React Tutorial

React is a JavaScript library for building user interfaces. React is used to build single-page applications. React allows us to create reusable UI components.

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

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

Current version of React.JS is V18.0.0 (April 2022). React.JS was first used in 2011 for Facebook's Newsfeed feature. Current version of create-react-app is v5.0.1 (April 2022). create-react-app includes built tools such as webpack, Babel, and ESLint.

To use React in production, you need npm which is included with Node.js. To get an overview of what React is, you can write React code directly in HTML.

/*

But in order to use React in production, you need npm and Node.js installed.

To learn and test React, you should set up a React Environment on your computer. This tutorial uses the **create-react-app. Node.js** is required to use create-react-app.

Open your terminal in the directory you would like to create your application. Run this command to create a React application named my-react-app:

npx create-react-app my-react-app

Run this command to move to the my-react-app directory:

cd my-react-app

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

npm start

A new browser window will pop up with your newly created React App! If not, open your browser and type localhost:3000 in the address bar.

*/

The quickest way start learning React is to write React directly in your HTML files.

Start by including three scripts, the first two let us write React code in our JavaScripts, and the third, Babel, allows us to write JSX syntax and ES6 in older browsers.

```
<!DOCTYPE html>

<html>

 <head>

  <script src="https://unpkg.com/react@18/umd/react.development.js" crossorigin>
</script>

  <script src="https://unpkg.com/react-dom@18/umd/react-dom.development.js"
crossorigin></script>

  <script src="https://unpkg.com/@babel/standalone/babel.min.js"></script>

 </head>

 <body>


  <div id="mydiv"></div>


  <script type="text/babel">

   function Hello() {

    return <h1>Hello World!</h1>;

   }
```

```
    ReactDOM.render(<Hello />, document.getElementById('mydiv'))
  </script>

 </body>
</html>
```

What is ES6?

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

ES6 introduced classes, assigned inside a constructor() method.

```
class Car {
 constructor(name) {
  this.brand = name;
 }
}
```

Now you can create objects using the Car class:

```
class Car {
 constructor(name) {
  this.brand = name;
 }
}

const mycar = new Car("Ford");
```

You can add your own methods in a class:

```
class Car {
```

```
  constructor(name) {
    this.brand = name;
  }

  present() {
    return 'I have a ' + this.brand;
  }
}

const mycar = new Car("Ford");
mycar.present();
```

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

```
class Car {
  constructor(name) {
    this.brand = name;
  }

  present() {
    return 'I have a ' + this.brand;
  }
}

class Model extends Car {
  constructor(name, mod) {
    super(name);
```

```
    this.model = mod;
  }
  show() {
    return this.present() + ', it is a ' + this.model
  }
}
const mycar = new Model("Ford", "Mustang");
mycar.show();
```

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

Destructuring

We may have an array or object that we are working with, but we only need some of the items contained in these. Destructuring makes it easy to extract only what is needed.

Here is the old way of assigning array items to a variable:

```
const vehicles = ['mustang', 'f-150', 'expedition'];

// old way
const car = vehicles[0];
const truck = vehicles[1];
const suv = vehicles[2];
```

Here is the new way of assigning array items to a variable:

```
const vehicles = ['mustang', 'f-150', 'expedition'];
const [car, truck, suv] = vehicles;
```

If we only want the car and suv we can simply leave out the truck but keep the comma:

```
const vehicles = ['mustang', 'f-150', 'expedition'];
```

```
const [car,, suv] = vehicles;
```

Spread Operator

The JavaScript spread operator (...) allows us to quickly copy all or part of an existing array or object into another array or object.

```
const numbersOne = [1, 2, 3];

const numbersTwo = [4, 5, 6];

const numbersCombined = [...numbersOne, ...numbersTwo];
```

Assign the first and second items from numbers to variables and put the rest in an array:

```
const numbers = [1, 2, 3, 4, 5, 6];

const [one, two, ...rest] = numbers;
```

Ternary Operator

The ternary operator is a simplified conditional operator like if / else.

Syntax: condition ? <expression if true> : <expression if false>

```
authenticated ? renderApp() : renderLogin();
```

React renders HTML to the web page by using a function called ReactDOM.render().

```
import React from 'react';

import ReactDOM from 'react-dom/client';

ReactDOM.render(<p>Hello</p>, document.getElementById('root'));
```

The HTML Code

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

```
const myelement = (
```

```
 <table>
  <tr>
   <th>Name</th>
  </tr>
  <tr>
   <td>John</td>
  </tr>
  <tr>
   <td>Elsa</td>
  </tr>
 </table>
);

ReactDOM.render(myelement, document.getElementById('root'));
```

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.

The root node can be called whatever you like:

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

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

```
ReactDOM.render(<p>Hallo</p>, document.getElementById('sandy'));
```

React JSX

JSX stands for JavaScript XML. JSX allows us to write 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);
```

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

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

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

```
const myElement = (
  <ul>
    <li>Apples</li>
```

```
  <li>Bananas</li>
  <li>Cherries</li>
 </ul>
);
```

React Components:

Some are built-in but you can create your own components as well:

React Class, props, events, conditionals, lists, forms, router, memo.

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:

Note that the filename must start with an uppercase character.

Example

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

```
function Car() {
 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 Props

Props are arguments passed into React components. Props are passed to components via HTML attributes. props stands for properties. React Props are like function arguments in JavaScript and attributes in HTML.

```
import React from 'react';

import ReactDOM from 'react-dom/client';

function Car(props) {

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

}

const myElement = <Car brand="Ford" />;

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

root.render(myElement);
```

React Events

React has the same events as HTML: click, change, mouseover etc.

React events are written in camelCase syntax:

onClick instead of onclick.

React event handlers are written inside curly braces:

onClick={shoot} instead of onClick="shoot()".

React:

```
<button onClick={shoot}>Take the Shot!</button>
```

HTML:

```
<button onclick="shoot()">Take the Shot!</button>
```

Put the shoot function inside the Football component:

```
function Football() {
  const shoot = () => {
    alert("Great Shot!");
  }

  return (
    <button onClick={shoot}>Take the shot!</button>
  );
}

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

To pass an argument to an event handler, use an arrow function.

Send "Goal!" as a parameter to the shoot function, using arrow function:

```
function Football() {
  const shoot = (a) => {
    alert(a);
  }
```

```
  return (
    <button onClick={() => shoot("Goal!")}>Take the shot!</button>
  );
}

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

if Statement

We'll use these two components:

```
function MissedGoal() {
  return <h1>MISSED!</h1>;
}

function MadeGoal() {
  return <h1>Goal!</h1>;
}
```

Now, we'll create another component that chooses which component to render based on a condition:

```
function Goal(props) {
  const isGoal = props.isGoal;
  if (isGoal) {
    return <MadeGoal/>;
  }
```

```
  return <MissedGoal/>;
}

const root = ReactDOM.createRoot(document.getElementById('root'));
root.render(<Goal isGoal={false} />);
```

Try changing the isGoal attribute to true.

React Forms

Submitting Forms

You can control the submit action by adding an event handler in the onSubmit attribute for the <form>:

Example:

Add a submit button and an event handler in the onSubmit attribute:

```
import { useState } from 'react';
import ReactDOM from 'react-dom/client';

function MyForm() {
  const [name, setName] = useState("");

  const handleSubmit = (event) => {
    event.preventDefault();
    alert(`The name you entered was: ${name}`)
  }
```

```
  return (
    <form onSubmit={handleSubmit}>
      <label>Enter your name:
        <input
          type="text"
          value={name}
          onChange={(e) => setName(e.target.value)}
        />
      </label>
      <input type="submit" />
    </form>
  )
}

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

Anytime we link to an internal path, we will use <Link> instead of <a href="">.

Layout.js:

```
import { Outlet, Link } from "react-router-dom";

const Layout = () => {
  return (
    <>
```

```
    <nav>
     <ul>
      <li>
        <Link to="/">Home</Link>
      </li>
      <li>
        <Link to="/blogs">Blogs</Link>
      </li>
      <li>
        <Link to="/contact">Contact</Link>
      </li>
     </ul>
    </nav>

    <Outlet />
   </>
  )
};

export default Layout;
```

Styling React Using CSS

There are many ways to style React with CSS, this tutorial will take a closer look at three common ways:

- Inline styling
- CSS stylesheets

- CSS Modules

Inline Styling

To style an element with the inline style attribute, *the value must be a JavaScript object:*

```
const Header = () => {
 return (
  <>
   <h1 style={{color: "red"}}>Hello Style!</h1>
   <p>Add a little style!</p>
  </>
 );
}
```

You can also create an object with styling information, and refer to it in the style attribute:

Create a style object named myStyle:

```
const Header = () => {
 const myStyle = {
  color: "white",
  backgroundColor: "DodgerBlue",
  padding: "10px",
  fontFamily: "Sans-Serif"
 };
 return (
  <>
   <h1 style={myStyle}>Hello Style!</h1>
```

```
    <p>Add a little style!</p>
  </>
 );
}
```

CSS Stylesheet

You can write your CSS styling in a separate file, just save the file with the .css file extension, and import it in your application.

Create a new file called "**App.css**" and insert some CSS code in it:

```
body {
 background-color: #282c34;
 color: white;
 padding: 40px;
 font-family: Sans-Serif;
 text-align: center;
}
```

Note: You can call the file whatever you like, just remember the correct file extension.

Import the stylesheet in your application:

index.js:

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

```
const Header = () => {
  return (
    <>
      <h1>Hello Style!</h1>
      <p>Add a little style!.</p>
    </>
  );
}

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

CSS Modules

CSS Modules are convenient for components that are placed in separate files. The CSS inside a module is available only for the component that imported it, and you do not have to worry about name conflicts. Create the CSS module with the **.module.css** extension, example: **my-style.module.css**

Create a new file called "**my-style.module.css**" and insert some CSS code in it:

my-style.module.css:

```
.bigblue {
  color: DodgerBlue;
  padding: 40px;
  font-family: Sans-Serif;
  text-align: center;
}
```

Import the stylesheet in your component:

Car.js:

```
import styles from './my-style.module.css';
const Car = () => {
  return <h1 className={styles.bigblue}>Hello Car!</h1>;
}
export default Car;
```

Import the component in your application:

index.js:

```
import ReactDOM from 'react-dom/client';
import Car from './Car.js';
const root = ReactDOM.createRoot(document.getElementById('root'));
root.render(<Car />);
```