

Written by **Tania Rascia** on August 20, 2018

# React Tutorial: An Overview and Walkthrough

[api](#)[javascript](#)[react](#)[tutorial](#)

I've been hearing about React since I first started learning JavaScript, but I'll admit I took one look at it and it scared me. I saw what looked like a bunch of HTML mixed with JavaScript and thought, *isn't this what we've been trying to avoid? What's the big deal with React?*

Instead, I focused on just learning vanilla JavaScript and working with jQuery in a professional setting. After a few frustrated, failed attempts to get started with React, I finally started to get it, and I began to see why I might want to use React instead of vanilla JS or jQuery.

I tried to condense everything I've learned into a nice introduction to share with you, so here it is.



with React. If you've never used JavaScript or the DOM at all before, for example, I would get more familiar with those before trying to tackle React.

Here are what I consider to be React prerequisites.

- Basic familiarity with **HTML & CSS**.
- Basic knowledge of **JavaScript** and programming.
- Basic understanding of **the DOM**.
- Familiarity with **ES6 syntax and features**.
- **Node.js and npm** installed globally.

## GOALS

- Learn about essential React concepts and related terms, such as Babel, Webpack, JSX, components, props, state, and lifecycle.
- Build a very simple React app that demonstrates the above concepts.

Here's the source and a live demo of the end result.

- [View Source on GitHub](#)
- [View Demo](#)

# What is React?

- React is a JavaScript library - one of the most popular ones, with **over 100,000 stars on GitHub**.
- React is not a framework (unlike Angular, which is more opinionated).
- React is an open-source project created by Facebook.



One of the most important aspects of React is the fact that you can create **components**, which are like custom, reusable HTML elements, to quickly and efficiently build user interfaces. React also streamlines how data is stored and handled, using **state** and **props**.

We'll go over all of this and more throughout the article, so let's get started.

## Setup and Installation

There are a few ways to set up React, and I'll show you two so you get a good idea of how it works.

### Static HTML File

This first method is not a popular way to set up React and is not how we'll be doing the rest of our tutorial, but it will be familiar and easy to understand if you've ever used a library like jQuery, and it's the least scary way to get started if you're not familiar with Webpack, Babel, and Node.js.

Let's start by making a basic `index.html` file. We're going to load in three CDNs in the `head` - React, React DOM, and Babel. We're also going to make a `div` with an id called `root`, and finally we'll create a `script` tag where your custom code will live.

index.html

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="utf-8" />

    <title>Hello React!</title>
```



```
<script src="https://unpkg.com/babel-standalone@6.26.0/babel.js"></script>
</head>

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

  <script type="text/babel">
    // React code will go here
  </script>
</body>
</html>
```

I'm loading in the latest stable versions of the libraries as of the time of this writing.

- **React** - the React top level API
- **React DOM** - adds DOM-specific methods
- **Babel** - a JavaScript compiler that lets us use ES6+ in old browsers

The entry point for our app will be the `root` `div` element, which is named by convention. You'll also notice the `text/babel` script type, which is mandatory for using Babel.

Now, let's write our first code block of React. We're going to use ES6 classes to create a React component called `App`.

index.html

```
class App extends React.Component {
  //...
}
```

Now we'll add the `render()` method, the only required method in a class component, which is used to render DOM nodes.



```
class App extends React.Component {  
  render() {  
    return (  
      //...  
    );  
  }  
}
```

Inside the `return`, we're going to put what looks like a simple HTML element. Note that we're not returning a string here, so don't use quotes around the element. This is called `JSX`, and we'll learn more about it soon.

index.html

```
class App extends React.Component {  
  render() {  
    return <h1>Hello world!</h1>  
  }  
}
```

Finally, we're going to use the React DOM `render()` method to render the `App` class we created into the `root` div in our HTML.

index.html

```
ReactDOM.render(<App />, document.getElementById('root'))
```

Here is the full code for our `index.html`.

index.html

```
<!DOCTYPE html>  
<html>  
  <head>  
    <meta charset="utf-8" />
```



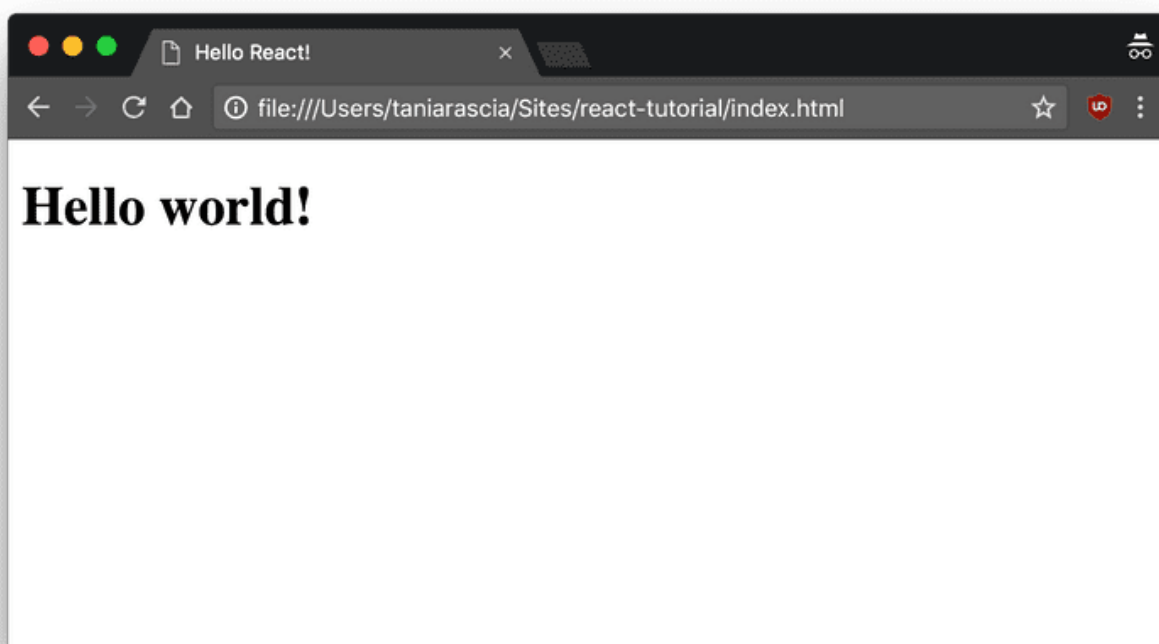
```
<script src="https://unpkg.com/react@16/umd/react.development.js"></scrip
<script src="https://unpkg.com/react-dom@16/umd/react-dom.development.js
<script src="https://unpkg.com/babel-standalone@6.26.0/babel.js"></scrip
</head>

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

  <script type="text/babel">
    class App extends React.Component {
      render() {
        return <h1>Hello world!</h1>
      }
    }

    ReactDOM.render(<App />, document.getElementById('root'))
  </script>
</body>
</html>
```

Now if you view your `index.html` in the browser, you'll see the `h1` tag we created rendered to the DOM.





Cool! Now that you've done this, you can see that React isn't so insanely scary to get started with. It's just some JavaScript helper libraries that we can load into our HTML.

We've done this for demonstration purposes, but from here out we're going to use another method: Create React App.

## Create React App

The method I just used of loading JavaScript libraries into a static HTML page and rendering the React and Babel on the fly is not very efficient, and is hard to maintain.

Fortunately, Facebook has created [Create React App](#), an environment that comes pre-configured with everything you need to build a React app. It will create a live development server, use Webpack to automatically compile React, JSX, and ES6, auto-prefix CSS files, and use ESLint to test and warn about mistakes in the code.

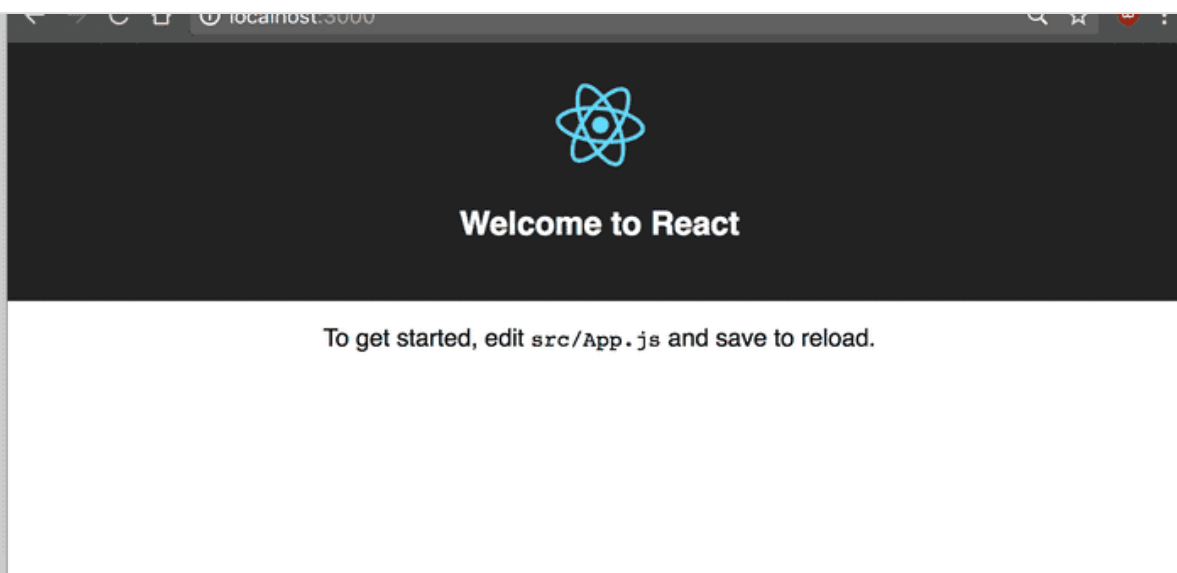
To set up `create-react-app`, run the following code in your terminal, one directory up from where you want the project to live.

```
$ npx create-react-app react-tutorial
```

Once that finishes installing, move to the newly created directory and start the project.

```
$ cd react-tutorial && npm start
```

Once you run this command, a new window will popup at `localhost:3000` with your new React app.



Create React App is very good for getting started for beginners as well as large-scale enterprise applications, but it's not perfect for every workflow. You can also create your own Webpack setup for React.

If you look into the project structure, you'll see a `/public` and `/src` directory, along with the regular `node_modules`, `.gitignore`, `README.md`, and `package.json`.

In `/public`, our important file is `index.html`, which is very similar to the static `index.html` file we made earlier - just a `root` `div`. This time, no libraries or scripts are being loaded in. The `/src` directory will contain all our React code.

To see how the environment automatically compiles and updates your React code, find the line that looks like this in `/src/App.js`:

To get started, edit ``src/App.js`` and save to reload.





Go ahead and delete all the files out of the `/src` directory, and we'll create our own boilerplate file without any bloat. We'll just keep `index.css` and `index.js`.

For `index.css`, I just copy-and-pasted the contents of [Primitive CSS](#) into the file. If you want, you can use Bootstrap or whatever CSS framework you want, or nothing at all. I just find it easier to work with.

Now in `index.js`, we're importing React, ReactDOM, and the CSS file.

src/index.js

```
import React from 'react'
import ReactDOM from 'react-dom'
import './index.css'
```

Let's create our `App` component again. Before, we just had an `<h1>`, but now I'm adding in a `div` element with a class as well. You'll notice that we use `className` instead of `class`. This is our first hint that the code being written here is JavaScript, and not actually HTML.

src/index.js

```
class App extends React.Component {
  render() {
    return (
      <div className="App">
        <h1>Hello, React!</h1>
      </div>
    )
  }
}
```

Finally, we'll render the `App` to the root as before.



```
ReactDOM.render(<App />, document.getElementById('root'))
```

Here's our full `index.js`. This time, we're loading the `Component` as a property of `React`, so we no longer need to extend `React.Component`.

src/index.js

```
import React, {Component} from 'react'
import ReactDOM from 'react-dom'

import './index.css'

class App extends Component {
  render() {
    return (
      <div className="App">
        <h1>Hello, React!</h1>
      </div>
    )
  }
}

ReactDOM.render(<App />, document.getElementById('root'))
```

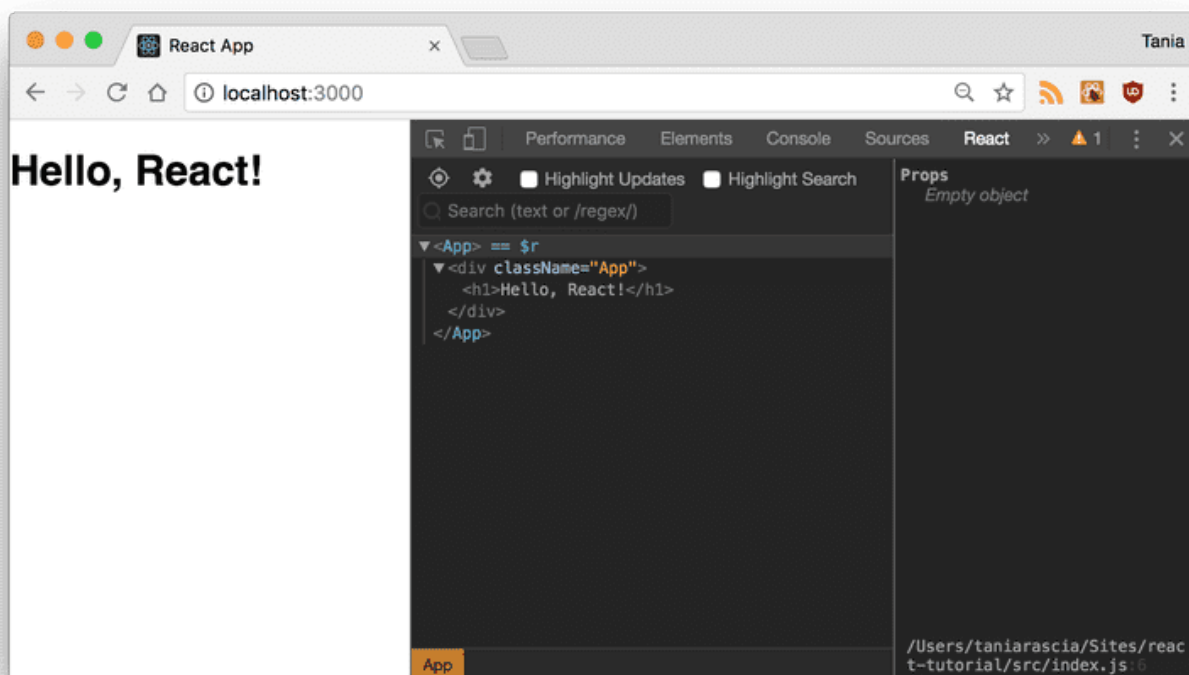
If you go back to `localhost:3000`, you'll see "Hello, React!" just like before. We have the beginnings of a React app now.

## React Developer Tools

There is an extension called React Developer Tools that will make your life much easier when working with React. Download [React DevTools for Chrome](#), or whatever browser you prefer to work on.



Elements tab to see the actual DOM output. It may not seem like that much of a deal now, but as the app gets more complicated, it will become increasingly necessary to use.



Now we have all the tools and setup we need to actually begin working with React.

## JSX: JavaScript + XML

As you've seen, we've been using what looks like HTML in our React code, but it's not quite HTML. This is **JSX**, which stands for JavaScript XML.

With JSX, we can write what looks like HTML, and also we can create and use our own XML-like tags. Here's what JSX looks like assigned to a variable.

JSX



Using JSX is not mandatory for writing React. Under the hood, it's running `createElement`, which takes the tag, object containing the properties, and children of the component and renders the same information. The below code will have the same output as the JSX above.

No JSX

```
const heading = React.createElement('h1', {className: 'site-heading'}, 'Hello
```

JSX is actually closer to JavaScript, not HTML, so there are a few key differences to note when writing it.

- `className` is used instead of `class` for adding CSS classes, as `class` is a reserved keyword in JavaScript.
- Properties and methods in JSX are camelCase - `onclick` will become `onClick`.
- Self-closing tags *must* end in a slash - e.g. `<img />`

JavaScript expressions can also be embedded inside JSX using curly braces, including variables, functions, and properties.

```
const name = 'Tania'  
const heading = <h1>Hello, {name}</h1>
```

JSX is easier to write and understand than creating and appending many elements in vanilla JavaScript, and is one of the reasons people love React so much.

## Components



## components.

Most React apps have many small components, and everything loads into the main `App` component. Components also often get their own file, so let's change up our project to do so.

Remove the `App` class from `index.js`, so it looks like this.

src/index.js

```
import React from 'react'
import ReactDOM from 'react-dom'
import App from './App'
import './index.css'

ReactDOM.render(<App />, document.getElementById('root'))
```

We'll create a new file called `App.js` and put the component in there.

src/App.js

```
import React, {Component} from 'react'

class App extends Component {
  render() {
    return (
      <div className="App">
        <h1>Hello, React!</h1>
      </div>
    )
  }
}

export default App
```



of-hand if you don't.

## Class Components

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

src/Table.js

```
import React, {Component} from 'react'

class Table extends Component {
  render() {
    return (
      <table>
        <thead>
          <tr>
            <th>Name</th>
            <th>Job</th>
          </tr>
        </thead>
        <tbody>
          <tr>
            <td>Charlie</td>
            <td>Janitor</td>
          </tr>
          <tr>
            <td>Mac</td>
            <td>Bouncer</td>
          </tr>
          <tr>
            <td>Dee</td>
            <td>Aspiring actress</td>
          </tr>
          <tr>
            <td>Dennis</td>
            <td>Bartender</td>
          </tr>
        </tbody>
      </table>
    )
  }
}
```



```
}  
}  
  
export default Table
```

This component we created is a custom class component. We capitalize custom components to differentiate them from regular HTML elements. Back in `App.js`, we can load in the `Table`, first by importing it in:

src/App.js

```
import Table from './Table'
```

Then by loading it into the `render()` of `App`, where before we had "Hello, React!". I also changed the class of the outer container.

src/App.js

```
import React, {Component} from 'react'  
import Table from './Table'  
  
class App extends Component {  
  render() {  
    return (  
      <div className="container">  
        <Table />  
      </div>  
    )  
  }  
}  
  
export default App
```

If you check back on your live environment, you'll see the `Table` loaded in.



Name	Job
Charlie	Janitor
Mac	Bouncer
Dee	Aspiring actress
Dennis	Bartender

Now we've seen what a custom class component is. We could reuse this component over and over. However, since the data is hard-coded into it, it wouldn't be too useful at the moment.

## 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.

src/Table.js

```
const TableHeader = () => {  
  return (  
    <thead>  
      <tr>  
        <th>Name</th>  
        <th>Job</th>  
      </tr>  
    </thead>  
  )  
}
```





```
}
```

Then the body.

src/Table.js

```
const TableBody = () => {  
  return (  
    <tbody>  
      <tr>  
        <td>Charlie</td>  
        <td>Janitor</td>  
      </tr>  
      <tr>  
        <td>Mac</td>  
        <td>Bouncer</td>  
      </tr>  
      <tr>  
        <td>Dee</td>  
        <td>Aspiring actress</td>  
      </tr>  
      <tr>  
        <td>Dennis</td>  
        <td>Bartender</td>  
      </tr>  
    </tbody>  
  )  
}
```

Now our `Table` file will look like this. Note that the `TableHeader` and `TableBody` components are all in the same file, and being used by the `Table` class component.

src/Table.js

```
const TableHeader = () => { ... }  
const TableBody = () => { ... }
```



```
    return (  
      <table>  
        <TableHeader />  
        <TableBody />  
      </table>  
    )  
  }  
}
```

Everything should appear as it did before. As you can see, components can be nested in other components, and simple and class components can be mixed.

A class component must include `render()`, and the `return` can only return one parent element.

As a wrap up, let's compare a simple component with a class component.

#### Simple Component

```
const SimpleComponent = () => {  
  return <div>Example</div>  
}
```

#### Class Component

```
class ClassComponent extends Component {  
  render() {  
    return <div>Example</div>  
  }  
}
```

Note that if the `return` is contained to one line, it does not need parentheses.



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.

First, let's remove all the data from our `TableBody` component.

src/Table.js

```
const TableBody = () => {  
  return <tbody />  
}
```

Then let's move all that data to an array of objects, as if we were bringing in a JSON-based API. We'll have to create this array inside our `render()`.

src/App.js

```
class App extends Component {  
  render() {  
    const characters = [  
      {  
        name: 'Charlie',  
        job: 'Janitor',  
      },  
      {  
        name: 'Mac',  
        job: 'Bouncer',  
      },  
      {  
        name: 'Dee',  
        job: 'Aspring actress',  
      },  
      {  
        name: 'Dennis',  
        job: 'Bartender',  
      },  
    ],
```



```
        <div className="container">
          <Table />
        </div>
      )
    }
  }
}
```

Now, we're going to pass the data through to the child component ( `Table` ) with properties, kind of how you might pass data through using `data-` attributes. We can call the property whatever we want, as long as it's not a reserved keyword, so I'll go with `characterData`. The data I'm passing through is the `characters` variable, and I'll put curly braces around it as it's a JavaScript expression.

src/App.js

```
return (
  <div className="container">
    <Table characterData={characters} />
  </div>
)
```

Now that data is being passed through to `Table`, we have to work on accessing it from the other side.

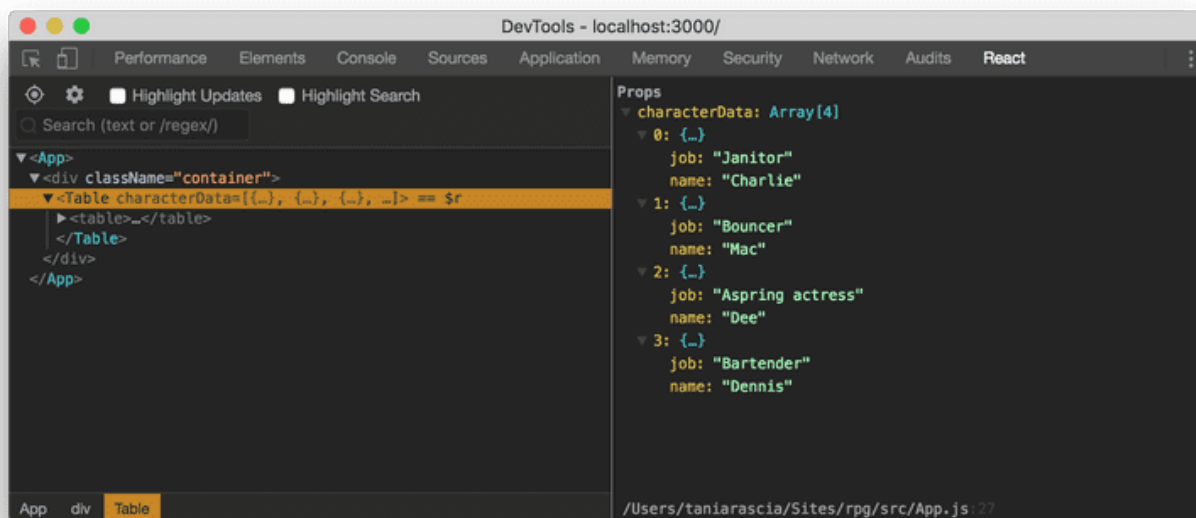
src/Table.js

```
class Table extends Component {
  render() {
    const {characterData} = this.props

    return (
      <table>
        <TableHeader />
        <TableBody characterData={characterData} />
      </table>
    )
  }
}
```



If you open up React DevTools and inspect the `Table` component, you'll see the array of data in the property. The data that's stored here is known as the **virtual DOM**, which is a fast and efficient way of syncing data with the actual DOM.



This data is not in the actual DOM yet, though. In `Table`, we can access all props through `this.props`. We're only passing one props through, `characterData`, so we'll use `this.props.characterData` to retrieve that data.

I'm going to use the ES6 property shorthand to create a variable that contains `this.props.characterData`.

```
const {characterData} = this.props
```

Since our `Table` component actually consists of two smaller simple components, I'm going to pass it through to the `TableBody`, once again through props.



```
render() {  
  const {characterData} = this.props  
  
  return (  
    <table>  
      <TableHeader />  
      <TableBody characterData={characterData} />  
    </table>  
  )  
}
```

Right now, `TableBody` takes no parameters and returns a single tag.

src/Table.js

```
const TableBody = () => {  
  return <tbody />  
}
```

We're going to pass the props through as a parameter, and [map through the array](#) to return a table row for each object in the array. This map will be contained in the `rows` variable, which we'll return as an expression.

src/Table.js

```
const TableBody = (props) => {  
  const rows = props.characterData.map((row, index) => {  
    return (  
      <tr key={index}>  
        <td>{row.name}</td>  
        <td>{row.job}</td>  
      </tr>  
    )  
  })  
  
  return <tbody> {rows} </tbody>
```



If you view the front end of the app, all the data is loading in now.

You'll notice I've added a key index to each table row. You should always use keys when making lists in React, as they help identify each list item. We'll also see how this is necessary in a moment when we want to manipulate list items.

Props are an effective way to pass existing data to a React component, however the component cannot change the props - they're read-only. In the next section, we'll learn how to use state to have further control over handling data in React.

## State

Right now, we're storing our character data in an array in a variable, and passing it through as props. This is good to start, but imagine if we want to be able to delete an item from the array. With props, we have a one way data flow, but with state we can update private data from a component.

You can think of state as any data that should be saved and modified without necessarily being added to a database - for example, adding and removing items from a shopping cart before confirming your purchase.

To start, we're going to create a `state` object.

src/App.js

```
class App extends Component {  
  state = {}  
}
```

The object will contain properties for everything you want to store in the state. For us, it's `characters`.



```
class App extends Component {  
  state = {  
    characters: [],  
  }  
}
```

Move the entire array of objects we created earlier into `state.characters`.

src/App.js

```
class App extends Component {  
  state = {  
    characters: [  
      {  
        name: 'Charlie',  
        // the rest of the data  
      },  
    ],  
  }  
}
```

Our data is officially contained in the state. Since we want to be able to remove a character from the table, we're going to create a `removeCharacter` method on the parent `App` class.

To retrieve the state, we'll get `this.state.characters` using the same ES6 method as before. To update the state, we'll use `this.setState()`, a built-in method for manipulating state. We'll filter the array based on an `index` that we pass through, and return the new array.

You must use `this.setState()` to modify an array. Simply applying a new value to `this.state.property` will not work.





```
const {characters} = this.state

this.setState({
  characters: characters.filter((character, i) => {
    return i !== index
  }),
})
}
```

`filter` does not mutate but rather creates a new array, and is a preferred method for modifying arrays in JavaScript. This particular method is testing an index vs. all the indices in the array, and returning all but the one that is passed through.

Now we have to pass that function through to the component, and render a button next to each character that can invoke the function. We'll pass the `removeCharacter` function through as a prop to `Table`.

src/App.js

```
render() {
  const { characters } = this.state

  return (
    <div className="container">
      <Table characterData={characters} removeCharacter={this.removeCharacte}
    </div>
  )
}
```

Since we're passing it down to `TableBody` from `Table`, we're going to have to pass it through again as a prop, just like we did with the character data.

In addition, since it turns out that the only components having their own states in our project are `App` and `Form`, it would be best practice to transform `Table` into a simple



src/Table.js

```
const Table = (props) => {  
  const {characterData, removeCharacter} = props  
  
  return (  
    <table>  
      <TableHeader />  
      <TableBody characterData={characterData} removeCharacter={removeCharacter} />  
    </table>  
  )  
}
```

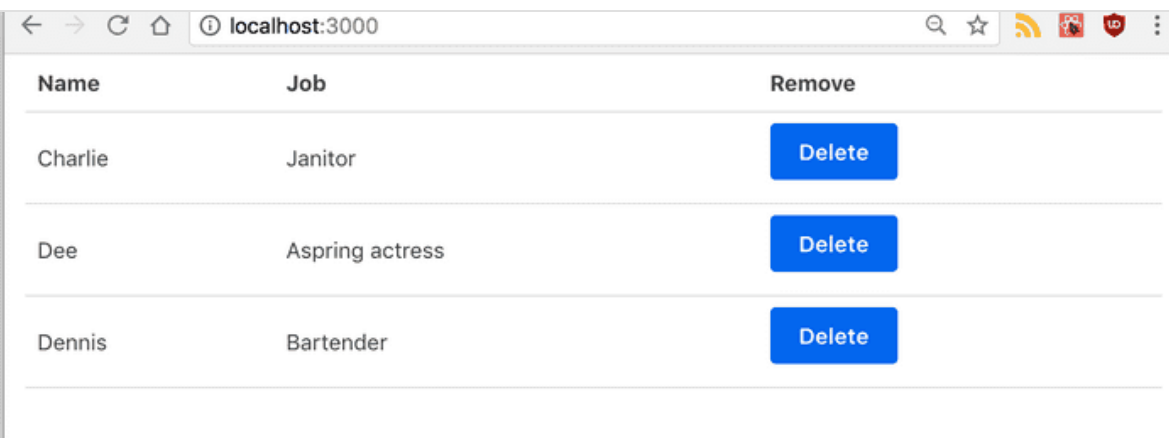
Here's where that index we defined in the `removeCharacter()` method comes in. In the `TableBody` component, we'll pass the key/index through as a parameter, so the filter function knows which item to remove. We'll create a button with an `onClick` and pass it through.

src/Table.js

```
<tr key={index}>  
  <td>{row.name}</td>  
  <td>{row.job}</td>  
  <td>  
    <button onClick={() => props.removeCharacter(index)}>Delete</button>  
  </td>  
</tr>
```

The `onClick` function must pass through a function that returns the `removeCharacter()` method, otherwise it will try to run automatically.

Awesome. Now we have delete buttons, and we can modify our state by deleting a character.



The screenshot shows a web browser window at localhost:3000. It displays a table with three columns: Name, Job, and Remove. The table contains three rows of data, each with a blue 'Delete' button in the Remove column.

Name	Job	Remove
Charlie	Janitor	Delete
Dee	Aspring actress	Delete
Dennis	Bartender	Delete

I deleted Mac.

Now you should understand how state gets initialized and how it can be modified.

## 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.characters`, as we'll be updating that through the form now.

src/App.js

```
class App extends Component {  
  state = {  
    characters: [],  
  }  
}
```



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`.

src/Form.js

```
import React, {Component} from 'react'

class Form extends Component {
  initialState = {
    name: '',
    job: '',
  }

  state = this.initialState
}
```

Previously, it was necessary to include a `constructor()` on React class components, but it's not required anymore.

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.

src/Form.js

```
handleChange = (event) => {
  const {name, value} = event.target

  this.setState({
```



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.

src/Form.js

```
render() {  
  const { name, job } = this.state;  
  
  return (  
    <form>  
      <label htmlFor="name">Name</label>  
      <input  
        type="text"  
        name="name"  
        id="name"  
        value={name}  
        onChange={this.handleChange} />  
      <label htmlFor="job">Job</label>  
      <input  
        type="text"  
        name="job"  
        id="job"  
        value={job}  
        onChange={this.handleChange} />  
    </form>  
  );  
}  
  
export default Form;
```

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

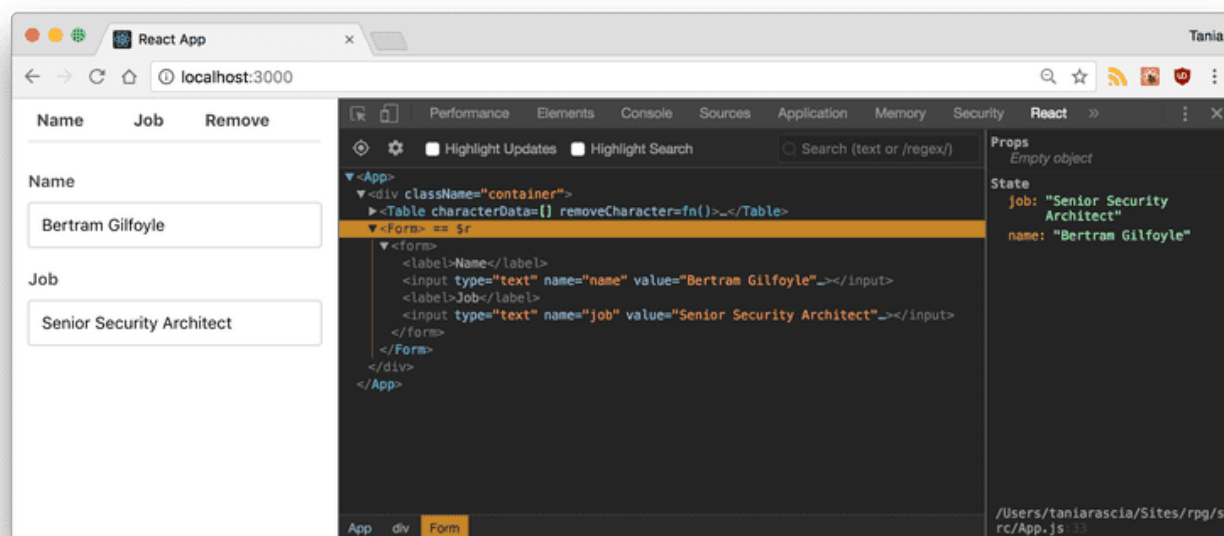
src/App.js



src/App.js

```
return (  
  <div className="container">  
    <Table characterData={characters} removeCharacter={this.removeCharacter}>  
      <Form />  
    </div>  
  )
```

Now if we go to the front end of our app, we'll see a form that doesn't have a submit yet. Update some fields and you'll see the local state of Form being updated.



Cool. 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.characters` and adding the new `character` parameter, using the [ES6 spread operator](#).

src/App.js

```
handleSubmit = (character) => {
```



Let's make sure we pass that through as a parameter on `Form`.

```
<Form handleSubmit={this.handleSubmit} />
```

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

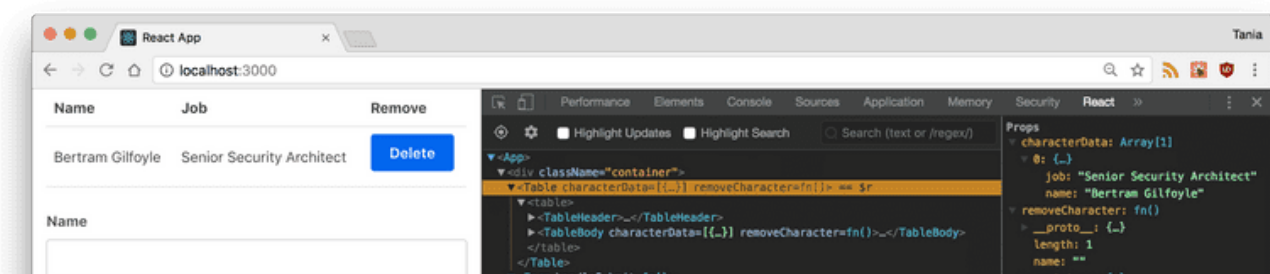
src/Form.js

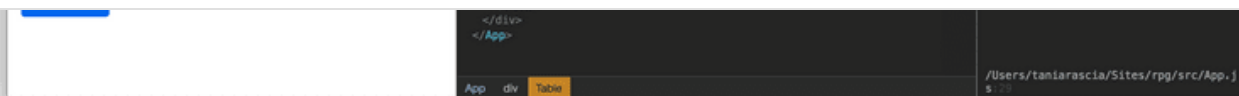
```
submitForm = () => {  
  this.props.handleSubmit(this.state)  
  this.setState(this.initialState)  
}
```

Finally, we'll add a submit button to submit the form. We're using an `onClick` instead of an `onSubmit` since we're not using the standard submit functionality. The click will call the `submitForm` we just made.

```
<input type="button" value="Submit" onClick={this.submitForm} />
```

And that's it! The app is complete. We can create, add, and remove users from our table. Since the `Table` and `TableBody` were already pulling from the state, it will display properly.





If you got lost anywhere along the way, you can view [the complete source on GitHub](#).

## Pulling in API Data

One very common usage of React is pulling in data from an API. If you're not familiar with what an API is or how to connect to one, I would recommend reading [How to Connect to an API with JavaScript](#), which will walk you through what APIs are and how to use them with vanilla JavaScript.

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](#), and I have a [URL endpoint right here](#) for a random\* search. You can go to that link to see the API - and make sure you have [JSONView](#) installed on your browser.

We're going to use [JavaScript's built-in Fetch](#) to gather the data from that URL endpoint and display it. You can switch between the app we created and this test file by just changing the URL in `index.js` - `import App from './Api';`.

I'm not going to explain this code line-by-line, as we've already learned about creating a component, rendering, and mapping through a state array. The new aspect to this code is `componentDidMount()`, a React lifecycle method. **Lifecycle** is the order in which methods are called in React. **Mounting** refers to an item being inserted into the DOM.

When we pull in API data, we want to use `componentDidMount`, because we want to make sure the component has rendered to the DOM before we bring in the data. In





Api.js

```
import React, {Component} from 'react'

class App extends Component {
  state = {
    data: [],
  }

  // Code is invoked after the component is mounted/inserted into the DOM tree
  componentDidMount() {
    const url =
      'https://en.wikipedia.org/w/api.php?action=opensearch&search=Seona+Danc

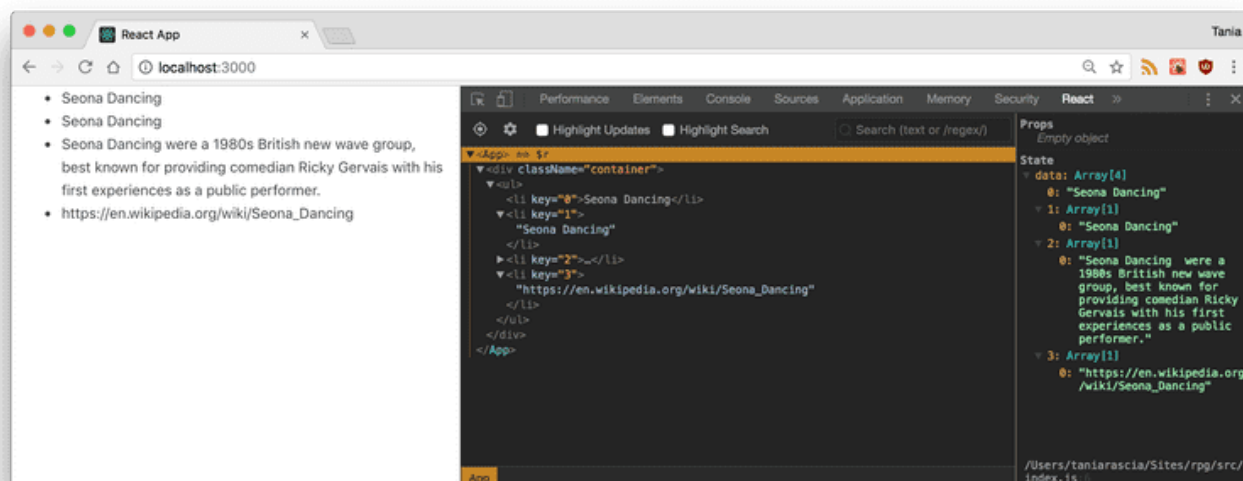
    fetch(url)
      .then((result) => result.json())
      .then((result) => {
        this.setState({
          data: result,
        })
      })
  }

  render() {
    const {data} = this.state

    const result = data.map((entry, index) => {
      return <li key={index}>{entry}</li>
    })

    return <ul>{result}</ul>
  }
}

export default App
```



There are other lifecycle methods, but going over them will be beyond the scope of this article. You can [read more about React components here](#).

*\*Wikipedia search choice may not be random. It might be an article that I spearheaded back in 2005.*

## Building and Deploying a React App

Everything we've done so far has been in a development environment. We've been compiling, hot-reloading, and updating on the fly. For production, we're going to want to have static files loading in - none of the source code. We can do this by making a build and deploying it.

Now, if you just want to compile all the React code and place it in the root of a directory somewhere, all you need to do is run the following line:

```
$ npm run build
```



We can also take it a step further, and have npm deploy for us. We're going to build to GitHub pages, so you'll already have to [be familiar with Git](#) and getting your code up on GitHub.

Make sure you've exited out of your local React environment, so the code isn't currently running. First, we're going to add a `homepage` field to `package.json`, that has the URL we want our app to live on.

```
package.json
```

```
"homepage": "https://taniarascia.github.io/react-tutorial",
```

We'll also add these two lines to the `scripts` property.

```
"scripts": {  
  // ...  
  "predeploy": "npm run build",  
  "deploy": "gh-pages -d build"  
}
```

In your project, you'll add `gh-pages` to the `devDependencies`.

```
$ npm install --save-dev gh-pages
```

We'll create the `build`, which will have all the compiled, static files.

```
$ npm run build
```

Finally, we'll deploy to `gh-pages`.



And we're done! The app is now available live at <https://taniarascia.github.io/react-tutorial>.

## Conclusion

This article should have given you a good introduction to React, simple and class components, state, props, working with form data, pulling data in from an API, and deploying an app. There is much more to learn and do with React, but I hope you feel confident delving in and playing around with React yourself now.

- [View Source on GitHub](#)
- [View Project](#)

Please let me know if anything was unclear, or if there's anything else you'd like to see in this or a subsequent article.

## Comments

95 Comments - powered by [utteranc.es](#)

Chyma commented on 5 thg 8, 2020

Hello Tania...

Awesome tutorial,

I have a question though, please can you shed more light on this particular line in your code, why did you use three periods (...) before 'this' in the function below. Thanks

```
handleSubmit = (character) => {
```



1

dinesh-maripi commented on 6 thg 8, 2020

... is javaScript **spread** operator.

I will try to explain the **spread** operator with the following example.

```
const citrusFruits = ['orange', 'lime', 'mandarin']
const tropicalFruits = ['mango', 'banana']
const allFruits = [...citrusFruits, ...tropicalFruits] // using ... operator to copy citrusFruits
console.log(allFruits) // o/p: ['orange', 'lime', 'mandarin', 'mango', 'banana']
```



10



4

Chyma commented on 6 thg 8, 2020

Thank you very much.

...

alemRangel commented on 12 thg 8, 2020

Thanks for sharing your knowledge, you are amazing.

ekta96 commented on 23 thg 8, 2020

Thanks Tania for using the simplest way to explain ReactJs .

hidelryn commented on 25 thg 8, 2020

thx! good post

khalilhimura commented on 30 thg 8, 2020

Thank you for this. A good & quick refresher.



haomingxiao commented on 1 thg 9, 2020

Thanks for the tutorial. One question for the code below, why there is [ ] around `name` variable. If I remove the bracket, vscode tells me `name` extracted from `event.target` is not used, and the form is not working as designed. What does this [ ] mean?

```
handleChange = (event) => {  
  const {name, value} = event.target  
  
  this.setState({  
    [name]: value,  
  })  
}
```

HengJunXi commented on 2 thg 9, 2020

@haomingxiao [ ] around `name` variable is an ES6 feature, called the computed property name. You may refer to <https://ui.dev/computed-property-names/>



4

cgossain commented on 3 thg 9, 2020

Seriously awesome tutorial! I had react the react docs a few months ago, but this is so much more concise and feels like you're getting the same information. Thanks!

dylan-ztm commented on 9 thg 9, 2020

Hi Tania,

This was a super awesome tutorial! I built out this app a few days ago and then used as a basis for an inventory management app (frontend only) using the same structure. This hands on lesson was well written and organized and helped me understand some key React concepts like state and event handling. Thank you so much for making this available!

haroon-dev commented on 9 thg 9, 2020

Thanks Tania for explaining React so simple.



Great Blog for React. Nice to learn for beginners.

I learnt from this blog for the first time as a beginners and now am developing many react application.  
Thanks for this blog.

**alexleduc76** commented on 13 thg 9, 2020

Thanks for writing this article. It was my first introduction to React and learning it has sparked a tidal wave of interest the npm/JS ecosystem around it. I really needed something to kick me out of my complacency, i.e. the attitude that jQuery loaded from a script tag was good enough for anything I wanted to do on the client side.

**namsefroehlich** commented on 17 thg 9, 2020

Thank you :D

**zodocaring** commented on 17 thg 9, 2020

Hi,

I am new one to react, I just follow the tutorial. when I execute the command:  
`npx create-react-app react-tutorial`

the console the windows 10 show me an error:

npm ERR! code ENOLOCAL

npm ERR! Could not install from "Education\AppData\Roaming\npm-cache\_npx\3932" as it does not contain a package.json file.

npm ERR! A complete log of this run can be found in:

npm ERR! C:\Users\Studo Education\AppData\Roaming\npm-cache\_logs\2020-09-17T00\_57\_06\_469Z-debug.log

Install for [ 'create-react-app@latest' ] failed with code 1

I do not know how to resolve the problem!!

I will be grateful for the help.

**HerllonCardoso** commented on 17 thg 9, 2020

Thank you for making so clear!



**spconway** commented on 23 thg 9, 2020

Thanks for this! It's to learn about the basics before jumping in to the more advanced stuff.

**taniarascia** commented on 24 thg 9, 2020

Owner

[@zodocaring](#) it's `create-react-app` not `greate-react-app` , I assume that's why.

**IkramMaududi** commented on 4 thg 10, 2020

Hello Tania [@taniarascia](#) and whoever reads this, help me please, i've some questions

1. Do we still need to use constructor method in the new react? here Tania doesn't use constructor, but in her github source there is the constructor in form.js
2. What's the difference between using arrow function and method inside class? for things like `handleChange` and `handleSubmit`, Tania uses arrow function, but for `render` she uses method

Thank You.....



2

**boldrack** commented on 12 thg 10, 2020

[@IkramMaududi](#)

1. Now we don't need constructor any longer, WE DON'T
2. arrow function helps you automatically bind to the component class so you don't need to do something like this  
`handleChange = this.handleChange.bind(this);`  
binding the method to the component, arrow functions do this automatically.

**el-tio-victor** commented on 22 thg 10, 2020

Wow!! great tutorial.

I am learning react and this help me to understand more it.

Thanks for your time [@taniarascia](#)

Saludos!!





Thanks Taniarascia.

**cjdavies** commented on 22 thg 11, 2020

I thouhgt that you might find this intersting: <https://github.com/JennaSys/tictacreact>

**lorddexon** commented on 22 thg 11, 2020

Nice guide to get a quick overview, really useful, thanks

50 hidden items

[Load more...](#)

**thenewforktimes** commented on 8 thg 7, 2021

Hello React Friends <3

I've been stuck on refactoring this tutorial without Class based components, and with the React Hooks that have been released since this went live. When I fire off my delete button to remove my least favorite character from the show, I don't get any compilation errors, but I get this – Error: Invalid hook call. Hooks can only be called inside of the body of a function component. I'll CC my app.js and table.js files below, thank you for any insight and learnings that will help me get unstuck.

```
import React, { useState } from "react";
import Table from "./Table";
```

```
const App = () => {
  const characters = [
    {
      name: "Charlie",
      job: "Janitor",
    },
    {
      name: "Mac",
      job: "Bouncer",
    },
  ],
```

**renosman33** commented on 8 thg 7, 2021

**Tania Rascia**

Articles



Projects



About me



not superfluous since the predeploy hook already runs `npm run build` ?

**MestreALMO** commented on 21 thg 7, 2021

This is a pretty good tutorial! good job and thank you for it! 🍷

**Tran-Van-Bong-95** commented on 26 thg 7, 2021

how to know App.js and Form.js have their own states ? because I still don't understand "since it turns out that the only components having their own states in our project are App and Form," in the article ?



1

**NgoranA** commented on 6 thg 9, 2021

@ thenewforktimes

Remove your `useState()` hook from the delete function! It should be placed right in the App component

**youssef-aitali** commented on 16 thg 9, 2021

Great Overview for React foundations! Thanks Tania.

**maendoza** commented on 27 thg 9, 2021

Thank you, very nice tutorial

**videvelop** commented on 6 thg 10, 2021

Thank you! you saved lot of time for me!

**aarontgrogg** commented on 8 thg 10, 2021

Hey there, truly impressive!



Solved by passing the `event` into the `submitForm` method, then using the classic `event.preventDefault()`; . Is there a better version of this?

Thanks again!



1

**KutanaDev** commented on 12 thg 10, 2021

Thank you for creating this tutorial - I think it is really excellent.  
There is a tiny typo: **Aspring** actress

**Gozde027** commented on 24 thg 10, 2021

Great tutorial! Thanks

**tractortoby** commented on 1 thg 11, 2021

Thanks Tania. If you get an error message like me after `npm run deploy`: Failed to get remote.origin.url (task must either be run in a git repository with a configured origin remote or must be configured with the "repo" option), you probably forget `git remote add origin https://github.com/yourname/project` and `git push --set-upstream origin master` after creating new repository.

**Kaunaj** commented on 14 thg 11, 2021

Thanks for the easy-to-follow tutorial with clear explanations. Really good for people at all levels looking to get their feet wet in React.

I followed along and created my own version of the app, adding very basic checks and edit functionality. For anyone interested, here is the demo: <https://kaunaj.github.io/first-react-app/>

**t4selva** commented 3 months ago

Today, Many React js developers are not using Class based components, including me, they are using Functional based components. on the other side, React js is a fully js library. Its very hard to scale up big projects. So NEXT.JS is very popular now. I am also using next js ( React Framework ).



... is javascript **spread** operator. I will try to explain the **spread** operator with the following example.

```
const citrusFruits = ['orange', 'lime', 'mandarin']
const tropicalFruits = ['mango', 'banana']
const allFruits = [...citrusFruits, ...tropicalFruits] // using ... operator to copy
console.log(allFruits) // o/p: ['orange', 'lime', 'mandarin', 'mango', 'banana']
```

Why do I get a "Parsing error: Identifier 'TableHeader' has already been declared in the javascript spread object. I am new to react

**kfusilier** commented 3 weeks ago

About halfway through you stop showing what the whole page should look like so the content of the changes is lost on a newbie. Can't get past the adding delete button section, it's unclear what needs to stay and go. I've tried it 5 different ways and they all show errors. Update this tutorial if it's going to be linked on the react docs please!

**thaitina5** commented 3 weeks ago

Hi Tania, i'm finished at "Submitting Form Data" step but after i pressed Submit button nothing appear in Table. I did checked again handleChange() method, it still works. Any ideas why i fail? Thanks

**julesrufo26** commented 3 weeks ago

Thank you Tania :)

**rhammell** commented a week ago

Thanks for writing this, its very helpful!

**manuellaines** commented 3 days ago

hello Tania  
thank you for your explanations they are very interesting you really touch the basics and suddenly it helps us a lot we beginners

**Tania Rascia**

Articles



Projects



About me



Write



Preview

Sign in to comment

 Styling with Markdown is supported

Sign in with GitHub

Made by Tania Rascia   Newsletter   Ko-Fi   Patreon   RSS

Gatsby GitHub Netlify 