**[React JS](https://reactjs.org/tutorial/tutorial.html)**

**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.

[Node.js](https://nodejs.org/) is required to use create-react-app

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

**npx create-react-app my-react-app -** command to create the react application  
  
**Note:** If you've previously installed create-react-app globally, it is recommended that you uninstall the package to ensure npx always uses the latest version of create-react-app. To uninstall, run this command: npm uninstall -g create-react-app.

***Run react application:***

* cd my-react-app
* npm start
* it will pop up new window in browser with localhost:3000

**ES6 concepts useful in react:**

* Classes
* Arrow Functions
* Variables (let, const, var)
* Array Methods like .map()
* Destructuring
* Modules
* Ternary Operator
* Spread Operator

**The Render Function**

The ReactDOM.render() function takes two arguments, HTML code and an HTML element. The purpose of the function is to display the specified HTML code inside the specified HTML element.

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.

**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.

JSX is an extension of the JavaScript language based on ES6, and is translated into regular JavaScript at runtime.

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.

**Components**

Components are like functions that return HTML elements.

Components are independent and reusable bits of code. They serve the same purpose as JavaScript functions but work in isolation and return HTML. Components come in two types, Class components and Function components.

*Before React 16.8, Class components were the only way to track state and lifecycle on a React component. Function components were considered "state-less".*

With the addition of Hooks, Function components are now almost equivalent to Class components. The differences are so minor that you will probably never need to use a Class component in React.

**Class Component**

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

The 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.

class Car extends React.Component {

render() {

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

}

}

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

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

root.render(<Car />);

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.

class Car extends React.Component {

constructor() {

super();

this.state = {color: "red"};

}

render() {

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

}

}

**React Class Component State**

React Class components have a built-in state object. The state object is where you store property values that belongs to the component. When the state object changes, the component re-renders.

**Create state object**

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:

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 state object:

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>

);

}

}

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).

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>

);

}

}

Always use the setState() method to change the state object, it will ensure that the component knows its been updated and calls the render() method (and all the other lifecycle methods).

**Lifecycle of Components**

Each component in React has a lifecycle which you can monitor and manipulate during its three main phases.

The three phases are: **Mounting**, **Updating**, and **Unmounting**.

Mounting means putting elements into the DOM.

React has four built-in methods that gets called, in this order, when mounting a component:

1. constructor()
2. getDerivedStateFromProps()
3. render()
4. componentDidMount()

The render() method is required and will always be called, the others are optional and will be called if you define them.

The next phase in the lifecycle is when a component is *updated*.

A component is updated whenever there is a change in the component's state or props.

React has five built-in methods that gets called, in this order, when a component is updated:

1. getDerivedStateFromProps()
2. shouldComponentUpdate()
3. render()
4. getSnapshotBeforeUpdate()
5. componentDidUpdate()

The render() method is required and will always be called, the others are optional and will be called if you define them.

**Unmounting**

The next phase in the lifecycle is when a component is removed from the DOM, or unmounting as React likes to call it.

React has only one built-in method that gets called when a component is unmounted:

1. componentWillUnmount()

**Props**

Props are arguments passed into React components. Props are passed to components via HTML attributes.

props stands for properties.

**Note:** React Props are read-only! You will get an error if you try to change their value.

**Events**

Just like HTML DOM events, React can perform actions based on user 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()".

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

Event handlers have access to the React event that triggered the function.

function Football() {

const shoot = (a, b) => {

alert(b.type);

/\*

'b' represents the React event that triggered the function,

in this case the 'click' event

\*/

}

return (

<button onClick={(event) => shoot("Goal!", event)}>Take the shot!</button>

);

}

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

root.render(<Football />);

**Conditional Rendering**

If statement

Logical && operator

Ternary operator

**Lists**

In React, you will render lists with some type of loop. The JavaScript map() array method is generally the preferred method.

function Car(props) {

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

}

function Garage() {

const cars = ['Ford', 'BMW', 'Audi'];

return (

<>

<h1>Who lives in my garage?</h1>

<ul>

{cars.map((car) => <Car brand={car} />)}

</ul>

</>

);

}

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

root.render(<Garage />);

When you run this code in your create-react-app, it will work but you will receive a warning that there is no "key" provided for the list items.

## Keys

Keys allow React to keep track of elements. This way, if an item is updated or removed, only that item will be re-rendered instead of the entire list. Keys need to be unique to each sibling. But they can be duplicated globally.

Generally, the key should be a unique ID assigned to each item. As a last resort, you can use the array index as a key.

function Car(props) {

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

}

function Garage() {

const cars = [

{id: 1, brand: 'Ford'},

{id: 2, brand: 'BMW'},

{id: 3, brand: 'Audi'}

];

return (

<>

<h1>Who lives in my garage?</h1>

<ul>

{cars.map((car) => <Car key={car.id} brand={car.brand} />)}

</ul>

</>

);

}

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

root.render(<Garage />);

**Forms**

Just like in HTML, React uses forms to allow users to interact with the web page.

function MyForm() {

return (

<form>

<label>Enter your name:

<input type="text" />

</label>

</form>

)

}

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

root.render(<MyForm />);

This will work as normal, the form will submit and the page will refresh.

We want to prevent this default behavior and let React control the form.

**Handling Forms**

Handling forms is about how you handle the data when it changes value or gets submitted.

In HTML, form data is usually handled by the DOM.

In React, form data is usually handled by the components.

When the data is handled by the components, all the data is stored in the component state.

You can control changes by adding event handlers in the onChange attribute.

We can use the useState Hook to keep track of each inputs value and provide a "single source of truth" for the entire application.

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 />);

## Multiple Input Fields

You can control the values of more than one input field by adding a name attribute to each element.

We will initialize our state with an empty object.

To access the fields in the event handler use the event.target.name and event.target.value syntax.

To update the state, use square brackets [bracket notation] around the property name.

import { useState } from 'react';

import ReactDOM from 'react-dom/client';

function MyForm() {

const [inputs, setInputs] = useState({});

const handleChange = (event) => {

const name = event.target.name;

const value = event.target.value;

setInputs(values => ({...values, [name]: value}))

}

const handleSubmit = (event) => {

event.preventDefault();

alert(inputs);

}

return (

<form onSubmit={handleSubmit}>

<label>Enter your name:

<input

type="text"

name="username"

value={inputs.username || ""}

onChange={handleChange}

/>

</label>

<label>Enter your age:

<input

type="number"

name="age"

value={inputs.age || ""}

onChange={handleChange}

/>

</label>

<input type="submit" />

</form>

)

}

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

root.render(<MyForm />);

**Note:** We use the same event handler function for both input fields, we could write one event handler for each, but this gives us much cleaner code and is the preferred way in React.

Text-area

import { useState } from 'react';

import ReactDOM from 'react-dom/client';

function MyForm() {

const [textarea, setTextarea] = useState(

"The content of a textarea goes in the value attribute"

);

const handleChange = (event) => {

setTextarea(event.target.value)

}

return (

<form>

<textarea value={textarea} onChange={handleChange} />

</form>

)

}

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

root.render(<MyForm />);

**Select**

function MyForm() {

const [myCar, setMyCar] = useState("Volvo");

const handleChange = (event) => {

setMyCar(event.target.value)

}

return (

<form>

<select value={myCar} onChange={handleChange}>

<option value="Ford">Ford</option>

<option value="Volvo">Volvo</option>

<option value="Fiat">Fiat</option>

</select>

</form>

)

}

**Router**

Create React App doesn't include page routing.

React Router is the most popular solution.

**npm i -D react-router-dom** - To add React Router in your application

If you are upgrading from v5, you will need to use the @latest flag:

**npm i -D react-router-dom@latest**

**Memo**

Using memo will cause React to skip rendering a component if its props have not changed. This can improve performance.

**Hooks**

Hooks were added to React in version 16.8.

Hooks allow function components to have access to state and other React features. Because of this, class components are generally no longer needed.

Hooks allow us to "hook" into React features such as state and lifecycle methods.

import React, { useState } from "react";

import ReactDOM from "react-dom/client";

function FavoriteColor() {

const [color, setColor] = useState("red");

return (

<>

<h1>My favorite color is {color}!</h1>

<button

type="button"

onClick={() => setColor("blue")}

>Blue</button>

<button

type="button"

onClick={() => setColor("red")}

>Red</button>

<button

type="button"

onClick={() => setColor("pink")}

>Pink</button>

<button

type="button"

onClick={() => setColor("green")}

>Green</button>

</>

);

}

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

root.render(<FavoriteColor />);

You must import Hooks from react.

Here we are using the useState Hook to keep track of the application state.

State generally refers to application data or properties that need to be tracked.

**Hook Rules**

There are 3 rules for hooks:

Hooks can only be called inside React function components.

Hooks can only be called at the top level of a component.

Hooks cannot be conditional

**Note:** Hooks will not work in React class components.

**useState**

The React useState Hook allows us to track state in a function component.

State generally refers to data or properties that need to be tracking in an application.

To use the useState Hook, we first need to import it into our component.

import { useState } from "react";

We initialize our state by calling useState in our function component.

useState accepts an initial state and returns two values:

* The current state.
* A function that updates the state.

import { useState } from "react";

function FavoriteColor() {

const [color, setColor] = useState("");

}

Notice that again, we are destructuring the returned values from useState.

The first value, color, is our current state.

The second value, setColor, is the function that is used to update our state.

These names are variables that can be named anything you would like.

Lastly, we set the initial state to an empty string: useState("")

**Read State**

We can now include our state anywhere in our component.

import { useState } from "react";

import ReactDOM from "react-dom/client";

function FavoriteColor() {

const [color, setColor] = useState("red");

return <h1>My favorite color is {color}!</h1>

}

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

root.render(<FavoriteColor />);

## Update State

To update our state, we use our state updater function.

We should never directly update state. Ex: color = "red" is not allowed.

Use a button to update the state:

import { useState } from "react";

import ReactDOM from "react-dom/client";

function FavoriteColor() {

const [color, setColor] = useState("red");

return (

<>

<h1>My favorite color is {color}!</h1>

<button

type="button"

onClick={() => setColor("blue")}

>Blue</button>

</>

)

}

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

root.render(<FavoriteColor />);

The useState Hook can be used to keep track of strings, numbers, booleans, arrays, objects, and any combination of these!

We could create multiple state Hooks to track individual values.

import { useState } from "react";

import ReactDOM from "react-dom/client";

function Car() {

const [car, setCar] = useState({

brand: "Ford",

model: "Mustang",

year: "1964",

color: "red"

});

return (

<>

<h1>My {car.brand}</h1>

<p>

It is a {car.color} {car.model} from {car.year}.

</p>

</>

)

}

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

root.render(<Car />);

Since we are now tracking a single object, we need to reference that object and then the property of that object when rendering the component. (Ex: car.brand)

**Updating Objects and Arrays in State**

When state is updated, the entire state gets overwritten.

What if we only want to update the color of our car?

If we only called setCar({color: "blue"}), this would remove the brand, model, and year from our state.

We can use the JavaScript spread operator to help us.

import { useState } from "react";

import ReactDOM from "react-dom/client";

function Car() {

const [car, setCar] = useState({

brand: "Ford",

model: "Mustang",

year: "1964",

color: "red"

});

const updateColor = () => {

setCar(previousState => {

return { ...previousState, color: "blue" }

});

}

return (

<>

<h1>My {car.brand}</h1>

<p>

It is a {car.color} {car.model} from {car.year}.

</p>

<button

type="button"

onClick={updateColor}

>Blue</button>

</>

)

}

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

root.render(<Car />);

Because we need the current value of state, we pass a function into our setCar function. This function receives the previous value.

We then return an object, spreading the previousState and overwriting only the color.

**useEffect**

The useEffect Hook allows you to perform side effects in your components.

Some examples of side effects are: fetching data, directly updating the DOM, and timers.

useEffect accepts two arguments. The second argument is optional.

useEffect(<function>, <dependency>)

import { useState, useEffect } from "react";

import ReactDOM from "react-dom/client";

function Timer() {

const [count, setCount] = useState(0);

useEffect(() => {

setTimeout(() => {

setCount((count) => count + 1);

}, 1000);

});

return <h1>I've rendered {count} times!</h1>;

}

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

root.render(<Timer />);

It keeps counting even though it should only count once!

useEffect runs on every render. That means that when the count changes, a render happens, which then triggers another effect.

This is not what we want. There are several ways to control when side effects run.

We should always include the second parameter which accepts an array. We can optionally pass dependencies to useEffect in this array.

### **1. No dependency passed:**

useEffect(() => {

//Runs on every render

});

### **2. An empty array:**

useEffect(() => {

//Runs only on the first render

}, []);

### **3. Props or state values:**

useEffect(() => {

//Runs on the first render

//And any time any dependency value changes

}, [prop, state]);

So, to fix this issue, let's only run this effect on the initial render.

import { useState, useEffect } from "react";

import ReactDOM from "react-dom/client";

function Timer() {

const [count, setCount] = useState(0);

useEffect(() => {

setTimeout(() => {

setCount((count) => count + 1);

}, 1000);

}, []); // <- add empty brackets here

return <h1>I've rendered {count} times!</h1>;

}

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

root.render(<Timer />);

Here is an example of a useEffect Hook that is dependent on a variable. If the count variable updates, the effect will run again:

import { useState, useEffect } from "react";

import ReactDOM from "react-dom/client";

function Counter() {

const [count, setCount] = useState(0);

const [calculation, setCalculation] = useState(0);

useEffect(() => {

setCalculation(() => count \* 2);

}, [count]); // <- add the count variable here

return (

<>

<p>Count: {count}</p>

<button onClick={() => setCount((c) => c + 1)}>+</button>

<p>Calculation: {calculation}</p>

</>

);

}

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

root.render(<Counter />);

If there are multiple dependencies, they should be included in the useEffect dependency array.

## Effect Cleanup

Some effects require cleanup to reduce memory leaks.

Timeouts, subscriptions, event listeners, and other effects that are no longer needed should be disposed.

We do this by including a return function at the end of the useEffect Hook.

Clean up the timer at the end of the useEffect Hook:

import { useState, useEffect } from "react";

import ReactDOM from "react-dom/client";

function Timer() {

const [count, setCount] = useState(0);

useEffect(() => {

let timer = setTimeout(() => {

setCount((count) => count + 1);

}, 1000);

return () => clearTimeout(timer)

}, []);

return <h1>I've rendered {count} times!</h1>;

}

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

root.render(<Timer />);

**Note:** To clear the timer, we had to name it.