ReactJS

What is react JS

- A Javascript library created by Facebook in 2011 for building user interfaces
- Single page application
- Dynamic web apps
- Many giants use React including Facebook, Whats-app, Instagram,…

Core Concepts of React

- Component
- JSX
- Virtual DOM

Server-less Hello World Example

- Lets start by building a single HTML file that uses react to display a simple page on the browser..Add the basic html tags then include the react library.

index.html

```
<!DOCTYPE HTML>
<html>
<head>
        <meta charset="utf-8">
        <title>Pro MERN Stack</title>
        <script src="https://unpkg.com/react@16/umd/react.development.js"></script>
        <script src="https://unpkg.com/react-dom@16/umd/react-dom.development.js"></script>
</head>
<body>
        <div id="contents"></div>
            ……
</body>
</html>
```

React library

- The React library is available as a JavaScript file that we can include in the HTML file using the<script> tag.
- It comes in two parts:
  - the first is the React core module, the one that is responsible for dealing with React components, their state manipulation, etc.
  - the second is the ReactDOM module, which deals with converting React components to a DOM that a browser can understand.
- The development version of the libraries can be accessed via the following URLs:

React: https://unpkg.com/react@16/umd/react.development.js

ReactDOM: https://unpkg.com/react-dom@16/umd/react-dom.development.js

Hello World Example 1

- Make sure you have a with an id where all view will be rendered in

```
<!DOCTYPE HTML>
<html>
<head>
        ……..
</head>
<body>
        <div id="contents"></div>


                ……..
</body>
</html>
```

React module createElement() function

- To create the React element, the createElement() function of the React module needs to be called.
- The function takes up to three arguments and its prototype is as follows:

```
React.createElement(type, [props], [...children])
```

- type can be any HTML tag such as the string 'div', or a React component
- props is an object containing HTML attributes or custom component properties.
- children is zero or more children elements, which again are created using the createElement() function itself.

Hello World Example 1

- Using createElment(), create a simple <h1> Hello World </h1>

```
<!DOCTYPE HTML>
<html>
<head>
        ……..
</head>
<body>
        <div id="contents"></div>
        <script>
        const title = React.createElement('h1', {}, ‘Hello World');
                        …….
 </script>
</body>
</html>
```

Hello World Example 2

- Using createElment(), create a simple that includes a nested <h1> Hello World </h1>

```
<!DOCTYPE HTML>
<html>
<head>
        ……..
</head>
<body>
        <div id="contents"></div>
        <script>
          const element = React.createElement('div', { title: 'Outer div' },
          React.createElement('h1', null, 'Hello World!')
          );
                ……..
 </script>
</body>
</html>
```

Hello World Example 2 version2

- Using createElement(), create a simple that includes a nested <h1> Hello World </h1>

```
<!DOCTYPE HTML>
<html>
<head>
        ……..
</head>
<body>
        <div id="contents"></div>
        <script>
          const title = React.createElement(‘h1', { }, ‘Hello World’);
          const element=React.createElement(‘div', { title: 'Outer div' }, title);

                ……..
 </script>
</body>
</html>
```

Hello World Example 3

- Add a paragraph as new sibling to the title element.

```
<!DOCTYPE HTML>
<html>
<head>
        ……..
</head>
<body>
        <div id="contents"></div>
        <script>
          const title = React.createElement(‘h1', { }, ‘Hello World’);
          const paragraph = React.createElement('p', {}, 'Writing some more HTML. Cool stuff!');
          const element=React.createElement(‘div', { title: 'Outer div' }, [title,paragraph]);
                        …….
 </script>
</body>
</html>
```

Hello World Example 4

- We can nest children as much as we want. We also don't need to store our elements in variables before using them, we can declare them inline as well:

```
<body>
        <div id="contents"></div>
        <script>
          const list = React.createElement('div', {},
           React.createElement('h1', {}, 'My favorite ice cream flavors'),
           React.createElement('ul', {},
            [
              React.createElement('li', {}, 'Chocolate'),
             React.createElement('li‘, {}, 'Vanilla'),
              React.createElement('li', {}, 'Banana')
            ]
          )
       );
        </script>
</body>
</html>
```

Hello World Example

- After creating the React element, we need to render it.

```
<!DOCTYPE HTML>
<html>
<head>
        ……..
</head>
<body>
        <div id="contents"></div>
        <script>
                const element = React.createElement('div', {title: 'Outer div'},
                React.createElement('h1', null, 'Hello World!')
                     );
                ReactDOM.render(element, document.getElementById('content     );

</script>
</body>
</html>
```

ReactDOM.render()

- 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 ReactDOM.render().

Syntax

```
ReactDOM.render(element, container[, callback])
```

- It returns a reference to the component (or returns null for stateless components).

JSX (Javascript XML)

- Assume we are writing a deeply nested hierarchy of elements and components: it can get pretty complex.
- JSX looks very much like HTML, but there are some differences.
- JSX can be used to construct an element or an element hierarchy and make it look very much like HTML,making understanding how the screen will look like very easy

```
const Elem = (
<div title="Outer div">
<h1>Hello World!</h1>
</div>
);
...
```

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

Browsers' JavaScript engines don't understand JSX

- JSX has to be transformed into regular JavaScript.
- Babel provides a standalone compiler that can be used in the browser.
- <script src="https://unpkg.com/@babel/standalone@7/babel.min.js">
  </script>
- But the compiler also needs to be told which scripts have to be transformed.
- It looks for the attribute type="text/babel" in all scripts and transforms and runs any script with this attribute.

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:

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

Inserting a Large Block of HTML

- Inserting a Large Block of HTML
- The HTML code must be wrapped in *ONE* top level element or in a fragment <></>

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

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

- All elements must be closed

Hello World Example 5 (JSX)

```
<head>
  <meta charset="utf-8">
  <title>Pro MERN Stack</title>

  <script src="https://unpkg.com/react@16/umd/react.development.js"></script>
  <script src="https://unpkg.com/react-dom@16/umd/react-dom.development.js"></script>

  <script src="https://unpkg.com/@babel/standalone@7/babel.min.js"></script>
</head>

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

  <script type="text/babel">
    const element = React.createElement('div', {title: 'Outer div'},
      React.createElement('h1', null, 'Hello World!')
    );
    const element = (
      <div title="Outer div">
        <h1>Hello World!</h1>
      </div>
    );

    ReactDOM.render(element, document.getElementById('contents'));
  </script>
</body>
```

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:

```
const x = 5;
let text = "Goodbye";
if (x < 10) {
  text = "Hello";
}

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

```
const x = 5;

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

Loops in React

- The most common way of doing that is with the map function that will return JSX.
- Whenever you use a loop it is important to provide a unique key attribute.

Why key?

- Whenever you use a loop it is important to provide a unique key attribute.
- The reason is that React uses these keys to track if items were changed, added, or removed.
- As a general rule: if you have an array that can change, then use a unique id. If it is not available, then create one for each item before the list is rendered. Otherwise, it is ok to use an index for the key attribute.

Create React App

- To learn and test React, you should set up a React Environment on your computer.
- The **create-react-app** tool is an officially supported way to create React applications.
- Node.js is required to use create-react-app. (MAKE SURE IT IS INSTALLED)
- 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
```

React components

- 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.

Class Component

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

Function Component

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

}
```

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

Components in Components

```
function Car() {
  return <h2>I am a Car!</h2>;
}
function Garage() {
  return (
    <>
      <h1>Who lives in my Garage?</h1>
      <Car />
    </>
  );
}
ReactDOM.render(<Garage />, document.getElementById('root'));
```

Components in Files

- 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 for the component inside it
- Note that the filename must start with an uppercase character.
- Car.js

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

export default Car;
```

```
import React from 'react';
import ReactDOM from 'react-dom';
import Car from './Car.js';

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

Props

- Components let you split the UI into independent, reusable pieces, and think about each piece in isolation.
- Components are like JavaScript functions. They accept arbitrary inputs (called “props”) and return React elements describing what should appear on the screen.
- Props are like function arguments passed into React components
- You send props into the component via HTML attributes.
- Note: React Props are read-only! You will get an error if you try to change their value.

Example

```
const myelement = <Car brand="Ford" />;
```

- The component receives the argument as a props object:

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

This function is a valid React component because it accepts a single “props” (which stands for properties) object argument with data and returns a React element.

- For example, this code renders “I am a Ford” on the page:
- Let’s recap what happens in this example:

  - We call ReactDOM.render() with the <Car brand="Ford" /> element.
  - React calls the Car component with {brand: ‘Ford'} as the props.
  - Our Car component returns a <h2>I am Ford</h2> element as the result.
  - React DOM efficiently updates the DOM to match <h2>I am Ford</h2>

Passing data between components using props

- Props are also how you pass data from one component to another, as parameters.
- Example: Send the "brand" property from the Garage component to the Car component:

```
function Car(props) {
  return <h2>I am a { props.brand }!</h2>;}
function Garage() {
  return ( <>
     <h1>Who lives in my garage?</h1>
     <Car brand="Ford" />
   </>
  );
}
ReactDOM.render(<Garage />,
document.getElementById('root'));
```

```
function Car(props) {
  return <h2>I am a { props.brand
}!</h2>; }
function Garage() {
  const carName = "Ford";
  return (
   <>
     <h1>Who lives in my garage?</h1>
     <Car brand={ carName } />
   </>
  );  }
ReactDOM.render(<Garage />,
document.getElementById('root'));
```

First, you'll need to create two components, one parent and one child.

Next, you'll import the child component in the parent component and return it.

Passing an object to a component

```
function Car(props) {
  return <h2>I am a { props.brand.model }!</h2>;
}
function Garage() {
  const carInfo = { name: "Ford", model: "Mustang" };
  return (
    <>
      <h1>Who lives in my garage?</h1>
      <Car brand={ carInfo } />
    </>
  );
}

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

Using Props in React

- Firstly, define an attribute and its value(data) with interpolation {} in a react component
- Then pass it to child component(s) by using Props
- Finally, render the Props Data
- To send props into a component, use the same syntax as HTML attributes

React 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

- React events are written in camelCase syntax:

```
onClick
```

- React event handlers are written inside curly braces:

```
onClick={shoot}
```

React Syntax

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

HTML Syntax

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

Example (Football component with shoot function)

```
function Football() {
  const shoot = () => {
    alert("Great Shot!");
  }
  return (
    <button onClick={shoot}>Take the shot!</button>
  );
}
ReactDOM.render(<Football />, document.getElementById('root'));
```

Passing Arguments to event handlers

- 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>);
}
ReactDOM.render(<Football />, document.getElementById('root'));
```

React Event Object

- Event handlers have access to the React event that triggered the function. Example the "click" event.

```
function Football() {
  const shoot = (a,b) => {
/* 'b' represents the React event that triggered the function, in this case the 'click' event */
    alert(b.type);
  }
  return (
    <button onClick={() => shoot("Goal!", event)}>Take the shot!</button>
  );
}
ReactDOM.render(<Football />, document.getElementById('root'));
```

React Conditional Rendering

- In React, you can conditionally render components.
- There are several ways to do this. (see example)
  - if Statement
  - Logical && Operator
  - ternary Operator
-

React Lists

- In React, you will render lists with some type of loop.
- 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.
- 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.
- A key is a special string attribute that you need to include when including lists

React Forms

- Just like in HTML, React uses forms to allow users to interact with the web page.
- You add a form with React like any other element.
- HTML form elements work a bit differently from other DOM elements in React, because form elements naturally keep some internal state. For example, this form in plain HTML accepts a single name:

```
<form>
  <label>
    Name:
    <input type="text" name="name" />
  </label>
  <input type="submit" value="Submit" />
</form>
```

```
function MyForm() {
  return (
    <form>
      <label>Enter your name:
        <input type="text" />
      </label>
    </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.

Submitting Forms

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

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.

Textarea

- The textarea element in React is slightly different from ordinary HTML.

- In HTML the value of a textarea was the text between the start tag <textarea> and the end tag </textarea>.

- In React the value of a textarea is placed in a value attribute. We'll use the useState Hook to mange the value of the textarea:

Select

- A drop down list, or a select box, in React is also a bit different from HTML.

- In HTML, the selected value in the drop down list was defined with the selected attribute:
- In React, the selected value is defined with a value attribute on the select tag.

React Memo

- Using memo will cause React to skip rendering a component if its props have not changed.

- This can improve performance.

Problem – Todos re-renders always

index.js

```
import { useState } from "react";
import ReactDOM from "react-dom";
import Todos from "./Todos";
const App = () => {
  const [count, setCount] = useState(0);
  const [todos, setTodos] = useState(["todo 1", "todo 2"]);
  const increment = () => {
    setCount((c) => c + 1);
  };
  return (
    <>
      <Todos todos={todos} />
      <hr />
      <div>   Count: {count}   <button onClick={increment}>+</button </div>
    </>
  );
};

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

index.js

```
const Todos = ({ todos }) => {
  console.log("child render");
  return (
    <>
      <h2>My Todos</h2>
      {todos.map((todo, index) => {
        return <p key={index}>{todo}</p>;
      })}
    </>
  );
};

export default Todos;
```

Answer:

index.js

```
import { useState } from "react";
import ReactDOM from "react-dom";
import Todos from "./Todos";
const App = () => {
  const [count, setCount] = useState(0);
  const [todos, setTodos] = useState(["todo 1", "todo 2"]);

  const increment = () => {
    setCount((c) => c + 1);
  };
  return (
<>
      <Todos todos={todos} />
      <hr />
      <div>     Count: {cou  <button onClick={increment}>+</button     </div>
    </>
  );
};
ReactDOM.render(<App />, document.getElementById('root'));
```

Todo.js

```
import { memo } from "react";

const Todos = ({ todos }) => {
  console.log("child render");
  return (
    <>
      <h2>My Todos</h2>
      {todos.map((todo, index) => {
        return <p key={index}>{todo}</p>;
      })}
    </>
  );
};

export default memo(Todos);
```

Styling React Using CSS

- There are many ways to style React with CSS, the three common ways are:
  - Inline styling
  - CSS stylesheets
  - CSS Modules

Inline Styling

- To style an element with the inline style attribute, the value must be a JavaScript object.
- Since the inline CSS is written in a JavaScript object, properties with hyphen separators, like background-color, must be written with camel case syntax:

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

JavaScript Object

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

```
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.

```
import './App.css';
```

CSS Modules

- Another way of adding styles to your application is to use 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 with some CSS in it
- Import the stylesheet in your component:

```
import styles from './my-style.module.css';
```

- Import the component in your application:
-

```
import Car from './Car.js';
```

React Hooks

- 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.

- 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

React useState Hook

- 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.

Initialize useState

- We initialize our state by calling **useState** in our function component.
- useState accepts an initial state and returns an array of two values:

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

Read and Update State

- We can now include our state anywhere in our component in order to read it.

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

What Can State Hold

- 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

useEffect

- The useEffect Hook allows you to perform side effects in your components.
- useEffect runs on every render.

- 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>)

How to control when effects run?

- useEffect runs on every render. Any render that happens is going to trigger the useEffect.
- There are several ways to control when side effects run.
- Using the second parameter which accepts an array. We can optionally pass dependencies to useEffect in this array.

```
useEffect(() => {
  //Runs on every render
})
```

No dependencies passed

```
useEffect(() => {
  //Runs only on the first render
}, []);
```

An empty dependencies array

```
useEffect(() => {
  //Runs on the first render
  //And any time any dependency value changes
}, [prop, state]);
```

Props or state

useContext

- React Context is a way to manage state globally.

- It can be used together with the useState Hook to share state between deeply nested components more easily than with useState alone.
- To create context,
  - you must Import createContext and initialize it
  - Wrap child components in the Context Provider and supply the state value.
  - In order to use the Context in a child component, we need to access it using the useContext Hook.

useRef

- The useRef Hook is similar to useState, but different
- The Hook call returns an object that has a property current, which stores the actual value. If you pass an argument initialValue to useRef(initialValue), then this value is stored in current.
- To access a ref's value, you need to access its current property,
- The values of refs persist (specifically the current property) throughout render cycles. It's not a bug; it's a feature.
- useRef() only returns one item. It returns an Object called current.

- When we initialize useRef we set the initial value: useRef(0).

useMemo

- The React useMemo Hook returns a memoized value.
- Think of memoization as caching a value so that it does not need to be recalculated.
- The useMemo Hook only runs when one of its dependencies update.
- This can improve performance.