Introduction

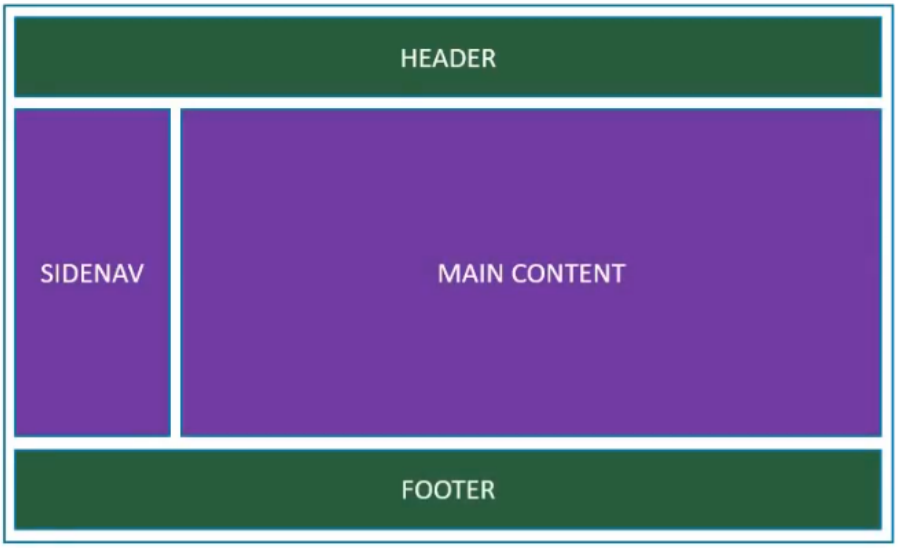
React JS is a frontend library to build rich client side user interfaces. React mainly focus on building user interfaces and its not a framework and its just a library. Because its not having its own http library and routing system.to do these React JS is having a rich eco system which really helpful in building web applications.

What is React

React JS is the project created and maintained by Facebook, in its own product Facebook is using React JS. It is having a huge community of developers and having more than 130k stars on github. React JS is one of the most popular in building rich user interfaces and also React is one of the most in demand skillset of software development.

Component Based Architecture

React way is to develop an application in a component based. Which means the whole application can be divided into small components which can be composed to build the most complex User Interfaces.



Each component can be divided into even small pieces of components. A component is a small piece of UI with some data and which can be reusable in the application any number of times.

React JS works in a declarative paradigm, it means we just need to declare the thing to do with React. Then internally React JS knows how to interact with DOM structure and how to display. React JS is using a virtual DOM and interact with actual DOM. We don't need to change the real DOM by ourselves.

First Application in React JS

https://github.com/facebook/create-react-app

You'll need to have Node 8.16.0 or Node 10.16.0 or later version on your local development machine (but it's not required on the server

To create a new app, you may choose one of the following methods:

npx

```
npx create-react-app my-app
```

(npx comes with npm 5.2+ and higher, see instructions for older npm versions)

npm

```
npm init react-app my-app
```

It will create a directory called my-app inside the current folder.

Inside that directory, it will generate the initial project structure and install the transitive dependencies:

```
my-app
├── README.md
├── node_modules
├── package.json
├── .gitignore
├── public
│   ├── favicon.ico
│   ├── index.html
│   └── manifest.json
└── src
    ├── App.css
    ├── App.js
    ├── App.test.js
    ├── index.css
    ├── index.js
    ├── logo.svg
    └── serviceWorker.js
```

No configuration or complicated folder structures, just the files you need to build your app.
Once the installation is done, you can open your project folder:

```
cd my-app
```

Inside the newly created project, you can run some built-in commands:

`npm start`

Runs the app in development mode.
Open http://localhost:3000 to view it in the browser.

Create-react-app

npx

npx create-react-app <project_name>

npm package runner

npm

npm install create-react-app -g

create-react-app<project_name>

Components

A Component is a small piece of code which represents a portion of User Interface. React JS is having two types of components. Like stateless functional component and state-full Class based component.

Stateless Functional Component

JavaScript Functions

```
function Welcome(props) {
  return <h1>Hello, {props.name}</h1>;
}
```

Stateful Class Component

Class extending Component class

Render method returning HTML

```
class Welcome extends React.Component {
  render() {
    return <h1>Hello, {this.props.name}</h1>;
  }
}
```

Functional Component

A Functional Component is a normal JavaScript function, which takes an optional parameter called ‘props’ and in turn returns JSX which is actually describes the UI.

```
import React from 'react';

let Welcome = () => {
    return (
        <div>
            <h1>Welcome to Functional Component</h1>
        </div>
    );
};
export default Welcome;
```

Class based Component

A Class based Components are the ES6 classes which takes in ‘props’ and in turn returns a JSX snippet.

Apart from functional component, a class based component maintains a private internal state. This state is like some data which is specific to that component and which will be used to describe the User Interface to that component.

Note: We can also maintain the state information for functional components using the new React Concepts called React Hooks from 16.8 version.

```
import React , {Component} from 'react';

class Greet extends Component{
    render(){
        return(
            <div>
                <h1>Welcome to Class based Component</h1>
            </div>
        );
    }
}

export  default Greet;
```

Functional Component VS Class based Components

Functional	Class
Simple functions	More feature rich
Use Func components as much as possible	Maintain their own private data - state
Absence of ‘this’ keyword	Complex UI logic
Solution without using state	Provide lifecycle hooks
Mainly responsible for the UI	Stateful/ Smart/ Container
Stateless/ Dumb/ Presentational	

JSX

JavaScript XML (JSX) this is an extension to JavaScript language syntax. In this we will write and XML-like code for elements and components.

In JSX we will write the tags , attributes and its children.

JSX is not the only option to make React apps, but this is more simpler and elegant way to write the React JS Components Code.

JSX transpiler converts the code to pure JavaScript and which is understood by the browsers.

Without JSX:

```
return React.createElement(
    'div',
    {id : 'abc',
    className : 'test'} ,
    React.createElement(
        'h1' ,
        {id : 'heading'} ,
        'Welcome without JSX'));
```

With JSX:

```
return(
    <div id="abc" className="test">
        <h1 id="heading">Welcome to JSX</h1>
    </div>
);
```

Props in Functional Components

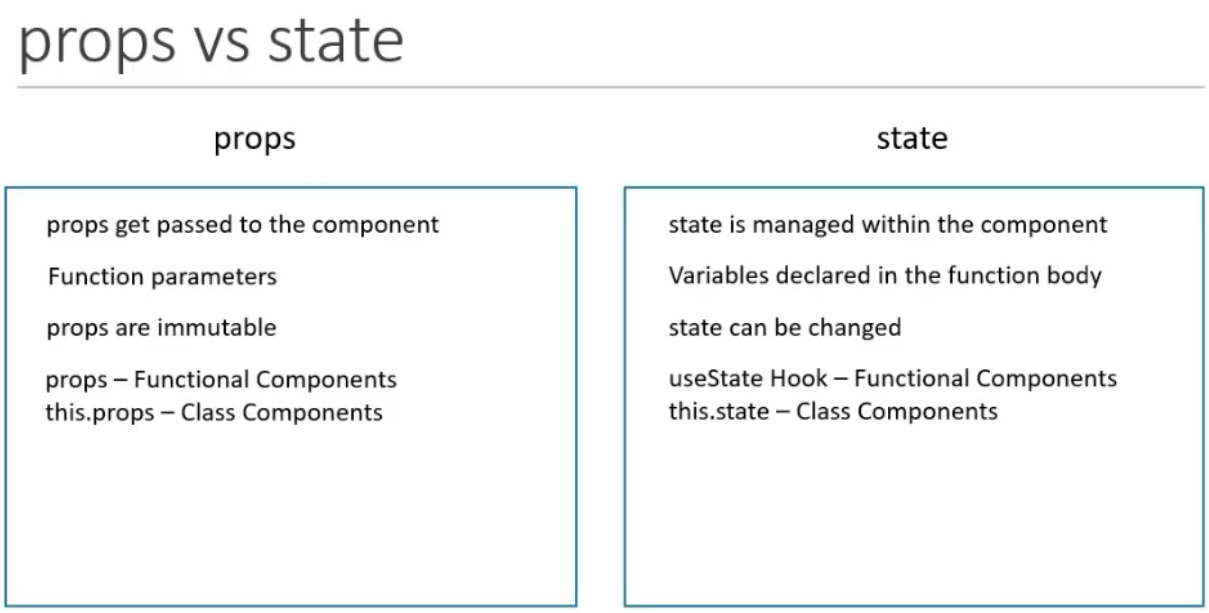
Each functional Component will only display a static data when it renders the UI. In order to display the dynamic data for each functional and also class component, we need to pass ‘props’ to that component as follows,

```
<div className="App">
  <Welcome name='John'  age='40'/>
  <Welcome name='Ramesh' age='45'/>
  <Welcome name='Mahesh' age='35'/>
</div>
```

```
let Welcome = (props) => {
    return (
        <div>
            <h1>Hello {props.name} You are {props.age} Yrs Old</h1>
        </div>
    );
};
```

```
<div className="App">
  <Greet name='John'  age='40'/>
  <Greet name='Ramesh' age='45'/>
  <Greet name='Mahesh' age='35'/>
</div>
```

```
class Greet extends Component{
    render(){
        return(
            <div>
                <h1>Welcome {this.props.name} You are {this.props.age} Yrs Old!</h1>
            </div>
        );
    }
}
```



State in Class based component

We can maintain the application state using this state object in class based component. This state can be modified when an event occurs in the component.

```
constructor(props){
    super(props);
    this.state = {
        message : 'Welcome Visitor'
    };
}

changeMessage(){
    this.setState({
        message : 'Thank You!'
    });
};
```

```
render(){
    return(
        <div>
            <h1>{this.state.message}</h1>
            <button onClick={() => this.changeMessage()}>Subscribe</button>
        </div>
    );
}
```

Event Handling in React JS

We can handle the JavaScript events in React JS in a different approach.

If we are planning to just apply a JavaScript event and not modifying the state, then we can go for the following approach.

In Functional Component:

```
let FunctionClick = () => {
    let clickHandler = () => {
        console.log('Button is Clicked');
    };
    return (
        <div>
            <button onClick={clickHandler}>Click Fn Button</button>
        </div>
    );
};
```

In Class Based Component

```
class ClassClick extends Component{

    clickHandler(){
        console.log('Button is Clicked');
    }

    render(){
        return(
            <div>
                <button onClick={this.clickHandler}>Click Class Button</button>
            </div>
        );
    };
}
```

Binding Event Handlers in React JS

We can bind the event handlers in React JS in 4 ways to update the application state as follows.

1) Binding in the Render
2) Arrow Function in Render
3) Binding in the class constructor
4) Class property as Arrow function

Binding in the Render

```
constructor(props){
    super(props);
    this.state = {
        message : 'Hello'
    };
}

clickHandler(){
    this.setState({
        message : 'Good Bye'
    });
}

render(){
    return(
        <div>
            <h1>{this.state.message}</h1>
            <button onClick={this.clickHandler.bind(this)}>Click Me</button>
        </div>
    );
};
```

Arrow Function in the Render

```
constructor(props){
    super(props);
    this.state = {
        message : 'Hello'
    };
}

clickHandler(){
    this.setState({
        message : 'Good Bye'
    });
}

render(){
    return(
        <div>
            <h1>{this.state.message}</h1>
            <button onClick={() => this.clickHandler()}>Click Me</button>
        </div>
    );
};
```

Binding in the Class Constructor

```
constructor(props){
    super(props);
    this.state = {
        message : 'Hello'
    };
    this.clickHandler = this.clickHandler.bind(this);
}

clickHandler(){
    this.setState({
        message : 'Good Bye'
    });
}

render(){
    return(
        <div>
            <h1>{this.state.message}</h1>
            <button onClick={this.clickHandler}>Click Me</button>
        </div>
    );
};
```

Class Property as Arrow Function

```
constructor(props){
    super(props);
    this.state = {
        message : 'Hello'
    };
}

clickHandler = () => {
    this.setState({
        message : 'Good Bye'
    });
};

render(){
    return(
        <div>
            <h1>{this.state.message}</h1>
            <button onClick={this.clickHandler}>Click Me</button>
        </div>
    );
};
```

Note: 1, 2 approaches are having some performance implications, better use approach 3 and 4.

Methods as props in React JS

As of now we saw how to call the methods in the process of event handling for JavaScript events. And also we saw how to pass information from parent components to child components. Now let's see how to pass data from child component to parent component.

```
class ParentComponent extends Component{
    constructor(props){
        super(props);
        this.state = {
            message : 'Iam from Parent Component'
        };
        this.greetParent = this.greetParent.bind(this);
    }

    greetParent(childName){
        alert(`${this.state.message} ${childName}`);
    }

render(){
     return(
         <div>
             <ChildComponent greenHandler={this.greetParent}/>
         </div>
     );
}

let ChildComponent = (props) => {
    return(
        <div>
            <button onClick={() => props.greenHandler('Child')}>Click Me</button>
        </div>
    );
};
```

Conditional Rendering in React JS

In this we can render various types of snippets depend on a given condition.

```
class UserGreeting extends Component {
    constructor(props){
        super(props);
        this.state = {
            isLoggedIn : false
        };
    }
```

```
render(){
    if(this.state.isLoggedIn){
        return(
            <div>
                <h2>Welcome Naveen</h2>
            </div>
        );
    }
    else{
        return(
            <div>
                <h2>Welcome Guest</h2>
            </div>
        );
    }
};
```

Form Handling in React JS

While handling the form data in React JS, we need to maintain the form data in the state of that component. All form fields need to read the value from the state and any change happens to form fields then we have to manually update the state of that respective fields in the state. This is also called controlled components in React JS.

Let's see how to handle the form data in React JS.

```
class Form extends Component{
    constructor(props){
        super(props);
        this.state = {
            username : '',
            password : ''
        };

    }
    inputHandler = (event) => {
        this.setState({
            [event.target.name] : event.target.value
        });
    };

submitForm = (event) => {
    alert(`${this.state.username} ${this.state.password}`);
    event.preventDefault();
};
```

```
render(){
    return(
        <div>
            <form onSubmit={this.submitForm} method='post'>
                <div>
                    <label>User Name</label>
                    <input type='text'
                           name='username'
                           value={this.state.username}
                           onChange={this.inputHandler}/>
                </div>

                <div>
                    <label>Password</label>
                    <input type='password'
                           name='password'
                           value={this.state.password}
                           onChange={this.inputHandler}/>
                </div>
                <div>
                    <input type='submit' value='Submit'/>
                </div>
            </form>
        </div>
    );
}
```

Lifecycle Methods

Mounting	When an instance of a component is being created and inserted into the DOM
Updating	When a component is being re-rendered as a result of changes to either its props or state
Unmounting	When a component is being removed from the DOM
Error Handling	When there is an error during rendering, in a lifecycle method, or in the constructor of any child component

Lifecycle Methods

Mounting	*constructor, static getDerivedStateFromProps, render* and *componentDidMount*
Updating	*static getDerivedStateFromProps, shouldComponentUpdate, render, getSnapshotBeforeUpdate* and *componentDidUpdate*
Unmounting	*componentWillUnmount*
Error Handling	*static getDerivedStateFromError* and *componentDidCatch*

Mounting Lifecycle Methods

constructor(props)

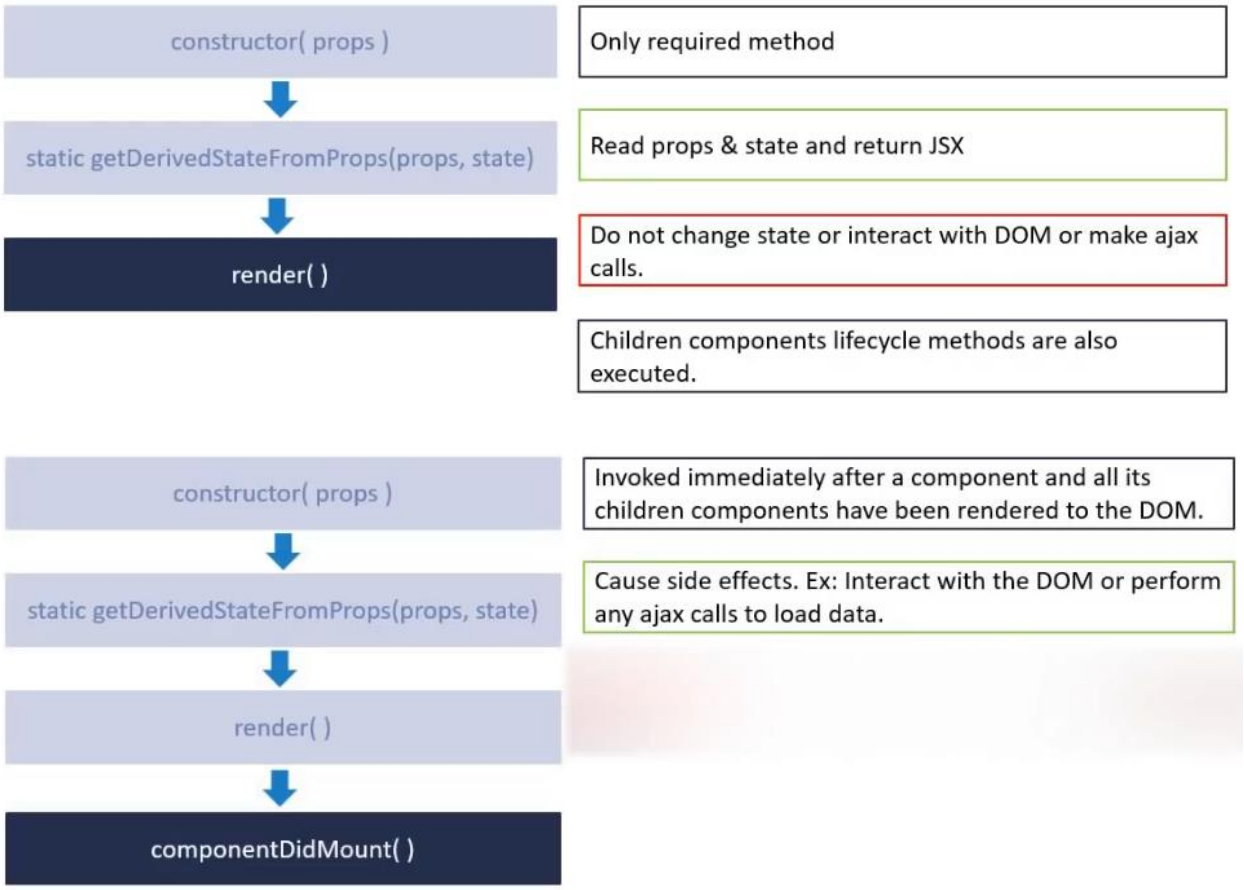
- A special function that will get called whenever a new component is created.
- Initializing state
  Binding the event handlers
- Do not cause side effects. Ex: HTTP requests
- super(props)
  Directly overwrite this.state

constructor(props)

↓

static getDerivedStateFromProps(props, state)

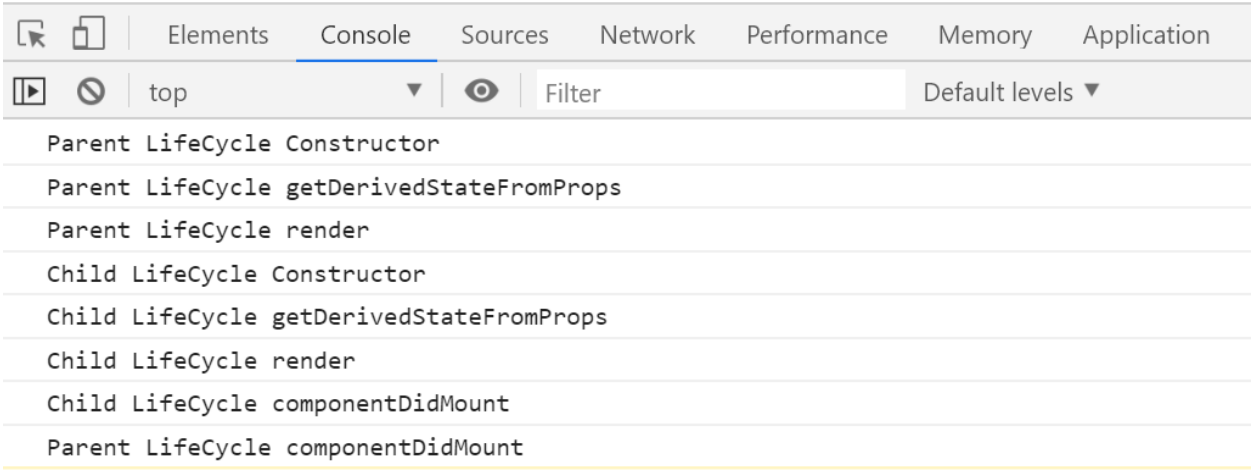
- When the state of the component depends on changes in props over time.
- Set the state
- Do not cause side effects. Ex: HTTP requests




```
class ParentLifeCycle extends Component{

    constructor(props){
        super(props);
        this.state = {
            name : 'Naveen'
        };
        console.log('Parent LifeCycle Constructor');
    }

    static getDerivedStateFromProps(props,state){
        console.log('Parent LifeCycle getDerivedStateFromProps');
        return null;
    }
```

```
    componentDidMount(){
        console.log('Parent LifeCycle componentDidMount');
    }

    render(){
        console.log('Parent LifeCycle render');
        return(
            <div>
                <h2>Parent Life Cycle</h2>
                <ChildLifeCycle/>
            </div>
        );
    };
}

class ChildLifeCycle extends Component{

    constructor(props){
        super(props);
        this.state = {
            name : 'Naveen'
        };
        console.log('Child LifeCycle Constructor');
    }

    static getDerivedStateFromProps(props,state){
        console.log('Child LifeCycle getDerivedStateFromProps');
        return null;
    }

   componentDidMount(){
       console.log('Child LifeCycle componentDidMount');
   }

   render(){
       console.log('Child LifeCycle render');
       return(
           <div>
               <h2>Child Life Cycle</h2>
           </div>
       );
   };
```

OUTPUT


Elements
Console
Sources
Network
Performance
Memory
Application
top
Filter
Default levels
Parent LifeCycle Constructor
Parent LifeCycle getDerivedStateFromProps
Parent LifeCycle render
Child LifeCycle Constructor
Child LifeCycle getDerivedStateFromProps
Child LifeCycle render
Child LifeCycle componentDidMount
Parent LifeCycle componentDidMount