**Objectives:**

1. Getting Started
2. Understanding the Base Features & Syntax
3. Creating a Functional Component
4. Class-based vs Functional Components
5. Component Update Lifecycle (for props Changes)
6. Props & State
7. Understanding JSX
8. Working with Lists and Conditionals
9. Forms and Form Validation

**Prerequisites:**

Before starting with React.JS, you should have intermediate experience in:

* 1. HTML
  2. CSS
  3. JavaScript
  4. ES6
  5. NPM and Node.js

You should also have some experience with the new JavaScript features introduced in ECMAScript 6 [ES6- Classes, Arrow Functions, Variables (let, const, var)]

**Online references:**

<https://reactjs.org/tutorial/tutorial.html>

<https://www.w3schools.com/react/default.asp>

**Environment Setup:**

1. <https://nodejs.org/en/download/>
2. https://code.visualstudio.com/download

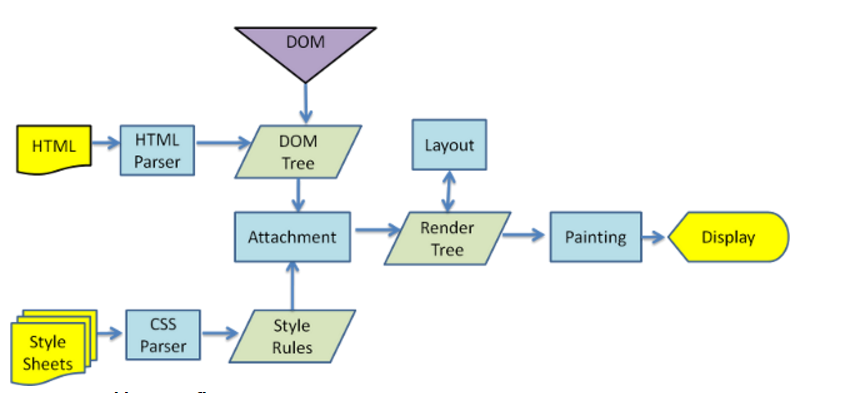
**What Is React?**

1. React is an open source JavaScript library for building user interfaces.
2. React is used for only building UI.
3. React is used to build single page applications.
4. React allows us to create reusable UI components.
5. React has component based architecture. This lets you break down your application into small encapsulated parts which can then be composed to make more complex UI.

**How does React Work?**

1. React creates a VIRTUAL DOM in memory.
2. 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.
3. React only changes what needs to be changed!
4. React finds out what changes have been made, and changes **only** what needs to be changed.
5. React will handle efficiently updating and rendering just the right components in your application when your data changes. DOM update which is one of the more expensive operations is handled gracefully in React with concept called Virtual DOM

**Real DOM vs. Virtual DOM:**



**Following thing happens:**

1. The browser has to parse the HTML
2. It removes the child element of elementId
3. Updates the DOM with the “New Value”
4. Re-calculate the CSS for the parent and child
5. Update the layout i.e. each elements exact coordinates on the screen
6. Traverse the render tree and paint it on the browser display

**Virtual DOM**

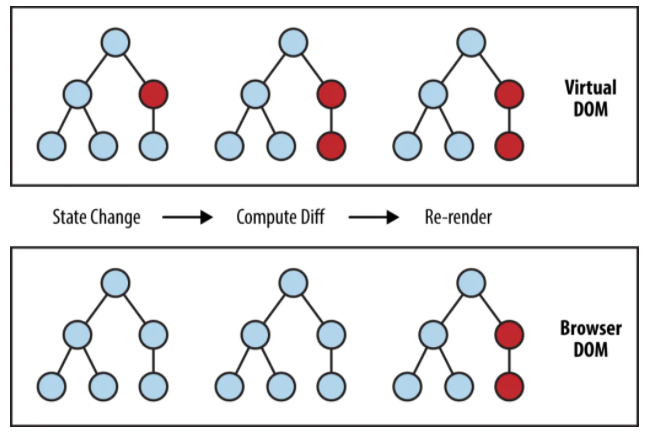
Virtual DOM performs significantly better than the real DOM. The virtual DOM is only a virtual representation of the DOM. Every time the state of our application changes, the virtual DOM gets updated instead of the real DOM.

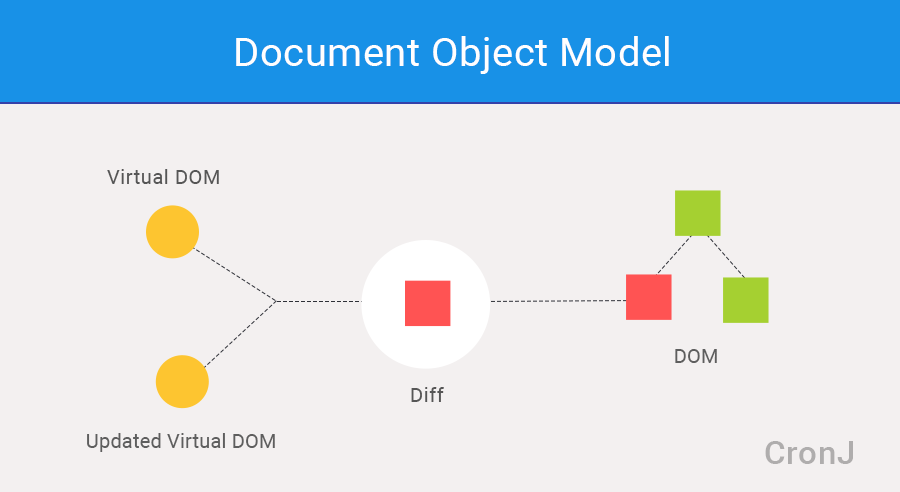
**How is Virtual DOM faster?**

When new elements are added to the UI, a virtual DOM, which is represented as a tree is created. Each element is a node on this tree. If the state of any of these elements changes, a new virtual DOM tree is created. This tree is then compared or “diffed” with the previous virtual DOM tree.

Once this is done, the virtual DOM calculates the best possible method to make these changes to the real DOM. This ensures that there are minimal operations on the real DOM. Hence, reducing the performance cost of updating the real DOM.

The image below shows the virtual DOM tree and the diffing process.





**Why React?**

1. React is declarative. React will make it painless for you to create complex UIs by abstracting away the difficult parts.
2. It works with other Java Script libraries very well.(Ex : It does not have routes and services , But it can work with other libraries like Redux, Flux and Thunk to

acheive this).

1. React is a project created and maintained by Facebook.
2. React has more than a 100 thousand stars on GitHub and a huge community behind it.
3. React has become increasingly popular among developers and is also one of the most sought out skill-sets by companies right now.

**Summary:** React is a declarative, efficient, and flexible JavaScript library for building user interfaces. It lets you compose complex UIs from small and isolated pieces of code called “components”.

**Create React App**

**Using NPX**

**C:\Users\Your Name>npx create-react-app <project-name>**

**cd project-name**

**npm start**

**Using NPM**

**C:\Users\Your Name>npm install -g create-react-app**

**create-react-app project-name**

**npm start**

**React folder structure:**

**Components:**

Two types of components

1. Functional Components
2. Class Components

**Functional Components:**

**Input - (props) Output – JSX(HTML)**

Functional Component

Example:

import React from'react'

function Greet() {

return (

<h1>Welcome to React. this is the first functional component</h1>

);

}

export default Greet;

**Arrow Function:**

const Greet = () => <h1>Welcome to React. this is the first functional component</h1>

Any Name we can give while importing

**Named Export**

import {Greet} from './Greet'

export const Greet = () => <h1>Welcome to React. this is the first functional component</h1>

**Class Component:**

**Input - (props) Output – JSX (HTML)**

Class Component

**Class Component**

import React, {Component} from 'react'

class Welcome extends Component{

render(){

return <h1>Welcome to React. this is the first class component which can maintain the state</h1>

}

}

export default Welcome

**Passing Props to functional component:**

**Example1:**

function Greet (props) {

return <h1>Hello {props.batch}</h1>;

}

**Example2:**

const Greet = props => {

return (

<div>

<h1>

Hello, {props. daypart } {props.batch }

</h1>

//{props.children}

</div>

)

}

export default Greet

App.js

<Greet daypart="Good Morning" batch="Batch" >

//<p>This is children props</p>

</Greet>

<Greet daypart="Good Afternoon" batch="Batch" >

//<button>Action</button>

</Greet>

<Greet daypart="Good Evening" batch="Batch" />

**Passing Props to class component:**

class Welcome extends Component {

render() {

return <h1>Welcome {this.props.batch} <h1>

// return <h1>Hello, {this.props. daypart } {this.props. batch} <h1>

}

}

export default Welcome

App.js

<Welcome batch ="batch" daypart ="Good Morning" />

<Welcome batch ="batch" daypart ="Good Afternoon" />

<Welcome batch ="batch" daypart ="Good Evening" />

Note: Props are immutable

**Difference between Functional and Class Components:**

|  |  |
| --- | --- |
| **Functional Components** | **Class Components** |
| A functional component is just a plain JavaScript function that accepts props as an argument and returns a React element. | A class component requires you to extend from **React.Component** class and create a **render** function which returns a React element. |
| There is no render method used in functional components. | It must have the render() method returning HTML |
| Also known as Stateless components as they simply accept data and display them in some form that is they are mainly responsible for rendering UI. In React 16.8 hooks are added to maintain the state in functional component. | Also known as Stateful components because they implement logic and state. |
| React lifecycle methods cannot be used in functional components. | React lifecycle methods can be used inside class components. |
| These components have less code can be easy to debug, test. |  |

**JSX (Java Script XML)**

JSX stands for JavaScript XML, extension to Java Script.

JSX allows us to write HTML in React.

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

React [doesn’t require](https://reactjs.org/docs/react-without-jsx.html) using JSX, but most people find it helpful as a visual aid when working with UI inside the JavaScript code. It also allows React to show more useful error and warning messages.

**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.

You are not required to use JSX, but JSX makes it easier to write React applications.

**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:**

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

Note:

* JS JSX follows XML rules, and therefore HTML elements must be properly closed.
* JSX will throw an error if the HTML is not correct, or if the HTML misses a parent element.
* It must return only one top level element and others will child.

**Example:**

const Hello = () => {

//with JSX

return (

<div className='dummyClass'>

<h1>Hello User</h1>

</div>

)

//Without Jsx

// return React.createElement(

// 'div',

// {id: 'hello', className: 'dummyClass'},

// React.createElement('h1', null, 'Hello User')

// )

}

export default Hello

**State in React:**

**State:** Components in React are independent and reusable pieces of code that often contain their own state. They return React elements that make up the UI of an application. Components that contain local state have a property called **state** when we want to change our how application looks or behaves; we need to change our component’s state. So, how do we update the state of our component? React components have a method available to them called setState Calling this.setState causes React to re-render your application and update the DOM.

|  |  |  |
| --- | --- | --- |
| **SN** | **Props** | **State** |
| 1. | Props are read-only and immutable. | State changes can be asynchronous and mutable. |
| 2. | Props allow you to pass data from one component to other components as an argument and accessed by the child component. | State holds information about the components and can’t be accessed in other components. |
| 3. | Props are used to communicate between components. | States can be used for rendering dynamic changes with the component. |
| 4. | Props make components reusable. | State cannot make components reusable. |
| 5. | Props are external and controlled by whatever renders the component. | The State is internal and controlled by the React Component itself. |

**Example 1:**

<ChildComponent />

<ChildComponent color=green />

class ChildComponent extends React.Component {

constructor(props) {

super(props)

console.log(props.color)

}

}

class ChildComponent extends React.Component {

constructor(props) {

super(props)

this.state.colorName = props.color

}

}

**Example 2:**

class Message extends Component {

constructor(props) {

super(props)

this.state = {

message: props.message

}

}

changeMessage() {

this.setState({

message: ‘Successfully registered Please login !'

})

}

render() {

return (

<div>

<h1>{this.state.message}</h1>

<button onClick={() => this.changeMessage()}>Register</button>

</div>

)

}

}

export default Message

**Example 3:**

import React, { Component } from 'react'

class Counter extends Component {

constructor() {

super()

this.state = {

count: 0

}

}

increment() {

this.setState((prevState) => ({

count: prevState.count + 1

}))

// this.setState({

// count: this.state.count + 1

// }, () => {

// console.log('Callback', this.state.count)

// })

// this.state.count = this.state.count + 1

// console.log(this.state.count)

}

incrementFive() {

console.log('Inside incrementFive')

this.increment()

this.increment()

this.increment()

this.increment()

this.increment()

}

render() {

return (

<div>

<div>Count - {this.state.count}</div>

<button onClick={() => this.incrementFive()}>Increment</button>

</div>

)

}

}

export default Counter

**Destructing:**

const Greet = ({daypart, batch}) => {

const {daypart, batch} =props

return (

<div>

<h1>

Hello, {daypart } {batch }

</h1>

//{props.children}

</div>

)

}

export default Greet

**Working with Lists and Conditionals**

**If Condition:**

import React, { Component } from 'react'

class UserGreeting extends Component {

constructor(props) {

super(props)

this.state = {

isLoggedIn: true

}

}

// #if-else approach

// render() {

// if (this.state.isLoggedIn) {

// return <div>Welcome User</div>

// } else {

// return <div>Welcome Guest</div>

// }

// }

// #element-variables approach

// render() {

// let message

// if (this.state.isLoggedIn) {

// message = <div>Welcome User</div>

// } else {

// message = <div>Welcome Guest</div>

// }

// return <div>{message}</div>

// }

// #ternary-operator-approach

// render() {

// return this.state.isLoggedIn ? (

// <div>Welcome User</div>

// ) : (

// <div>Welcome Guest</div>

// )

// }

// #short-circuit-operator-approach

render() {

return this.state.isLoggedIn && <div>Welcome User</div>

}

}

export default UserGreeting

**Switch**

function App() {

const userType = 'Admin';

return (

<div className="container">

<h1>React Switch Case Condition Example </h1>

{(() => {

switch (userType) {

case 'Admin':

return (

<div>You are a Admin.</div>

)

case 'Manager':

return (

<div>You are a Manager.</div>

)

default:

return (

<div>You are a User.</div>

)

}

})()}

</div>

);

}

export default App;

**List**

**Example 1 :**

function NameList() {

const names = ['Sachin', 'Kohli', 'Dhoni', 'Sachin']

return(

<div>

<h2>{names[0]}</h2>

<h2>{names[1]}</h2>

<h2>{names[2]}</h2>

<h2>{names[3]}</h2>

<h2>{names[4]}</h2>

names.map( name => <h2> name </h2>)

</div>)

}

export default NameList

import React from 'react'

import Person from './Person'

function NameList() {

const names = ['Sachin', 'Kohli', 'Dhoni', 'Sachin']

const persons = [

{

id: 1,

name: 'Sachin',

age: 30,

skill: 'React'

},

{

id: 2,

name: 'Kohli',

age: 25,

skill: 'Angular'

},

{

id: 3,

name: 'Dhoni',

age: 28,

skill: 'Vue'

}

]

const personList = persons.map(person => <Person key={person.id} person={person} />)

const nameList = names.map((name, index) => <h2 key={index}>{index} {name}</h2>)

return <div>{nameList}</div>

}

export default NameList

**Person.js**

import React from 'react'

function Person({person, key}) {

return (

<div>

<h2>

I am {person.name}. I am {person.age} years old. I know {person.skill}

</h2>

</div>

)

}

export default Person

**Event Binding**

Just like HTML, React can perform actions based on user events.

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

**Adding Events**

React events are written in camelCase syntax:

onClick instead of onclick.

React event handlers are written inside curly braces:

onClick={functionhandler}  instead of onClick="functionname()".

**Bind this**

For methods in React, the this keyword should represent the component that owns the method. That is why you should use arrow functions. With arrow functions, this will always represent the object that defined the arrow function.

**Example 1:**

import React from 'react'

export default () => {

function clickHandler() {

console.log('Button clicked')

}

return (

<div>

<button onClick={clickHandler}>Click</button>

</div>

)

}

**Example 2:**

import React, { Component } from 'react'

class ClassClick extends Component {

clickHandler() {

console.log('Clicked the button')

}

render() {

return (

<div>

<button onClick={this.clickHandler}>Click Me</button>

</div>

)

}

}

export default ClassClick

**Example 3 (Method Passing as argument):**

import React, { Component } from 'react'

import ChildComponent from './ChildComponent'

class ParentComponent extends Component {

constructor(props) {

super(props)

this.state = {

parentName: 'Parent'

}

this.greetParent = this.greetParent.bind(this)

}

greetParent(childName) {

alert(`Hello ${this.state.parentName} from ${childName}`)

}

render() {

return (

<div>

<ChildComponent greetHandler={this.greetParent} />

</div>

)

}

}

export default ParentComponent

import React from 'react'

const ChildComponent = (props) => {

return (

<div>

<button onClick={() => props.greetHandler('child')}>Greet Parent</button>

</div>

)

}

export default ChildComponent

**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.

class MyForm extends React.Component {

constructor(props) {

super(props);

this.state = { username: '' };

}

myChangeHandler = (event) => {

this.setState({username: event.target.value});

}

render() {

return (

<form>

<h1>Hello {this.state.username}</h1>

<p>Enter your name:</p>

<input

type='text'

onChange={this.myChangeHandler}

/>

</form>

);

}

}