcome(props) {

return <h1>Hello, {props.name}</h1>;

}

Components defined as classes currently provide more features. To define a React component class, you need to extend React.Component:

class Welcome extends React.Component {

render() {

return <h1>Hello, {this.props.name}</h1>;

}

}

The only method you must define in a React.Component subclass is called [render()](https://reactjs.org/docs/react-component.html#render). All the other methods described on this page are optional.

This is in similar way to write class as ES6,  you may use the create-react-class module or a similar custom abstraction instead.

The simplest way to define a component is to write a JavaScript function:

function Welcome(props) {

return <h1>Hello, {props.name}</h1>;

}

Or in ES6 ways

class Welcome extends React.Component {

render() {

return <h1>Hello, {this.props.name}</h1>;

}

}

## Rendering a Component

## Previously, we only encountered React elements that represent DOM tags:

const element = <div />;

However, elements can also represent user-defined components:

const element = <Welcome name="Sara" />;

When React sees an element representing a user-defined component, it passes JSX attributes and children to this component as a single object. We call this object “props”.

For example, this code renders “Hello, Sara” on the page:

function Welcome(props) {

return <h1>Hello, {props.name}</h1>;

}

const element = <Welcome name="Sara" />;

ReactDOM.render(

element,

document.getElementById('root')

);

Let’s recap what happens in this example:

1. We call ReactDOM.render() with the <Welcome name="Sara" /> element.
2. React calls the Welcome component with {name: 'Sara'} as the props.
3. Our Welcome component returns a <h1>Hello, Sara</h1> element as the result.
4. React DOM efficiently updates the DOM to match <h1>Hello, Sara</h1>.

**Always start component names with a capital letter.**

React treats components starting with lowercase letters as DOM tags. For example, <div /> represents an HTML div tag, but <Welcome /> represents a component and requires Welcome to be in scope.

## Composing Components

Components can refer to other components in their output. This lets us use the same component abstraction for any level of detail. A button, a form, a dialog, a screen: in React apps, all those are commonly expressed as components.

For example, we can create an App component that renders Welcome many times:

function Welcome(props) {

return <h1>Hello, {props.name}</h1>;

}

function App() {

return (

<div>

<Welcome name="Sara" />

<Welcome name="Cahal" />

<Welcome name="Edite" />

</div>

);

}

ReactDOM.render(

<App />,

document.getElementById('root')

);

## Extracting Components

Don’t be afraid to split components into smaller components.

For example, consider this Comment component:

function Comment(props) {

return (

<div className="Comment">

<div className="UserInfo">

<img className="Avatar"

src={props.author.avatarUrl}

alt={props.author.name}

/>

<div className="UserInfo-name">

{props.author.name}

</div>

</div>

<div className="Comment-text">

{props.text}

</div>

<div className="Comment-date">

{formatDate(props.date)}

</div>

</div>

);

}

his component can be tricky to change because of all the nesting, and it is also hard to reuse individual parts of it. Let’s extract a few components from it.

First, we will extract Avatar:

function Avatar(props) {

return (

<img className="Avatar"

src={props.user.avatarUrl}

alt={props.user.name}

/>

);

}

We can now simplify Comment a tiny bit:

function Comment(props) {

return (

<div className="Comment">

<div className="UserInfo">

<Avatar user={props.author} />

<div className="UserInfo-name">

{props.author.name}

</div>

</div>

<div className="Comment-text">

{props.text}

</div>

<div className="Comment-date">

{formatDate(props.date)}

</div>

</div>

);

}

Next, we will extract a UserInfo component that renders an Avatar next to the user’s name:

function UserInfo(props) {

return (

<div className="UserInfo">

<Avatar user={props.user} />

<div className="UserInfo-name">

{props.user.name}

</div>

</div>

);

}

This lets us simplify Comment even further:

function Comment(props) {

return (

<div className="Comment">

<UserInfo user={props.author} />

<div className="Comment-text">

{props.text}

</div>

<div className="Comment-date">

{formatDate(props.date)}

</div>

</div>

);

}

## Props are Read-Only

Whether you declare a component [as a function or a class](https://reactjs.org/docs/components-and-props.html#function-and-class-components), it must never modify its own props.

Consider this sum function:

function sum(a, b) {

return a + b;

}

Such functions are called [“pure”](https://en.wikipedia.org/wiki/Pure_function) because they do not attempt to change their inputs, and always return the same result for the same inputs.

In contrast, this function is impure because it changes its own input:

function withdraw(account, amount) {

account.total -= amount;

}

React is pretty flexible but it has a single strict rule:

**All React components must act like pure functions with respect to their props.**

# State and Lifecycle

Consider the ticking clock example from [one of the previous class](https://reactjs.org/docs/rendering-elements.html#updating-the-rendered-element). In [Rendering Elements](https://reactjs.org/docs/rendering-elements.html#rendering-an-element-into-the-dom), we have only learned one way to update the UI. We call ReactDOM.render() to change the rendered output:

function tick() {

const element = (

<div>

<h1>Hello, world!</h1>

<h2>It is {new Date().toLocaleTimeString()}.</h2>

</div>

);

ReactDOM.render(element, document.getElementById('root') );}

setInterval(tick, 1000);

In this section, we will learn how to make the Clock component truly reusable and encapsulated. It will set up its own timer and update itself every second.

We can start by encapsulating how the clock looks:

function Clock(props) {

return (

<div>

<h1>Hello, world!</h1>

<h2>It is {props.date.toLocaleTimeString()}.</h2>

</div>

);

}

function tick() {

ReactDOM.render(

<Clock date={new Date()} />,

document.getElementById('root')

);

}

setInterval(tick, 1000);

However, it misses a crucial requirement: the fact that the Clock sets up a timer and updates the UI every second should be an implementation detail of the Clock.

Ideally we want to write this once and have the Clock update itself:

ReactDOM.render(

<Clock />,

document.getElementById('root')

);

To implement this, we need to add “state” to the Clock component.

State is similar to props, but it is private and fully controlled by the component.

## Converting a Function to a Class

You can convert a function component like Clock to a class in five steps:

1. Create an [ES6 class](https://developer.mozilla.org/en/docs/Web/JavaScript/Reference/Classes), with the same name, that extends React.Component.
2. Add a single empty method to it called render().
3. Move the body of the function into the render() method.
4. Replace props with this.props in the render() body.
5. Delete the remaining empty function declaration.

class Clock extends React.Component {

render() {

return (

<div>

<h1>Hello, world!</h1>

<h2>It is {this.props.date.toLocaleTimeString()}.</h2>

</div>

);

}

Clock is now defined as a class rather than a function.

The render method will be called each time an update happens, but as long as we render <Clock /> into the same DOM node, only a single instance of the Clock class will be used. This lets us use additional features such as local state and lifecycle methods.

## Adding Local State to a Class

We will move the date from props to state in three steps:

1. Replace this.props.date with this.state.date in the render() method:

class Clock extends React.Component {

render() {

return (

<div>

<h1>Hello, world!</h1>

<h2>It is {this.state.date.toLocaleTimeString()}.</h2>

</div>

);

}}

1. Add a [class constructor](https://developer.mozilla.org/en/docs/Web/JavaScript/Reference/Classes#Constructor) that assigns the initial this.state:

class Clock extends React.Component {

constructor(props) {

super(props);

this.state = {date: new Date()};

}

render() {

return (

<div>

<h1>Hello, world!</h1>

<h2>It is {this.state.date.toLocaleTimeString()}.</h2>

</div>

);}}

Note how we pass props to the base constructor:

constructor(props) {

super(props);

this.state = {date: new Date()};

}

Class components should always call the base constructor with props.

1. Remove the date prop from the <Clock /> element:

ReactDOM.render(

<Clock />,

document.getElementById('root')

);

We will later add the timer code back to the component itself.

The result looks like this:

class Clock extends React.Component {

constructor(props) {

super(props);

this.state = {date: new Date()};

}

render() {

return (

<div>

<h1>Hello, world!</h1>

<h2>It is {this.state.date.toLocaleTimeString()}.</h2>

</div>

);

}

}

ReactDOM.render(

<Clock />,

document.getElementById('root')

);

Next, we’ll make the Clock set up its own timer and update itself every second.

## Adding Lifecycle Methods to a Class

In applications with many components, it’s very important to free up resources taken by the components when they are destroyed.

We want to [set up a timer](https://developer.mozilla.org/en-US/docs/Web/API/WindowTimers/setInterval) whenever the Clock is rendered to the DOM for the first time. This is called “mounting” in React.

We also want to [clear that timer](https://developer.mozilla.org/en-US/docs/Web/API/WindowTimers/clearInterval) whenever the DOM produced by the Clock is removed. This is called “unmounting” in React.

We can declare special methods on the component class to run some code when a component mounts and unmounts:

class Clock extends React.Component {

constructor(props) {

super(props);

this.state = {date: new Date()};

}

componentDidMount() {

}

componentWillUnmount() {

}

render() {

return (

<div>

<h1>Hello, world!</h1>

<h2>It is {this.state.date.toLocaleTimeString()}.</h2>

</div>

);

}

}

These methods are called “lifecycle methods”.

The componentDidMount() method runs after the component output has been rendered to the DOM. This is a good place to set up a timer:

componentDidMount() {

this.timerID = setInterval(

() => this.tick(),

1000

);

}

Note how we save the timer ID right on this (this.timerID).

While this.props is set up by React itself and this.state has a special meaning, you are free to add additional fields to the class manually if you need to store something that doesn’t participate in the data flow (like a timer ID).

We will tear down the timer in the componentWillUnmount() lifecycle method:

componentWillUnmount() {

clearInterval(this.timerID);

}

Finally, we will implement a method called tick() that the Clock component will run every second.

It will use this.setState() to schedule updates to the component local state:

class Clock extends React.Component {

constructor(props) {

super(props);

this.state = {date: new Date()};

}

componentDidMount() {

this.timerID = setInterval(

() => this.tick(),

1000

);

}

componentWillUnmount() {

clearInterval(this.timerID);

}

tick() {

this.setState({

date: new Date()

});

}

render() {

return (

<div>

<h1>Hello, world!</h1>

<h2>It is {this.state.date.toLocaleTimeString()}.</h2>

</div>

);

}

}

ReactDOM.render(

<Clock />,

document.getElementById('root')

);

Now the clock ticks every second.

Let’s quickly recap what’s going on and the order in which the methods are called:

1. When <Clock /> is passed to ReactDOM.render(), React calls the constructor of the Clock component. Since Clock needs to display the current time, it initializes this.state with an object including the current time. We will later update this state.
2. React then calls the Clock component’s render() method. This is how React learns what should be displayed on the screen. React then updates the DOM to match the Clock’s render output.
3. When the Clock output is inserted in the DOM, React calls the componentDidMount() lifecycle method. Inside it, the Clock component asks the browser to set up a timer to call the component’s tick() method once a second.
4. Every second the browser calls the tick() method. Inside it, the Clock component schedules a UI update by calling setState() with an object containing the current time. Thanks to the setState() call, React knows the state has changed, and calls the render() method again to learn what should be on the screen. This time, this.state.date in the render() method will be different, and so the render output will include the updated time. React updates the DOM accordingly.
5. If the Clock component is ever removed from the DOM, React calls the componentWillUnmount() lifecycle method so the timer is stopped.

Using State Correctly

There are three things you should know about setState().

Do Not Modify State Directly

For example, this will not re-render a component:

// Wrong

this.state.comment = 'Hello';

Instead, use setState():

// Correct

this.setState({comment: 'Hello'});

The only place where you can assign this.state is the constructor.

### State Updates May Be Asynchronous

React may batch multiple setState() calls into a single update for performance.

Because this.props and this.state may be updated asynchronously, you should not rely on their values for calculating the next state.

For example, this code may fail to update the counter:

// Wrong

this.setState({

counter: this.state.counter + this.props.increment,

});

To fix it, use a second form of setState() that accepts a function rather than an object. That function will receive the previous state as the first argument, and the props at the time the update is applied as the second argument:

// Correct

this.setState((state, props) => ({

counter: state.counter + props.increment

}));

### State Updates are Merged

When you call setState(), React merges the object you provide into the current state.

For example, your state may contain several independent variables:

constructor(props) {

super(props);

this.state = {

posts: [],

comments: []

};

}

Then you can update them independently with separate setState() calls:

componentDidMount() {

fetchPosts().then(response => {

this.setState({

posts: response.posts

});

});

fetchComments().then(response => {

this.setState({

comments: response.comments

});

});

}

The merging is shallow, so this.setState({comments}) leaves this.state.posts intact, but completely replaces this.state.comments.

## The Data Flows Down

Neither parent nor child components can know if a certain component is stateful or stateless, and they shouldn’t care whether it is defined as a function or a class.

This is why state is often called local or encapsulated. It is not accessible to any component other than the one that owns and sets it.

A component may choose to pass its state down as props to its child components:

<h2>It is {this.state.date.toLocaleTimeString()}.</h2>

This also works for user-defined components:

<FormattedDate date={this.state.date} />

The FormattedDate component would receive the date in its props and wouldn’t know whether it came from the Clock’s state, from the Clock’s props, or was typed by hand:

function FormattedDate(props) {

return <h2>It is {props.date.toLocaleTimeString()}.</h2>;

}

This is commonly called a “top-down” or “unidirectional” data flow. Any state is always owned by some specific component, and any data or UI derived from that state can only affect components “below” them in the tree.

If you imagine a component tree as a waterfall of props, each component’s state is like an additional water source that joins it at an arbitrary point but also flows down.

Parent

Child

Grand Child

gets props add state => gets props add state => gets props add state

To show that all components are truly isolated, we can create an App component that renders three <Clock>s:

function App() {

return (

<div>

<Clock />

<Clock />

<Clock />

</div>

);

}

ReactDOM.render(

<App />,

document.getElementById('root')

);

Each Clock sets up its own timer and updates independently.

In React apps, whether a component is stateful or stateless is considered an implementation detail of the component that may change over time. You can use stateless components inside stateful components, and vice versa.

# Handling Events

Handling events with React elements is very similar to handling events on DOM elements. There are some syntax differences:

* React events are named using camelCase, rather than lowercase.
* With JSX you pass a function as the event handler, rather than a string.

For Example, in HTML

<button onclick="activateLasers()">

Activate Lasers

</button>

In React

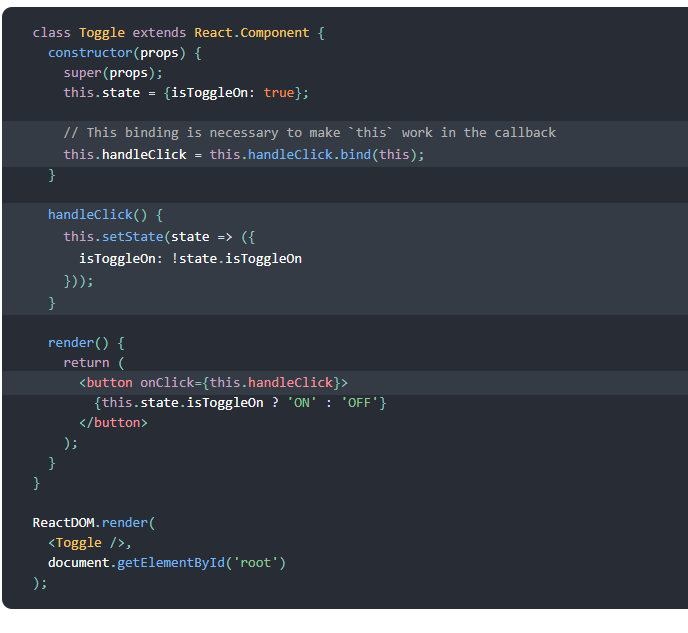
<button onClick={activateLasers}>

Activate Lasers

</button>

When using React, you generally don’t need to call addEventListener to add listeners to a DOM element after it is created. Instead, just provide a listener when the element is initially rendered.

When you define a component using an [ES6 class](https://developer.mozilla.org/en/docs/Web/JavaScript/Reference/Classes), a common pattern is for an event handler to be a method on the class. For example, this Toggle component renders a button that lets the user toggle between “ON” and “OFF” states:

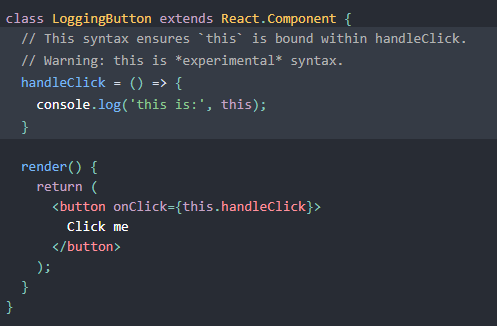


You have to be careful about the meaning of this in JSX callbacks. In JavaScript, class methods are not [bound](https://developer.mozilla.org/en/docs/Web/JavaScript/Reference/Global_objects/Function/bind) by default. If you forget to bind this.handleClick and pass it to onClick, this will be undefined when the function is actually called.

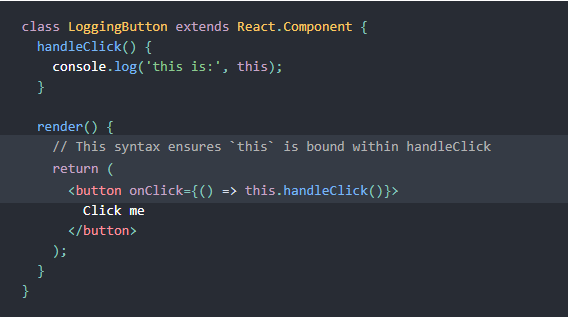
This is not React-specific behavior; it is a part of [how functions work in JavaScript](https://www.smashingmagazine.com/2014/01/understanding-javascript-function-prototype-bind/). Generally, if you refer to a method without () after it, such as onClick={this.handleClick}, you should bind that method.

If calling bind annoys you, there are two ways you can get around this. If you are using the experimental [public class fields syntax](https://babeljs.io/docs/plugins/transform-class-properties/), you can use class fields to correctly bind callbacks:

First->

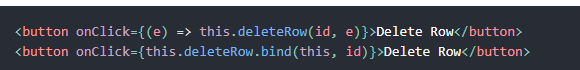


Second way is -> If you aren’t using class fields syntax, you can use an [arrow function](https://developer.mozilla.org/en/docs/Web/JavaScript/Reference/Functions/Arrow_functions) in the callback



## Passing Arguments to Event Handlers

Inside a loop, it is common to want to pass an extra parameter to an event handler. For example, if id is the row ID, either of the following would work:

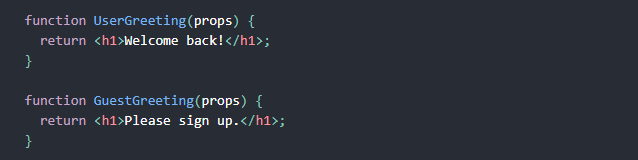


# Conditional Rendering

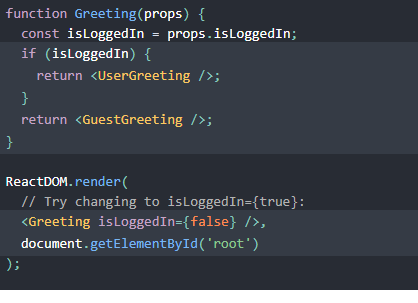
In React, you can create distinct components that encapsulate behavior you need. Then, you can render only some of them, depending on the state of your application.

Conditional rendering in React works the same way conditions work in JavaScript. Use JavaScript operators like [if](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/if...else) or the [conditional operator](https://developer.mozilla.org/en/docs/Web/JavaScript/Reference/Operators/Conditional_Operator) to create elements representing the current state, and let React update the UI to match them.

Example

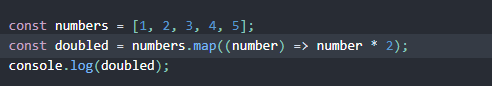


We’ll create a Greeting component that displays either of these components depending on whether a user is logged in:



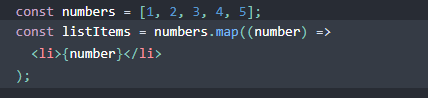
# Lists and Keys

First, let’s review how you transform lists in JavaScript.

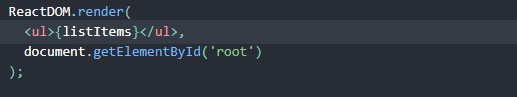


### Rendering Multiple Components

Below, we loop through the numbers array using the JavaScript [map()](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/map) function. We return a <li> element for each item. Finally, we assign the resulting array of elements to listItems:

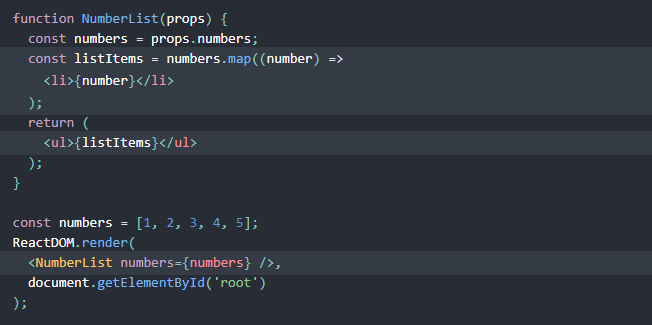


We include the entire listItems array inside a <ul> element, and [render it to the DOM](https://reactjs.org/docs/rendering-elements.html#rendering-an-element-into-the-dom):



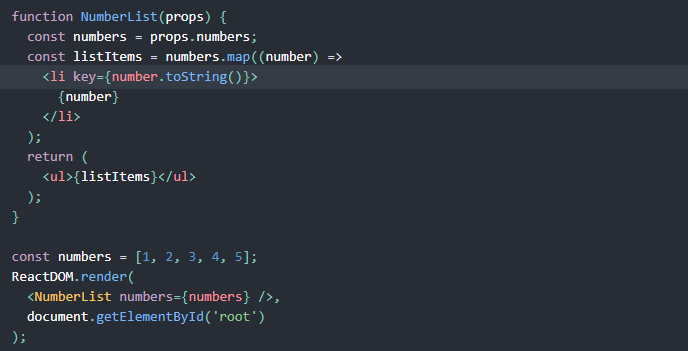
### Basic List Component

We can refactor the previous example into a component that accepts an array of numbers and outputs a list of elements.



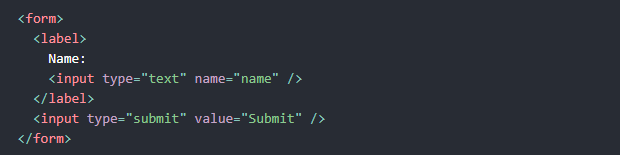
When you run this code, you’ll be given a warning that a key should be provided for list items. A “key” is a special string attribute you need to include when creating lists of elements. We’ll discuss why it’s important in the next section.

Let’s assign a key to our list items inside numbers.map() and fix the missing key issue.



# Forms

HTML form elements work a little bit differently from other DOM elements in React, because form elements naturally keep some internal state. For example, this form in plain HTML accepts a single name:



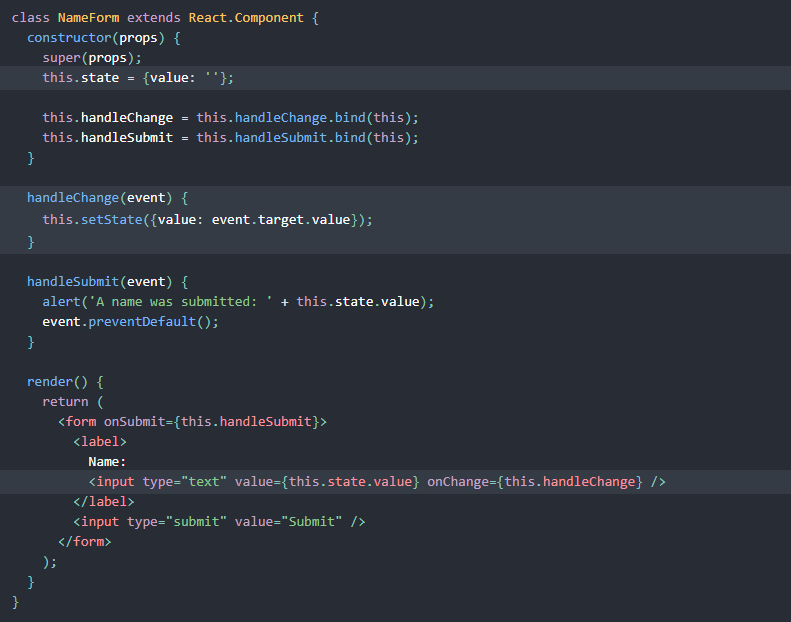
this form has the default HTML form behavior of browsing to a new page when the user submits the form. If you want this behavior in React, it just works. But in most cases, it’s convenient to have a JavaScript function that handles the submission of the form and has access to the data that the user entered into the form. The standard way to achieve this is with a technique called “controlled components”.

## Controlled Components

In HTML, form elements such as <input>, <textarea>, and <select> typically maintain their own state and update it based on user input. In React, mutable state is typically kept in the state property of components, and only updated with [setState()](https://reactjs.org/docs/react-component.html#setstate).

We can combine the two by making the React state be the “single source of truth”. Then the React component that renders a form also controls what happens in that form on subsequent user input. An input form element whose value is controlled by React in this way is called a “controlled component”.

For example, if we want to make the previous example log the name when it is submitted, we can write the form as a controlled component:



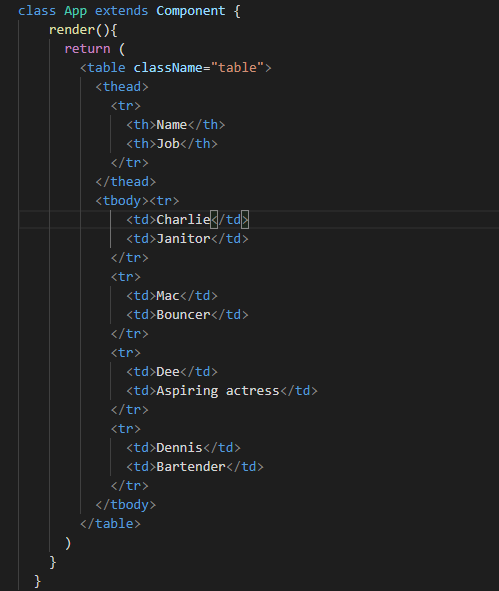
TextArea:

In HTML, a <textarea> element defines its text by its children:

In React, a <textarea> uses a value attribute instead. This way, a form using a <textarea> can be written very similarly to a form that uses a single-line input:

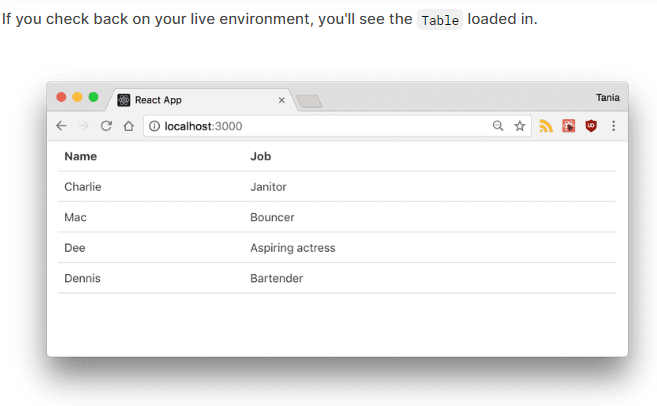
## Class Components revisit

Let's create another component. We're going to create a table. Make Table.js, and fill it with the following data.



In index.js include App

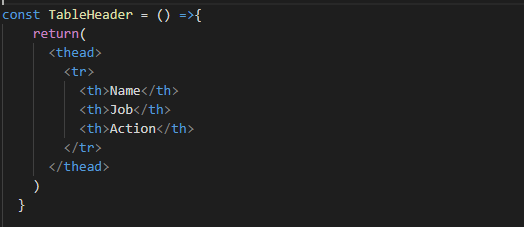




### Simple Components

The other type of component in React is the **simple component**, which is a function. This component doesn't use the class keyword. Let's take our Table and make two simple components for it - a table header, and a table body.

We're going to use ES6 arrow functions to create these simple components. First, the table header.

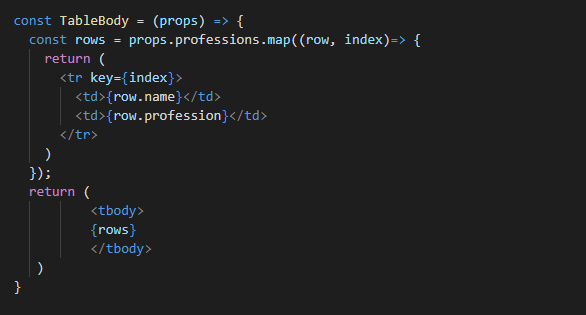


And table body as

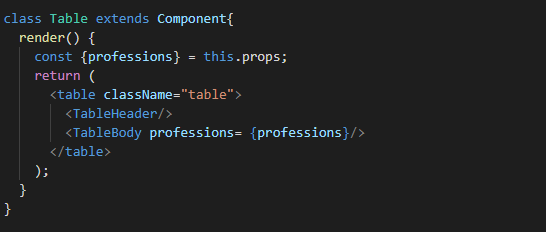
## Props

Right now, we have a cool Table component, but the data is being hard-coded. One of the big deals about React is how it handles data, and it does so with properties, referred to as **props**, and with state. Now, we'll focus on handling data with props.

**Here we are getting table data using props**



Create Table component as



Then App component will be changed to



Next we will include an event handler for every row using props

In Header we have added one more column called action to delete the row

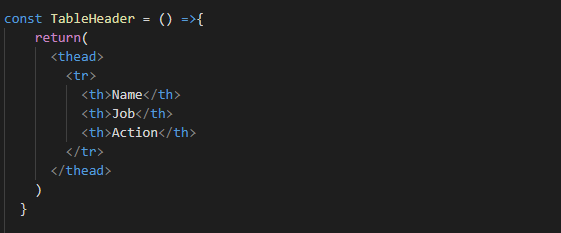
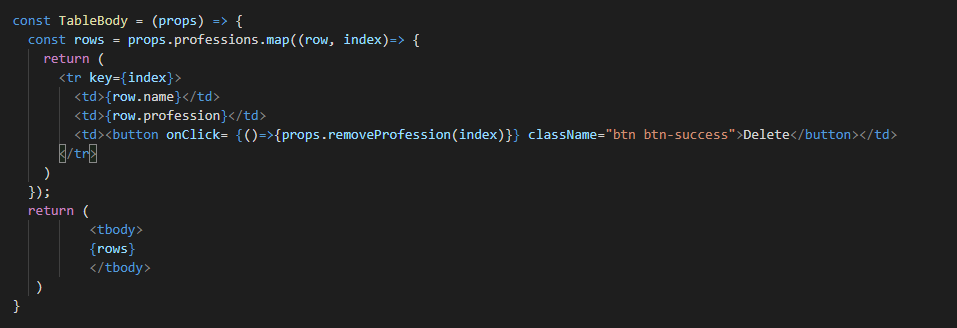
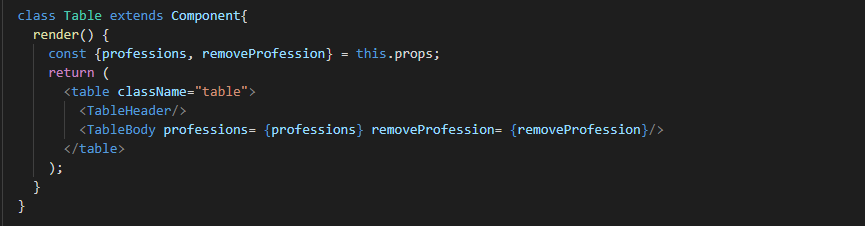


Table body be as



And Table will pass two props now



Here we are creating data on state and passing it to component as props



## Submitting Form Data

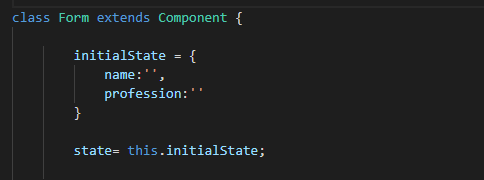
Now we have data stored in state, and we can remove any item from the state. However, what if we wanted to be able to add new data to state? In a real world application, you'd more likely start with empty state and add to it, such as with a to-do list or a shopping cart.

Before anything else, let's remove all the hard-coded data from state.professions, as we'll be updating that through the form now.

# 

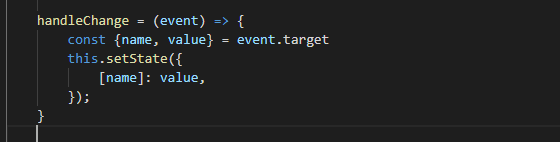
Now let's go ahead and create a Form component in a new file called Form.js.

We're going to set the initial state of the Form to be an object with some empty properties, and assign that initial state to this.state.



**Our goal for this form will be to update the state of Form every time a field is changed in the form, and when we submit, all that data will pass to the App state, which will then update the Table.**

First, we'll make the function that will run every time a change is made to an input. The event will be passed through, and we'll set the state of Form to have the name (key) and value of the inputs.

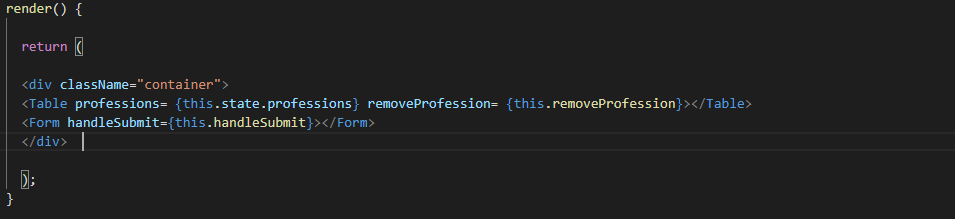


Let's get this working before we move on to submitting the form. In the render, let's get our two properties from state, and assign them as the values that correspond to the proper form keys. We'll run the handleChange() method as the onChange of the input, and finally we'll export the Form component.

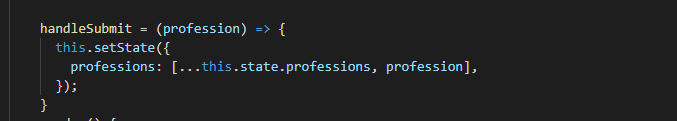


In App.js, we can render the form below the table.

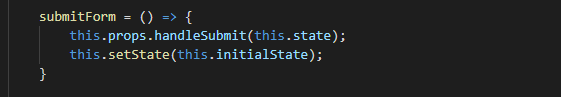
Import Form from ‘form’

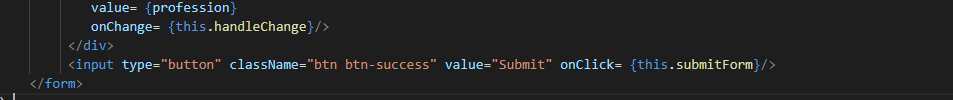


Last step is to allow us to actually submit that data and update the parent state. We'll create a function called handleSubmit() on App that will update the state by taking the existing this.state.professions and adding the new profession parameter, using the [ES6 spread operator](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Spread_syntax).



Now in Form, we'll create a method called submitForm() that will call that function, and pass the Form state through as the profession  parameter we defined earlier. It will also reset the state to the initial state, to clear the form after submit.





That’s It

# AJAX – **Asynchronous JavaScript and XML**

## Using XMLHttpRequest

To send an HTTP request, create an XMLHttpRequest object, open a URL, and send the request. After the transaction completes, the object will contain useful information such as the response body and the [HTTP status](https://developer.mozilla.org/en-US/docs/Web/HTTP/Status) of the result.

function reqListener () {

console.log(this.responseText);

}

var oReq = new XMLHttpRequest();

oReq.addEventListener("load", reqListener);

oReq.open("GET", "http://www.example.org/example.txt");

oReq.send();

## Types of requests

A request made via XMLHttpRequest can fetch the data in one of two ways, asynchronously or synchronously. The type of request is dictated by the optional async argument (the third argument) that is set on the [XMLHttpRequest.open()](https://developer.mozilla.org/en-US/docs/Web/API/XMLHttpRequest/open) method. If this argument is true or not specified, the XMLHttpRequest is processed asynchronously, otherwise the process is handled synchronously.

An [EventHandler](https://developer.mozilla.org/en-US/docs/Web/API/EventHandler) that is called whenever the readyState attribute changes. The callback is called from the user interface thread. The **XMLHttpRequest.onreadystatechange** property contains the event handler to be called when the [readystatechange](https://developer.mozilla.org/en-US/docs/Web/API/Document/readystatechange_event) event is fired, that is every time the [readyState](https://developer.mozilla.org/en-US/docs/Web/API/XMLHttpRequest/readyState) property of the [XMLHttpRequest](https://developer.mozilla.org/en-US/docs/Web/API/XMLHttpRequest) changes.

const xhr = new XMLHttpRequest(),

method = "GET",

url = "https://developer.mozilla.org/";

xhr.open(*method*, *url*, true);

xhr.onreadystatechange = function () {

  // In local files, status is 0 upon success in Mozilla Firefox

if(xhr.readyState === XMLHttpRequest.DONE) {

  var status = xhr.status;

if (status === 0 || (status >= 200 && status < 400)) {

  // The request has been completed successfully

  console.log(xhr.responseText);

  } else {

// Oh no! There has been an error with the request!

  }

  }

};

xhr.send();

The **XMLHttpRequest.readyState** property returns the state an XMLHttpRequest client is in. An XHR client exists in one of the following states:

|  |  |  |
| --- | --- | --- |
| **Value** | **State** | **Description** |
| 0 | UNSENT | Client has been created. open() not called yet. |
| 1 | OPENED | open() has been called. |
| 2 | HEADERS\_RECEIVED | send() has been called, and headers and status are available. |
| 3 | LOADING | Downloading; responseText holds partial data. |
| 4 | DONE | The operation is complete. |

# Pulling In API data

One very common usage of React is pulling in data from an API.

As a little test, we can create a new Api.js file, and create a new App in there. A public API we can test with is the [Wikipedia API](https://en.wikipedia.org/w/api.php), and I have a [URL endpoint right here](https://en.wikipedia.org/w/api.php?action=opensearch&search=Seona+Dancing&format=json&origin=*) for a random\* search. You can go to that link to see the API

[https://en.wikipedia.org/w/api.php?action=opensearch&search=Narendra+modi&format=json&origin=\*](https://en.wikipedia.org/w/api.php?action=opensearch&search=Narendra+modi&format=json&origin=*)

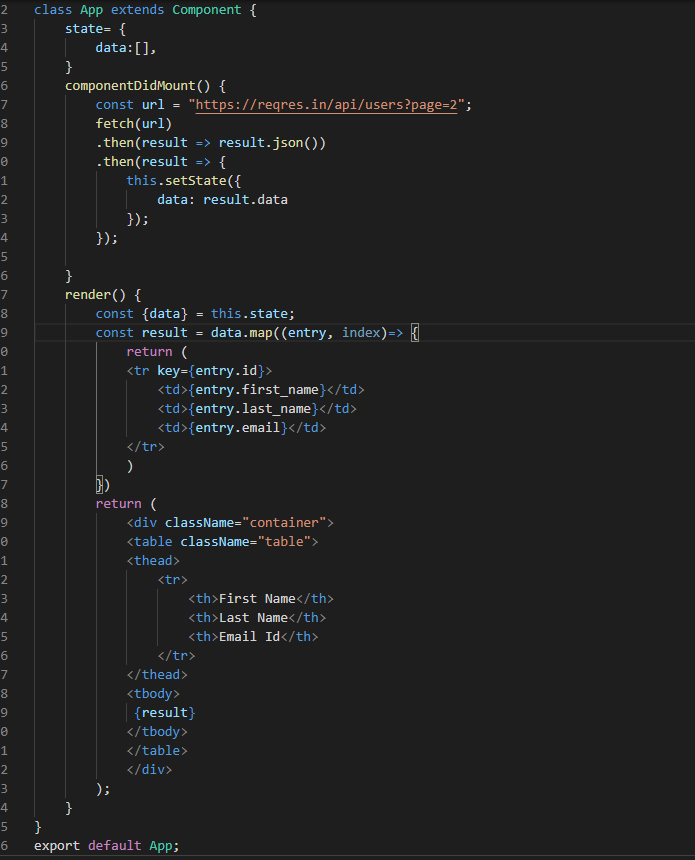
https://reqres.in/api/users?page=2

The [Fetch API](https://developer.mozilla.org/en-US/docs/Web/API/Fetch_API) is a newer built-in feature of JavaScript that makes working with requests and responses easier.



Note that with Fetch, even a 404 or 500 error will not return an error. Only a network error or a request not completing will throw an error.

Example – API.js



Output

