NOVEMBER 20, 2019 / #REACT

The Best React Examples

React (also known as React.js) is one of the most popular JavaScript front end development libraries. Here is a collection of React syntax and usage that you can use as a handy guide or reference.

React Component Example

Components are reusable in React.js. You can inject value into props as given below:

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

const element = <Welcome name="Faisal Arkan" />;
ReactDOM.render(
  element,
  document.getElementById('root')
);
```

`name="Faisal Arkan"` will give value into `{props.name}` from `function Welcome(props)` and returning a component that has given

Other ways to declare components

There are many ways to declare components when using React.js. There are two kinds of components, ***stateless*** components and ***stateful*** components.

Stateful

Class Type Components

```
class Cat extends React.Component {
  constructor(props) {
    super(props);

    this.state = {
      humor: 'happy'
    }
  }
  render() {
    return(
      <div>
        <h1>{this.props.name}</h1>
        <p>
          {this.props.color}
        </p>
      </div>
    );
  }
}
```

Stateless Components

Functional Components (Arrow Function from ES6)

```
      <h1>{props.name}</h1>
      <p>{props.color}</p>
    </div>;
  );
};
```

Implicit Return Components

```
const Cat = props =>
  <div>
    <h1>{props.name}</h1>
    <p>{props.color}</p>
  </div>;
```

Fragments are way to render multiple elements without using a wrapper element. When attempting to render elements without an enclosing tag in JSX, you will see the error message `Adjacent JSX elements must be wrapped in an enclosing tag`. This is because when JSX transpiles, it's creating elements with their corresponding tag names, and doesn't know what tag name to use if multiple elements are found.

In the past, a frequent solution to this was to use a wrapper div to solve this problem. However, version 16 of React brought the addition of `Fragment`, which makes this no longer necessary.

`Fragment` acts a wrapper without adding unnecessary divs to the DOM. You can use it directly from the React import, or deconstruct it:

```
  render(){
    return (
      <React.Fragment>
        <div>I am an element!</div>
        <button>I am another element</button>
      </React.Fragment>
    );
  }
}

export default MyComponent;
```

```
// Deconstructed
import React, { Component, Fragment } from 'react';

class MyComponent extends Component {
  render(){
    return (
      <Fragment>
        <div>I am an element!</div>
        <button>I am another element</button>
      </Fragment>
    );
  }
}

export default MyComponent;
```

React version 16.2 simplified this process further, allowing for empty JSX tags to be interpreted as Fragments:

```
return (
  <>
```

JSX

JSX is short for JavaScript XML.

JSX is an expression which uses valid HTML statements within JavaScript. You can assign this expression to a variable and use it elsewhere. You can combine other valid JavaScript expressions and JSX within these HTML statements by placing them within braces (`{}`). Babel further compiles JSX into an object of type `React.createElement()` .

Single-line & Multi-line expressions

Single-line expression are simple to use.

```
const one = <h1>Hello World!</h1>;
```

When you need to use multiple lines in a single JSX expression, write the code within a single parenthesis.

```
const two = (
  <ul>
    <li>Once</li>
    <li>Twice</li>
  </ul>
);
```

```
const greet = <h1>Hello World!</h1>;
```

Combining JavaScript expression with HTML tags

We can use JavaScript variables within braces.

```
const who = "Quincy Larson";
const greet = <h1>Hello {who}!</h1>;
```

We can also call other JavaScript functions within braces.

```
function who() {
  return "World";
}
const greet = <h1>Hello {who()}!</h1>;
```

Only a single parent tag is allowed

A JSX expression must have only one parent tag. We can add multiple tags nested within the parent element only.

```
// This is valid.
const tags = (
  <ul>
```

```
// This is not valid.
const tags = (
  <h1>Hello World!</h1>
  <h3>This is my special list:</h3>
  <ul>
    <li>Once</li>
    <li>Twice</li>
  </ul>
);
```

React State Example

State is the place where the data comes from.

We should always try to make our state as simple as possible and minimize the number of stateful components. If we have, for example, ten components that need data from the state, we should create one container component that will keep the state for all of them.

State is basically like a global object that is available everywhere in a component.

Example of a Stateful Class Component:

```
import React from 'react';

class App extends React.Component {
  constructor(props) {
    super(props);

    // We declare the state as shown below
```

```
    }
  }
  render() {
    return (
      <div>
        <h1>{this.state.x}</h1>
        <h2>{this.state.y}</h2>
      </div>
    );
  }
}
export default App;
```

Another Example:

```
import React from 'react';

class App extends React.Component {
  constructor(props) {
    super(props);

    // We declare the state as shown below
    this.state = {
      x: "This is x from state",
      y: "This is y from state"
    }
  }

  render() {
    let x1 = this.state.x;
    let y1 = this.state.y;

    return (
      <div>
        <h1>{x1}</h1>
        <h2>{y1}</h2>
      </div>
```

Updating State

You can change the data stored in the state of your application using the `setState` method on your component.

```
this.setState({ value: 1 });
```

Keep in mind that `setState` is asynchronous so you should be careful when using the current state to set a new state. A good example of this would be if you want to increment a value in your state.

The Wrong Way

```
this.setState({ value: this.state.value + 1 });
```

This can lead to unexpected behavior in your app if the code above is called multiple times in the same update cycle. To avoid this you can pass an updater callback function to `setState` instead of an object.

The Right Way

```
this.setState(prevState => ({ value: prevState.value + 1 }));
```

You can change the data stored in the state of your application using the `setState` method on your component.

```
this.setState({value: 1});
```

Keep in mind that `setState` may be asynchronous so you should be careful when using the current state to set a new state. A good example of this would be if you want to increment a value in your state.

The Wrong Way

```
this.setState({value: this.state.value + 1});
```

This can lead to unexpected behavior in your app if the code above is called multiple times in the same update cycle. To avoid this you can pass an updater callback function to `setState` instead of an object.

The Right Way

```
this.setState(prevState => ({value: prevState.value + 1}));
```

The Cleaner Way

When only a limited number of fields in the state object is required, object destructing can be used for cleaner code.

React State VS Props Example

When we start working with React components, we frequently hear two terms. They are `state` and `props`. So, in this article we will explore what are those and how they differ.

State:

- State is something that a component owns. It belongs to that particular component where it is defined. For example, a person's age is a state of that person.
- State is mutable. But it can be changed only by that component that owns it. As I only can change my age, not anyone else.
- You can change a state by using `this.setState()`

See the below example to get an idea of state:

Person.js

```
import React from 'react';

class Person extends React.Component{
  constructor(props) {
    super(props);
    this.state = {
      age:0
    this.incrementAge = this.incrementAge.bind(this)
```

```
      age:this.state.age + 1;
    });
  }

  render(){
    return(
      <div>
        <label>My age is: {this.state.age}</label>
        <button onClick={this.incrementAge}>Grow me older !!<button>
      </div>
    );
  }
}

export default Person;
```

In the above example, `age` is the state of `Person` component.

Props:

- Props are similar to method arguments. They are passed to a component where that component is used.
- Props is immutable. They are read-only.

See the below example to get an idea of Props:

Person.js

```
import React from 'react';

class Person extends React.Component{
  render(){
```

```
    );
  }
}

export default Person;

const person = <Person character = "good"></Person>
```

In the above example, `const person = <Person character = "good"> </Person>` we are passing `character = "good"` prop to `Person` component.

It gives output as "I am a good person", in fact I am.

There is lot more to learn on State and Props. Many things can be learnt by actually diving into coding. So get your hands dirty by coding.

React Higher-Order Component Example

In React, a **Higher-Order Component** (HOC) is a function that takes a component and returns a new component. Programmers use HOCs to achieve **component logic reuse**.

If you've used Redux's `connect` , you've already worked with Higher-Order Components.

The core idea is:

Where:

- `enhance` is the Higher-Order Component;
- `WrappedComponent` is the component you want to enhance; and
- `EnhancedComponent` is the new component created.

This could be the body of the `enhance` HOC:

```
function enhance(WrappedComponent) {
  return class extends React.Component {
    render() {
      const extraProp = 'This is an injected prop!';
      return (
        <div className="Wrapper">
          <WrappedComponent
            {...this.props}
            extraProp={extraProp}
          />
        </div>
      );
    }
  }
}
```

In this case, `enhance` returns an **anonymous class** that extends `React.Component`. This new component is doing three simple things:

- Rendering the `WrappedComponent` within a `div` element;
- Passing its own props to the `WrappedComponent`; and
- Injecting an extra prop to the `WrappedComponent`.

things you can do with them!

If this article was helpful, tweet it.

Learn to code for free. freeCodeCamp's open source curriculum has helped more than 40,000 people get jobs as developers.

Get started

freeCodeCamp is a donor-supported tax-exempt 501(c)(3) nonprofit organization (United States Federal Tax Identification Number: 82-0779546)

Our mission: to help people learn to code for free. We accomplish this by creating thousands of videos, articles, and interactive coding lessons - all freely available to the public. We also have thousands of freeCodeCamp study groups around the world.

Donations to freeCodeCamp go toward our education initiatives, and help pay for servers,

Trending Guides

Learn JavaScript
Linux ln Example
JS document.ready()
Delete a Row in SQL
Python Round to Int
What is msmpeng.exe?
Queue Data Structure
Learn Web Development
Install Node on Windows
Remove Char from String
Open Task Manager on Mac
parseInt() in JavaScript
Print statement in Python
Remove Directory in Linux
Python str.lower() Example
Rust Lang
Python Sets
C++ Strings
Python map()
Python .pop()
Python arrays
npm Uninstall
Insertion Sort
Python If-Else
All Caps in CSS
Second Monitor Not Detected
How to Declare Strings in C
How to Use .len() in Python
Python Convert String to Int
How to create a free website

Our Nonprofit

About Alumni Network Open Source Shop Support Sponsors Academic Honesty
Code of Conduct Privacy Policy Terms of Service Copyright Policy