React Introduction

What is React?

React, sometimes referred to as a frontend JavaScript framework, is a JavaScript library created by Facebook.

React is a tool for building UI components.

How does React Work?

React creates a VIRTUAL DOM in memory.

Instead of manipulating the browser's DOM directly, React creates a virtual DOM in memory, where it does all the necessary manipulating, before making the changes in the browser DOM.

React only changes what needs to be changed!

React finds out what changes have been made, and changes **only** what needs to be changed.

You will learn the various aspects of how React does this in the rest of this tutorial.

React.JS History

Current version of React.JS is V18.0.0 (April 2022).

Initial Release to the Public (V0.3.0) was in July 2013.

React.JS was first used in 2011 for Facebook's Newsfeed feature.

Facebook Software Engineer, Jordan Walke, created it.

Current version of `create-react-app` is v5.0.1 (April 2022).

`create-react-app` includes built tools such as webpack, Babel, and ESLint.

•React Render HTML

React's goal is in many ways to render HTML in a web page.

React renders HTML to the web page by using a function called `createRoot()` and its method `render()`.

The createRoot Function

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

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

The render Method

The `render()` method is then called to define the React component that should be rendered.

But render where?

There is another folder in the root directory of your React project, named "public". In this folder, there is an `index.html` file.

You'll notice a single `<div>` in the body of this file. This is where our React application will be rendered.

Example:

Display a paragraph inside an element with the id of "root":

```
const container = document.getElementById('root');

const root = ReactDOM.createRoot(container);

root.render(<p>Hello</p>);
```

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

```
<body>

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

</body>
```

The HTML Code

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

Do not worry if the syntax is unfamiliar, you will learn more about JSX in the next chapter.

Example:

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

```
const myelement = (

  <table>

    <tr>

      <th>Name</th> </tr>

    <tr>

      <td>John</td> </tr>

    <tr>

      <td>Elsa</td> </tr>

  </table>

);


const container = document.getElementById('root');

const root = ReactDOM.createRoot(container);

root.render(myelement);
```

The Root Node

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

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

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

Example:

The root node can be called whatever you like:

```
<body>


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


</body>
```

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

```
const container = document.getElementById('sandy');

const root = ReactDOM.createRoot(container);

root.render(<p>Hallo</p>);
```

•React JSX

What is JSX?

JSX stands for JavaScript XML.

JSX allows us to write HTML in React.

JSX makes it easier to write and add HTML in React.

Coding JSX

JSX allows us to write HTML elements in JavaScript and place them in the DOM without any `createElement()` and/or `appendChild()` methods.

JSX converts HTML tags into react elements.

Example 1:

JSX:

```
const myElement = <h1>I Love JSX!</h1>;


const root = ReactDOM.createRoot(document.getElementById('root'));

root.render(myElement);
```

Example 2:

Without JSX:

```
const myElement = React.createElement('h1', {}, 'I do not use JSX!');


const root = ReactDOM.createRoot(document.getElementById('root'));

root.render(myElement);
```

Expressions in JSX

With JSX you can write expressions inside curly braces `{ }`.

The expression can be a React variable, or property, or any other valid JavaScript expression. JSX will execute the expression and return the result:

Example:

Execute the expression `5 + 5`:

```
const myElement = <h1>React is {5 + 5} times better with JSX</h1>;
```

Inserting a Large Block of HTML

To write HTML on multiple lines, put the HTML inside parentheses:

Example:

Create a list with three list items:

```
const myElement = (
  <ul>
    <li>Apples</li>
    <li>Bananas</li>
    <li>Cherries</li>
  </ul>
);
```

One Top Level Element

The HTML code must be wrapped in *ONE* top level element.

So if you like to write *two* paragraphs, you must put them inside a parent element, like a `div` element.

Example:

Wrap two paragraphs inside one DIV element:

```
const myElement = (
  <div>
    <p>I am a paragraph.</p>
    <p>I am a paragraph too.</p>
  </div>
);
```

Alternatively, you can use a "fragment" to wrap multiple lines. This will prevent unnecessarily adding extra nodes to the DOM.

A fragment looks like an empty HTML tag: `<></>`.

Example:

Wrap two paragraphs inside a fragment:

```
const myElement = (
  <>
    <p>I am a paragraph.</p>
    <p>I am a paragraph too.</p>
  </>
);
```

Elements Must be Closed

JSX follows XML rules, and therefore HTML elements must be properly closed.

Example:

```
const myElement = <input type="text" />;
```

Attribute class = className

The `class` attribute is a much used attribute in HTML, but since JSX is rendered as JavaScript, and the `class` keyword is a reserved word in JavaScript, you are not allowed to use it in JSX.

JSX solved this by using `className` instead. When JSX is rendered, it translates `className` attributes into `class` attributes.

Example:

Use attribute `className` instead of `class` in JSX:

```
const myElement = <h1 className="myclass">Hello World</h1>;
```

Conditions - if statements

React supports `if` statements, but not *inside* JSX.

To be able to use conditional statements in JSX, you should put the `if` statements outside of the JSX, or you could use a ternary expression instead:

Write `if` statements outside of the JSX code:

Example:

Write "Hello" if `x` is less than 10, otherwise "Goodbye":

```
const x = 5;

let text = "Goodbye";

if (x < 10) {

  text = "Hello";

}



const myElement = <h1>{text}</h1>;
```

Use ternary expressions instead:

Example:

Write "Hello" if `x` is less than 10, otherwise "Goodbye":

```
const x = 5;


const myElement = <h1>{(x) < 10 ? "Hello" : "Goodbye"}</h1>;
```

•React Components

Create Your First Component

When creating a React component, the component's name *MUST* start with an upper case letter.

Class Component

A class component must include the `extends React.Component` statement. This statement creates an inheritance to React.Component, and gives your component access to React.Component's functions.

The component also requires a `render()` method, this method returns HTML.

Example:

Create a Class component called `Car`

```
class Car extends React.Component {

  render() {

    return <h2>Hi, I am a Car!</h2>;

  }

}
```

Function Component

Here is the same example as above, but created using a Function component instead.

A Function component also returns HTML, and behaves much the same way as a Class component, but Function components can be written using much less code, are easier to understand, and will be preferred in this tutorial.

Example:

Create a Function component called `Car`

```
function Car() {

  return <h2>Hi, I am a Car!</h2>;

}
```

Rendering a Component

Now your React application has a component called Car, which returns an `<h2>` element.

To use this component in your application, use similar syntax as normal HTML: `<Car />`

Example:

Display the `Car` component in the "root" element:

```
const root = ReactDOM.createRoot(document.getElementById('root'));

root.render(<Car />);
```

Props

Components can be passed as `props`, which stands for properties.

Props are like function arguments, and you send them into the component as attributes.

You will learn more about `props` in the next chapter.

Example:

Use an attribute to pass a color to the Car component, and use it in the render() function:

```
function Car(props) {

  return <h2>I am a {props.color} Car!</h2>;

}


const root = ReactDOM.createRoot(document.getElementById('root'));

root.render(<Car color="red"/>);
```

Components in Components

We can refer to components inside other components:

Example:

Use the Car component inside the Garage component:

```
function Car() {
  return <h2>I am a Car!</h2>;
}

function Garage() {
  return (
    <>
      <h1>Who lives in my Garage?</h1>
      <Car />
    </>
  );
}
const root = ReactDOM.createRoot(document.getElementById('root'));
root.render(<Garage />);
```

Components in Files

React is all about re-using code, and it is recommended to split your components into separate files.

To do that, create a new file with a `.js` file extension and put the code inside it:

Example:

This is the new file, we named it "Car.js":

```
function Car() {
  return <h2>Hi, I am a Car!</h2>;
}
export default Car;
```

•React Class Components

Create a Class Component

When creating a React component, the component's name must start with an upper case letter.

The component has to include the `extends React.Component` statement, this statement creates an inheritance to React.Component, and gives your component access to React.Component's functions.

The component also requires a `render()` method, this method returns HTML.

Example:

Create a Class component called `Car`

```
class Car extends React.Component {

  render() {

    return <h2>Hi, I am a Car!</h2>;

  }

}
```

Now your React application has a component called Car, which returns a `<h2>` element.

To use this component in your application, use similar syntax as normal HTML: `<Car />`

Example:

Display the `Car` component in the "root" element:

```
const root = ReactDOM.createRoot(document.getElementById('root'));

root.render(<Car />);
```

Component Constructor

If there is a `constructor()` function in your component, this function will be called when the component gets initiated.

The constructor function is where you initiate the component's properties.

In React, component properties should be kept in an object called `state`.

You will learn more about `state` later in this tutorial.

The constructor function is also where you honor the inheritance of the parent component by including the `super()` statement, which executes the parent component's constructor function, and your component has access to all the functions of the parent component (`React.Component`).

Example:

Create a constructor function in the Car component, and add a color property:

```
class Car extends React.Component {
  constructor() {
    super();
    this.state = {color: "red"};
  }
  render() {
    return <h2>I am a Car!</h2>;
  }
}
```

Use the color property in the render() function:

Example:

```
class Car extends React.Component {
  constructor() {
    super();
    this.state = {color: "red"};
  }
  render() {
    return <h2>I am a {this.state.color} Car!</h2>;
  }
}
```

Props

Another way of handling component properties is by using `props`.

Props are like function arguments, and you send them into the component as attributes.

You will learn more about `props` in the next chapter.

Example:

Use an attribute to pass a color to the Car component, and use it in the render() function:

```
class Car extends React.Component {
  render() {    return <h2>I am a {this.props.color} Car!</h2>;
  }
}
const root = ReactDOM.createRoot(document.getElementById('root'));
root.render(<Car color="red"/>);
```

Props in the Constructor

If your component has a constructor function, the props should always be passed to the constructor and also to the React.Component via the `super()` method.

Example:

```
class Car extends React.Component {

  constructor(props) {

    super(props);

  }

  render() {

    return <h2>I am a {this.props.model}!</h2>;

  }

}


const root = ReactDOM.createRoot(document.getElementById('root'));

root.render(<Car model="Mustang"/>);
```

Components in Components

We can refer to components inside other components:

Example

Use the Car component inside the Garage component:

```
class Car extends React.Component {
  render() {
    return <h2>I am a Car!</h2>;
  }
}

class Garage extends React.Component {
  render() {
    return (
      <div>
      <h1>Who lives in my Garage?</h1>
      <Car />
      </div>
    );
  }
}

const root = ReactDOM.createRoot(document.getElementById('root'));
root.render(<Garage />);
```

Components in Files

React is all about re-using code, and it can be smart to insert some of your components in separate files.

To do that, create a new file with a `.js` file extension and put the code inside it:

Note that the file must start by importing React (as before), and it has to end with the statement `export default Car;`.

Example:

This is the new file, we named it `Car.js`:

```
import React from 'react';


class Car extends React.Component {

  render() {

    return <h2>Hi, I am a Car!</h2>;

  }

}


export default Car;
```

To be able to use the `Car` component, you have to import the file in your application.

Example:

Now we import the `Car.js` file in the application, and we can use the `Car` component as if it was created here.

```
import React from 'react';

import ReactDOM from 'react-dom/client';

import Car from './Car.js';


const root = ReactDOM.createRoot(document.getElementById('root'));

root.render(<Car />);
```

React Class Component State

React Class components have a built-in `state` object.

You might have noticed that we used `state` earlier in the component constructor section.

The `state` object is where you store property values that belongs to the component.

When the `state` object changes, the component re-renders.

Creating the state Object

The state object is initialized in the constructor:

Example

Specify the `state` object in the constructor method:

```
class Car extends React.Component {

  constructor(props) {

    super(props);

  this.state = {brand: "Ford"};

  }   render() {

    return (

      <div>

        <h1>My Car</h1>

      </div>

    );

  }

}
```

The state object can contain as many properties as you like:

Example:

Specify all the properties your component need:

```
class Car extends React.Component {
  constructor(props) {
    super(props);
    this.state = {
      brand: "Ford",
      model: "Mustang",
      color: "red",
      year: 1964
    };
  }
  render() {
    return (
      <div>
        <h1>My Car</h1>
      </div>
    );
  }
}
```

Using the `state` Object

Refer to the `state` object anywhere in the component by using the `this.state.propertyname` syntax:

Example:

Refer to the `state` object in the `render()` method:

```
class Car extends React.Component {
  constructor(props) {
    super(props);
    this.state = {
      brand: "Ford",
      model: "Mustang",
      color: "red",
      year: 1964
    };
  }
  render() {
    return (
      <div>
        <h1>My {this.state.brand}</h1>
        <p>
          It is a {this.state.color}
          {this.state.model}
          from {this.state.year}.
        </p>
      </div>
    );
  }
}
```

Changing the `state` Object

To change a value in the state object, use the `this.setState()` method.

When a value in the `state` object changes, the component will re-render, meaning that the output will change according to the new value(s).

Example:

Add a button with an `onClick` event that will change the color property:

```
class Car extends React.Component {
  constructor(props) {
    super(props);
    this.state = {
      brand: "Ford",
      model: "Mustang",
      color: "red",
      year: 1964
    };
  }
  changeColor = () => {
    this.setState({color: "blue"});
  }
  render() {
    return (
      <div>
        <h1>My {this.state.brand}</h1>
        <p>
```

```
          It is a {this.state.color}
          {this.state.model}
          from {this.state.year}.
        </p>
        <button
          type="button"
          onClick={this.changeColor}
        >Change color</button>
      </div>
    );
  }
}
```