Introduction

Every front-end developer and web developer knows how frustrating and painful is to write the same code at multiple places. If developer needs to add a same button on multiple pages they are forced to do a lot of code. Developers using other frameworks face the same. Developers wanted a framework or library which allow them to break down complex components and reuse the components to complete their projects faster

What is react?

React is the most popular open source JavaScript library for building user interfaces.

It is declarative**,** efficient**,** fast, and flexible library and it also has a strong online community support.

It lets you compose complex UIs from small and independent pieces of code called "components".

It was initially developed and now maintained by Facebook.

React is used to build **single-page applications (S.P.A).**

React is 'V' means **v**iew in MVC(model view controller).

It designs simple views (UI) for each state(screen/web page) in your application, and React will efficiently update and render just the right component when your data changes.

Example

Let's say one of your friends posted a photograph on Facebook. Now you logged in and like the photograph and then you started checking out the comments too.

Now while you are browsing over comments you notice that the likes count has increased by 100, since you liked the picture, even without reloading the page.

This count change is because of Reactjs.

Why we use ReactJS?

We use ReactJS is to develop User Interfaces (UI) that improves the speed of the apps.
It uses virtual DOM (JavaScript object), which improves the performance of the app.
The ReactJS virtual DOM is faster than the regular DOM.
We can use ReactJS on the client-side with other frameworks.
It uses component and data patterns that improve developer's ability to understand and change code which leads easy maintenance of larger apps.

Features of React.js:

- **JSX:** JSX stands for **JavaScript XML. XML means extensible markup language**. **JSX allows us to write HTML tags in React without quotes**. JSX makes it easier to write and add HTML in React. It is faster than normal JavaScript as it performs optimizations while translating to regular JavaScript.
- It makes it easier for us to create templates in React. templates are sets of ready-to-use code built using React technology for the development of dynamic user interfaces.

- **Virtual DOM:** Virtual DOM exists which is like a lightweight copy of the actual DOM. So for every object that exists in the original DOM, there is an object for that in React Virtual DOM.
- It is exactly the same, but it does not have the power to directly change the layout of the document.
- Manipulating original DOM is slow, but manipulating Virtual DOM is fast as nothing gets drawn on the screen.
- **One-way Data Binding:** i.e. the data is transferred from top to bottom i.e. from parent components to child components. The properties (props) in the child component cannot return the data to its parent component but it can have communication with the parent components to modify the states according to the provided inputs. This is the working process of one-way data binding. This keeps everything modular and fast.
- **Component:** React.js divides the web page into multiple components as it is component-based. Each component is a part of the UI design which has **its own logic and design**. So the component logic which is written in JavaScript makes it easy and run faster and can be reusable.
- Components make the task of building UIs much easier.
- You broken down UI into multiple individual pieces called components and work on them independently and merge them all in a parent component which will be your final UI.
- **Performance:** ReactJS performance is better than other frameworks out there today. Because of virtual DOM. The DOM exists entirely in memory. Due to this, when we create a component, we did not insert component directly to the DOM. Instead, we are insert component virtual components that will turn into the DOM leading to smoother and faster performance.

Advantages:

1. **Component-Based Architecture**: ReactJS promotes a component-based architecture, which makes it easier to manage and organize the user interface. Components can be reused throughout the application, leading to a more modular and maintainable codebase.
2. **Virtual DOM**: React uses a virtual DOM, a lightweight copy of the actual DOM. When there are changes to the data, React compares the virtual **DOM with the actual DOM and only updates the necessary parts, making the rendering process more efficient and faster**.
3. **Declarative Syntax:** React uses a declarative approach, allowing developers to describe the desired UI state, and the library takes care of updating the DOM accordingly. This approach makes it easier to reason about the application's state and behavior.
4. **Large Community**: React has a massive and active community, leading to a vast ecosystem of third-party libraries, tools, and resources that can be used to enhance and streamline development.
5. **React Native:** React Native, built on top of React, allows developers to build mobile applications for iOS and Android platforms **using the same React codebase**. This enables cross-platform development and code reuse, saving time and effort.

Disadvantages of ReactJS:

1. **Steep Learning Curve**: ReactJS can be challenging for developers who are new to the component-based architecture and the concepts of virtual DOM. It may take some time for beginners to understand the basics and best practices.
2. **JSX Complexity**: React uses JSX (JavaScript XML) to define the components' structure, which can be unfamiliar to developers used to working with plain HTML. Some developers might find the JSX syntax cumbersome initially.
3. **SEO:** ReactJS is not SEO-friendly out of the box. This means that you will need to take additional steps to make your ReactJS application SEO-friendly.
4. **Frequent Updates**: React is an actively maintained library, which means it receives frequent updates and changes. It can lead to compatibility issues with older codebases when migrating to newer versions.
5. **Documentation:** The documentation for ReactJS can be difficult to understand. This is because the documentation is often incomplete or outdated.

What is npm?

- Npm means **node package manager**.
- npm is the world's largest Software Library (Registry)
- npm is also a software Package Manager and Installer
- The registry contains more then 800,000 code packages.
- Open-source developers use npm to share their software.
- Many organizations also use npm to manage private development.
- **npm** is free to use.
- You can download all npm public software packages without any registration or login.
- npm has a **CLI (Command Line interface)** that can be used to download and install software:
- npm is installed with Node.js
- This means that you have to install Node.js to get npm installed on our computer.

Set up Development Environment

Step 1: Download npm from the web site: https://nodejs.org and then Download & Install npm.

Step 2: Setting up react environment. This may very as per the node version you have.

To set up React Boilerplate. We will install the boilerplate globally. Run the below command in your terminal or command prompt to install the React Boilerplate.

npm install -g create-react-app

After running the above command you should have successfully installed boilerplate code (minimum amount code required to create react project):

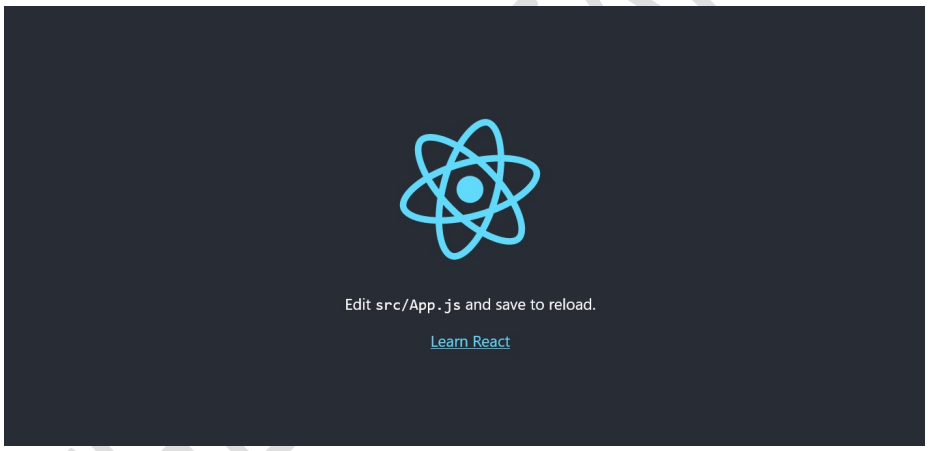
Now open command prompt and open Drive & Directory in where you want to create project and then run the below command to create a new project

npx create-react-app my-app

You can run the project by typing the command cd my-app.

npm start

Now you can view your app in the browser by typing http://localhost:3000. Let us see how it looks.



Each react project has a number of files and folders.

The main files we will be working on within the basic course are ***public/index.html*** and ***src/index.js***.

The index.html file has div element with id = "root", inside which everything will be rendered and all of our React code will be inside the index.js file.

That's all! We have a development environment set up and ready. Now we will move ahead to start learning ReactJS development to make some use of it.

ReactJS | Introduction to JSX

Lesson 1:

```
import React from 'react';
import ReactDOM from 'react-dom/client';
import './index.css';
var root = ReactDOM.createRoot(document.getElementById('root'));
var output = <div><h1>Another Example</h1><hr/><p>this is body</p></div>;
root.render(output);
```

- JSX stands for JavaScript XML.
- As you can see in the first example, JSX allows us to write HTML directly within the JavaScript code.
- JSX is an extension of the JavaScript language based on ES6, and is translated into regular JavaScript at runtime.
- JSX allows us to write HTML in React.
- JSX makes it easier to write and add HTML in React.
- It is faster than normal JavaScript as it performs optimizations while translating to regular JavaScript.
- It makes it easier for us to create templates.
- **Instead of separating the markup and logic in different files, React uses *components* for this purpose.**

In React we are allowed to use normal JavaScript expressions with JSX.

Characteristics of JSX:

- JSX is not compulsory to use there are other ways to achieve the same thing but using JSX makes it easier to develop react application.
- JSX allows writing expression in { }. The expression can be any JS expression or React variable.
- **To insert a large block of HTML inside a parenthesis i.e, ().**
- JSX produces react elements (group of html tag).
- JSX follows XML rule.
- After compilation, JSX expressions become regular JavaScript function calls.
- JSX uses camelcase notation for naming HTML attributes. For example, tabindex in HTML is used as tabIndex in JSX.

Advantages of JSX:

- JSX makes it easier to write or add HTML in React.
- JSX can easily convert HTML tags into react elements.
- It is faster than regular JavaScript.
- JSX allows us to put HTML elements in DOM without using appendChild() or createElement() method.
- As JSX is an expression, we can use it inside of **if statements and for loops, store into variables, accept it as arguments, or return it from functions**.
- **JSX prevents XSS (cross-site-scripting) attacks popularly known as injection attacks.**

- It is type-safe, and most of the errors can be found at compilation time.

Disadvantages of JSX:

- **JSX throws an error if the HTML is not correct.** If HTML elements are not properly closed JSX will give an error.
- In JSX HTML code must be wrapped in one top-level element otherwise it will give an error.

Lesson 2

```
import React from 'react';
import ReactDOM from 'react-dom/client';
import './index.css';
const root = ReactDOM.createRoot(document.getElementById('root'));
const output = React.createElement('h1',{},"html element created without using JSX")
root.render(output);
```

How to write 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:

Lesson 3

```
import React from 'react';
import ReactDOM from 'react-dom/client';
import './index.css';
const root = ReactDOM.createRoot(document.getElementById('root'));
let num1 = 20;
let num2 = 3;
var output = <div><h1>Expression in JSX</h1> Addition = {num1 + num2} <br/> Subtraction = {num1 -
num2} <br/></div>;
root.render(output);
```

How to insert a Large Block of HTML code in JSX

To write large block of HTML we use multiple lines but put the HTML inside **parentheses()**

lesson - 4

```
import React from 'react';
import ReactDOM from 'react-dom/client';
import './index.css';
var root = ReactDOM.createRoot(document.getElementById('root'));
var page = (
  <div>
    <h1>List of Countries in Asia</h1><hr/>
    <ul>
      <li>India</li>
      <li>Sri-lanka</li>
      <li>Bhutan</li>
      <li>Nepal</li>
      <li>Bangladesh</li>
      <li>China</li>
    </ul>
  </div>
)
root.render(page);
```

Attribute class must be used as className

The `class` attribute is used to apply class style in HTML, but because JSX is rendered as JavaScript, and the `class` keyword is a reserved word in JavaScript, you are not allowed to use it in JSX instead of class **className** is used.

How to create form inside JSX using bootstrap? (How to use bootstrap?)

```
import React from 'react';
import ReactDOM from 'react-dom/client';
const root = ReactDOM.createRoot(document.getElementById('root'));
const page = (
<div className='container'>
  <div className='row'>
    <div className='col-12'>
      <div className='card'>
        <div className='card-header text-bg-danger'>
          <h3>Login Form</h3>
        </div>
        <div className='card-body'>
          <form>
            <div className='mb-3'>
              <input type="email" className="form-control" placeholder="Email Address" />
            </div>
            <div className='mb-3'>
              <input type="Password" className="form-control" placeholder="Password" />
            </div>
            <div className='d-flex justify-content-end'>
              <input type="button" class="btn btn-danger" value="Login" />  
              <input type="reset" class="btn btn-warning" value="reset" />
            </div>
          </form>
        </div>
      </div>
    </div>
  </div>
</div>
);
root.render(page);
```

Comments in JSX: JSX allows us to use comments as it allows us to use JavaScript expressions. Comments in JSX begins with **/*** and ends with ***/**. These comments must be given in curly braces {} just like we did in the case of expressions.

HOW TO CREATE & CALL USER DEFINED FUNCTION FROM JSX?

We can call function (with or without argument) from JSX which can return value. If we call function from render method then it must return something.

```
import React from 'react';
import ReactDOM from 'react-dom/client';
const root = ReactDOM.createRoot(document.getElementById('root'));
function Greeting(user) // creating function that return expression
{
    return <div>Hello Mr/miss/mrs {user.name} {user.surname}</div>
}
//creating object
let user = {
  name : "ankit",
  surname : "patel"
};
root.render(Greeting(user)); // calling function
```

How to use JSX with decision making?

```
import React from 'react';
import ReactDOM from 'react-dom/client';
const root = ReactDOM.createRoot(document.getElementById('root'));
function Greeting(user) // creating function that return expression
{
    if (user !== undefined) //check whether variable exist or not
      return <p>Hello Good Morning Mr/MISS/Mrs {user.name} {user.surname}</p>;
    else
      return <p>Hello Good Morning Mr/MISS/Mrs unknown</p>;
}
let user = {
  name : "ankit",
  surname : "patel"
};
root.render(Greeting(user)); // calling function
```

You can't use loop because you can't use JS code within JSX without enclosing them in `{}`. The curly braces are used to run an expression. The expression could be one line calculation, a method call, or a ternary condition. But you can't have variable declaration or loops or any other complex stuff in them. It has to be done differently. We will see later on how to do it.

How to specify attributes with JSX?

You may use single or double quotes to specify string literals as attributes:
You may also use curly braces to embed a JavaScript expression in an attribute:

What is Babel?

Babel is a free and open-source JavaScript Transcompiler that is mainly used to convert ES6 (2015) and above code into a backward compatible code that can be run by older JS engines.

It allows us to use future JavaScript in today's browsers. In simple words, it can convert the latest version of JavaScript code into the one that the browser understands.

The latest standard version which JavaScript follows is ES2020 which is not fully supported by all the browsers and hence we make use of a tool such as 'babel' so that we can convert it into the code that today's browser understands.

JSX was created as an extension to ECMAScript with the purpose of designing a more concise and easy-to-understand syntax for building DOM tree structures and attributes.

With JSX, it is easy to define UI components in React.

JSX should not be implemented directly by browsers, **but instead requires a compiler to transform it into ECMAScript. This is what Babel actually does.**

Babel acts as this compiler allowing us to take advantage all the latest features of JSX while building react components.

How to update the Rendered Element (already displayed)

React elements are immutable means part of it can't be changed. Once you create an element, you can't change its children or attributes. the only way to update the UI is to create a new element, and pass it to `root.render()`. Let us see example

```
import React from 'react';
import ReactDOM from 'react-dom/client';
const root = ReactDOM.createRoot(document.getElementById('root'));
function getDateTime() {
   var now = new Date(); //Date is library class in javascript
   var output = (
     <div>
       <h1>Current Time</h1>
       <a className='btn btn-danger'>{now.toLocaleTimeString()}.</a>
       <h1>Current Date</h1>
       <a className='btn btn-info'>{now.getDate()}-{now.getMonth()+1}-{now.getFullYear()}</a>
     </div>
   );
   root.render(output);
}
setInterval(getDateTime, 1000); //setInterval is JS function called function passed as first
argument at given interval in 2nd argument.
```

React only updates what is Necessary

React DOM compares the element and its children to the previous one, and only applies the DOM updates necessary to bring the DOM to the desired state.

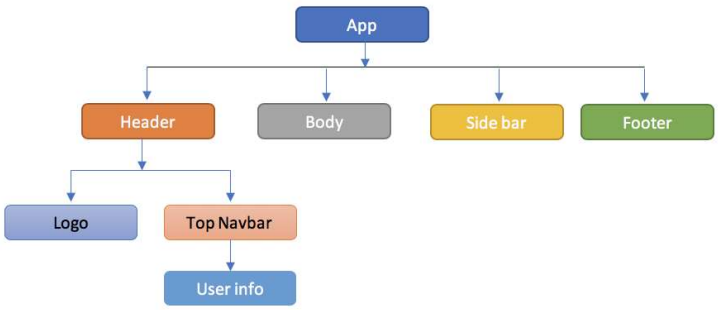
Even though we create an element describing the whole UI tree on every function call, only the text node whose contents have changed gets updated by React DOM.

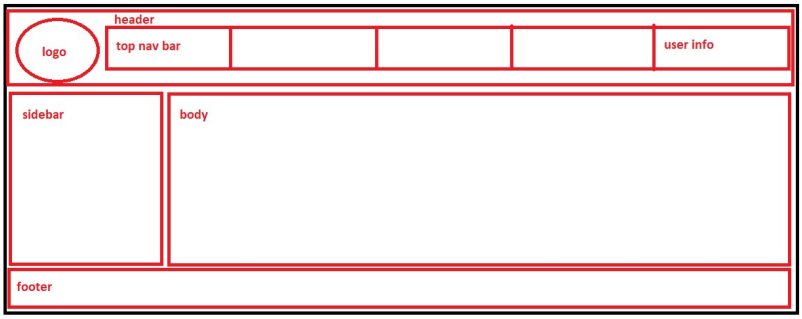
React Components (parts)

Before React was developed by facebook, the developers were writing more than thousands of lines of code for developing a single page application. These applications have the traditional DOM structure, and making changes in them was a very challenging task. If any mistake found, it manually searched in the entire application and solved accordingly. Therefore the component-based approach was introduced in react to solve this problem. In this technique**, the entire application is divided into a small logical group of code, which is known as components.**

A Component is the basic building blocks of a React application. It makes the task of building UIs much easier. **Each component exists in the same space, but they work independently from one another** and merge all in a component, which will be the final UI of your application.

Every React component has it's own structure parent, methods as well as APIs. They can be reusable as per your need. Think of it as the entire UI is an inverted tree. Here, the root is the starting component, and each of the other pieces becomes branches, which are further divided into sub-branches. **And these branches and sub branches are also components**





In ReactJS, we have mainly two types of components.

1. Functional Components
2. Class Components

Functional Components

Functional Components are simple JavaScript function that may or may not receive inputs (parameters) and return some UI (JSX) code. In React, function components don't have their own state. **We can create a function that takes props (properties) as input and returns what should be rendered means displayed**. The functional component is also known as a stateless component because they do not hold or manage state means it cannot keep data over multiple calls of the same functions. Let us see example.

```
import React from 'react';
import ReactDOM from 'react-dom/client';
function Page() {
  return (
    <div className='container'>
      <div className='row'>
        <div className='col-12'>
            <h1>Example of component</h1> <hr/>
            <p>this is an example of page</p>
        </div>
      </div>
    </div>
  );
}
const root = ReactDOM.createRoot(document.getElementById('root'));
root.render(<Page />);
```

It is possible that we have multiple function components and we can use one functional component inside other functional components. Let us see an example

```
import React from 'react';
import ReactDOM from 'react-dom/client';
function Header() //first letter of component should be capital
{
  return (
    <div><h1>Example of component</h1> <hr/></div>
  );
}
function Footer(){
  return(
    <div><p className='text-muted text-center'>this is footer section</p></div>
  )
}
function Page() {
  return (
    <div className='container'>
      <div className='row'>
        <div className='col-12'>
            <Header />
            <p>this is an example of page</p>
            <Footer />
        </div>
      </div>
    </div>
  );
}
const root = ReactDOM.createRoot(document.getElementById('root'));
root.render(<Page />);
```

How to pass value into functional components?

It is possible to pass value in functional components. **Such values are called props.** Using props we can pass as many values as we want. In this way we can create dynamic and reusable functional components let see an example.

Please remember that: - props are ready only.

A component cannot update its own props unless they are arrays or objects.

```
import React from 'react';
import ReactDOM from 'react-dom/client';
function Student(props)
{
//props.name = 'ankit patel' //will stop react app because props are read only

  return (<div className='container'>
      <div className='row'>
        <div className='col-12'>
            <p>
                Student Name: - {props.name} <br/>
                Student Email: - {props.email} <br/>
                Student Birth Date: - {props.dob} <br/>
            </p>
        </div>
      </div>
   </div>)
}
const root = ReactDOM.createRoot(document.getElementById('root'));
root.render(<Student name='ankit' email='ankit@gmail.com' dob='12-july-1985' />);
```

Class Components

Class components is advance version of functional components. **It requires you to create class & inherit/extends React Component and creation a render function which returns a React element.** You can pass data from one class to other class components. Class component are Stateful components. Let us see example

```
import React from 'react';
import ReactDOM from 'react-dom/client';
class Page extends React.Component {
  render() {
    return(<div className='container'>
      <div className=’row’>
       <div className=’col-12’>
              <h2>Class Component example</h2>
              <p>this is some text inside paragraph</p>
       </div>
      </div>
  </div>);
  }
}
const root = ReactDOM.createRoot(document.getElementById('root'));
root.render(<Page />);
```

Component Constructor

If there is a `constructor()` function in your component class, then it will be called automatically when the component gets initiated (object gets created). The constructor function is used to initialize the component's properties. In React, the constructor function must call parent class constructor using `super()` statement otherwise render method will not execute. super() executes the parent component's constructor function, so that you can access all the functions of the parent component (`React.Component`).

State

A State is an special object created inside the constructor method of a class. It's name must be state. We can store any number of key value pair in this object. It is used for internal communication inside a component. It allows you to create components that are interactive and reusable. **If you change state object then render method will be automatically called to update UI of the app.** It is mutable and can be changed by using setState() method.

```
import React from 'react';
import ReactDOM from 'react-dom/client';
class Page extends React.Component {
  constructor(props) {  //component constructor
  super(props); //caling parent class constructor
  this.state = {
    name: 'the easylearn academy',
    contact : '9662512857'
  }
}
 render() {
   return (<div className='container'>
      <div className='row'>
        <div className='col-12'>
            <h2>Example of state </h2> <hr/>
            <p>
                Class Name :- {this.state.name} <br/>
                Contact No  :- {this.state.contact}
            </p>
        </div>
      </div>
   </div>)
 }
}
const root = ReactDOM.createRoot(document.getElementById('root'));
root.render(<Page />);
```

How to pass value into component constructor?

```
import React from 'react';
import ReactDOM from 'react-dom/client';
class Book extends React.Component {
  constructor(props)
  {
    super(props); //required
    this.state = {
      name : this.props.name,
      price : this.props.price,
      pages : this.props.pages
    }
  }
  render() {
    return (
      <div className='container'>
        <div className='row'>
          <div className='col-lg-3'>
            <div className='card'>
                <div className='card-body'>
                  <p>
                    Name {this.state.name} <br/>
                    Price {this.state.price} <br/>
                    Pages {this.state.pages} <br/>
                  </p>
                </div>
            </div>
          </div>
        </div>
      </div>
    );
  }
}
const root = ReactDOM.createRoot(document.getElementById('root'));
root.render(<Book name='learn React' price='100' pages='249' />);
```

How to store components in the separate files?

In React we can easily reuse existing code. Making code reusable is very necessary to avoid duplication of code and make update fast and easy in app. We can make reusable components by writing component code into separate file.

To do that, create a new file with a `.js` file extension in src directory of your project directory and put the code inside it:

Teacher.js

```
import React from 'react';
class Teacher extends React.Component {
  constructor(props)
  {
    super(props); //calling parent class constructor (required) if we don't do we will get error
    //create state object
    this.state = {
        name: this.props.name,
        subject: this.props.subject,
    }
  }
  render() {
    return (
        <div className='container'>
            <div className='row'>
                <div className='col-12'>
                    <h3>Teacher Name : {this.state.name}</h3> <hr/>
                    <b>Subject</b> : {this.state.subject}
                </div>
            </div>
        </div>
    );
  }
}
export default Teacher; //this is required.
```

Index.js

```
import React from 'react';
import ReactDOM from 'react-dom/client';
import Teacher from './Teacher.js'; //must import this file
const root = ReactDOM.createRoot(document.getElementById('root'));
root.render(<Teacher name='Ankit Patel' subject='React js' />),
```

How to change variable's value in state object?

To change a value in the state object, use the `this.setState()` method.

When a value in the `state` object changes, the component will automatically re-render, it simple means that the output will change according to the new value(s).

```
import React from 'react';
import ReactDOM from 'react-dom/client';
class Movie extends React.Component {
  constructor(props) {
    super(props);
    this.state = {
      name: 'KGF',
      photo: "https://www.picsum.photos/300",
      year: 2021,
    };
  }
//arrow function
  ChangeDetail = () =>
  {
    this.setState({
        photo: "https://www.picsum.photos/400",
        year : 2022
      });
  }
  render() {
    return (
      <div>
        <h1>Movie Name {this.state.name}</h1> <hr/>
        <img className='img-fluid' src={this.state.photo} />
        <p>
          <b>Year</b>  {this.state.year}
        </p>
        <button
          type="button" onClick={this.ChangeDetail}>Change Detail</button>
      </div>
    );
  }
}
const root = ReactDOM.createRoot(document.getElementById('root'));
root.render(<Movie   />);
```

How to set state from dynamic input (input given in textbox)?

```
import React from 'react';
import ReactDOM from 'react-dom/client';
class Movie extends React.Component {
  constructor(props) {
    super(props);
    this.state = {
      name: 'KGF',
      year: 2021,
    };
  }
  changeName = (event) =>
  {
    this.setState({name : event.target.value});
  }
  changeYear = (event) =>
  {
    this.setState({year : event.target.value});
  }
  render() {
    return (
      <div>
        <h1>Movie Name {this.state.name}</h1> <hr/>
        <p>
          <b>Year</b>  {this.state.year}
        </p>
        <input type='text' id='name' placeholder='movie name' onBlur={this.changeName} /> <br/>
        <input type='number' id='year' placeholder='release year'  onBlur={this.changeYear} /> <br/>
      </div>
    );
  }
}
const root = ReactDOM.createRoot(document.getElementById('root'));
root.render(<Movie />);
```

Lifecycle in react app

What is the lifecycle?

In general, we might think a lifecycle as birth, growth & death. And our React components follow this cycle as well:
Birth means components creation (**mounted** on the DOM),
Growth means components got changed (**updating**) (by updating) and
Death means components are removed (**unmounted** from the DOM).
This is the component lifecycle! Within the lifecycle of a component, there are different phases. Each phase has its own lifecycle methods. A component's lifecycle can be broken down into four parts:

1. Initialization
2. Mounting
3. Updating
4. Unmounting

Lets look at each in detail.

Initialization

In this phase, our component will be setting its state & props. **This is usually done inside a constructor method**

Mounting

Once the initialization phase completes, we enter into the mounting phase. This is when our React component "mounts" on the DOM (it's created and inserted into the DOM). This is when our component renders for the first time. The methods available in this phase are componentWillMount() and componentDidMount().

componentWillMount()

This method is called just before component is mounted into the DOM (or when the render method is called). Then our component gets mounted. It is only called once.

componentDidMount()

This method is called after the component is mounted into the DOM. it is called only once in a lifecycle after the render method is called, this method is called. We can make API calls and update the state with the API response.

```
import React from 'react';
import ReactDOM from 'react-dom/client';
class LifeCycle extends React.Component {
  constructor(props)
  {
     super(props);
     console.log('constructor method is called');
  }
  componentWillMount() {
      console.log('Component will mount!')
   }
  componentDidMount() {
      console.log('Component did mount!')
  }
  render() {
      return (
         <div className='container'>
            <div className='row'>
              <div className='col-12'>
              <h1>LifeCycle Example</h1>  <hr />
              <b>use console to see output</b>
              </div>
            </div>
         </div>
      );
   }
}
const root = ReactDOM.createRoot(document.getElementById('root'));
root.render(<LifeCycle   />);
```

Updating

After the mounting phase, we enter into the update phase. This is time when component's state changes and thus, re-rendering takes place.
The data of the component (its state & props) will update in response to user events such as clicking, typing, etc. This results in the re-rendering of the component. The methods we use here are following

shouldComponentUpdate(nextProp,nextState)

This method determines whether the component should be updated or not. By default, it'll return true. If at some point, if you want to re-render the component on a condition, then shouldComponentUpdate() method should conditionally return true/false.

If for example, you want to only re-render your component when there is a change in prop — this would be when you use this method. It receives arguments ***nextProps*** and ***nextState*** which help us decide whether to re-render by doing a comparison with the current prop value.

componentWillUpdate(nextProp,nextState)

We call this method before re-rendering our component. It is called once after shouldComponentUpdate().If you want to perform a calculation before re-rendering the component & after updating the state and prop, then you would use this method. Like shouldComponentUpdate(), it also receives arguments *nextProps* and *nextState*.

componentDidUpdate(prevProp,prevState)

We call this method after the re-rendering our component. After the updated component gets updated on the DOM, the componentDidUpdate() method executes. This method will receive arguments *prevProps* and *prevState*.

```
import React from 'react';
import ReactDOM from 'react-dom/client';
class LifeCycle extends React.Component {
  constructor(props){
    super(props);
    this.state = {
      time: new Date().toLocaleTimeString(),
    };
  }
  componentWillMount() {
      console.log('Component will mount!')
   }
  componentDidMount() {
      console.log('Component did mount!')
  }
  shouldComponentUpdate(nextProps, nextState) {
    console.log('shouldComponentUpdate() method called....');
    return true;
  }
  componentWillUpdate(nextProps, nextState) {
    console.log('Component will update!');
  }
  componentDidUpdate(prevProps, prevState) {
    console.log('Component did update!')
  }
  UpdateTime = () => {
    this.setState({
      time : new Date().toLocaleTimeString(),
      });
  }
  render() {
      return (
         <div className='container'>
            <div className='row'>
              <div className='col-12'>
              <h1>LifeCycle Example</h1>  <hr />
              <button className='btn btn-primary'>{this.state.time}</button> <br />
              <button className='btn btn-danger' onClick={this.UpdateTime}>update time</button> <br />
              <b>use console to see output</b>
              </div>
```

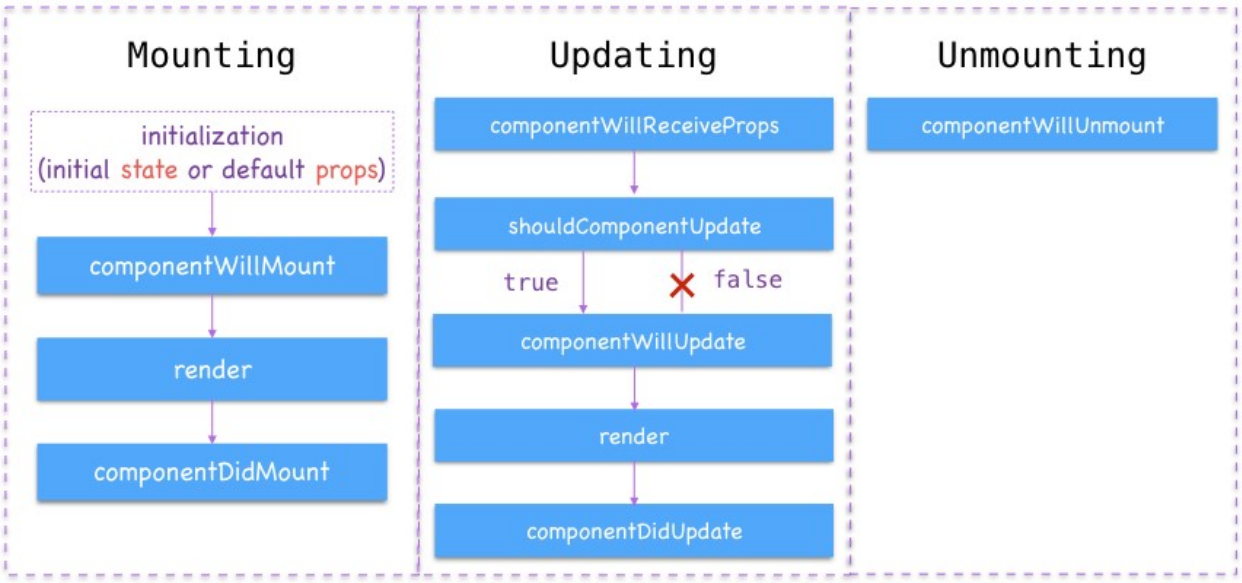
```
            </div>
          </div>
        );
    }
}
const root = ReactDOM.createRoot(document.getElementById('root'));
root.render(<LifeCycle   />);
```

Unmounting

The last phase is unmounting. It has one method ComponentWillUnmount method which is called just before a component is removed from the DOM:

componentWillUnmount()

this method is called automatically before the unmounting takes place. Just before the removal of the component from the DOM, This method is the end of the component's lifecycle!



Let us see one more example

```
import React from 'react';
import ReactDOM from 'react-dom/client';
class Clock extends React.Component {
  constructor(props) {
    super(props);
    this.state = {date: new Date()};
  }
  componentDidMount() {
    this.timerID = setInterval(() => this.tick(),1000);
  }
```

```
  componentWillUnmount() {
    clearInterval(this.timerID);
  }
  tick() {
      this.setState({
          date: new Date()
        });
  }
  render() {
    return (
      <div>
        <h1>Digital Clock</h1>
        <h2>It is {this.state.date.toLocaleTimeString()}.</h2>
      </div>
    );
  }
}
const root = ReactDOM.createRoot(document.getElementById('root'));
root.render(<Clock />);
```

Let's quickly recap what's going on and the order in which the methods are called:

1. When <Clock /> is passed to root.render(), React calls the constructor of the Clock component. Since Clock needs to display the current time, it initializes this.state with an object including the current time. We will later update this state.
2. React then calls the Clock component's render() method. This is how React learns what should be displayed on the screen. React then updates the DOM to match the Clock's render output.
3. When the Clock output is inserted in the DOM, React calls the componentDidMount() lifecycle method. Inside it, the Clock component asks the browser to set up a timer to call the component's tick() method once a second.
4. Every second the browser calls the tick() method. Inside it, the Clock component schedules a UI update by calling setState() with an object containing the current time. Thanks to the setState() call, React knows the state has changed, and calls the render() method again to learn what should be on the screen. This time, this.state.date in the render() method will be different, and so the render output will include the updated time. React updates the DOM accordingly.
5. If the Clock component is ever removed from the DOM, React calls the componentWillUnmount() lifecycle method so the timer is stopped.

State Updates May Be Asynchronous

React may batch multiple setState() calls into a single update for performance. Because this this.props and this.state may be updated asynchronously, **you should not rely on their values for calculating the next state.**

For example, this code **may** fail to update the counter:

```
// Wrong
this.setState({
  counter: this.state.counter + this.props.increment,
});
```

To fix it, use a second form of setState() that accepts a function rather than an object. That function will receive the previous state as the first argument, and the props at the time the update is applied as the second argument:

```
// Correct
this.setState((state, props) => ({
  counter: state.counter + props.increment
}));
```

We used an arrow function above, but it also works with regular functions:

```
// Correct
this.setState(function(state, props) {
  return {
    counter: state.counter + props.increment
  };
});
```

Very important point to remember

Neither parent nor child components can know if a certain component is Stateful or stateless, and they shouldn't care whether it is defined as a function or a class.

This is why state is often called local or encapsulated. **State is not accessible to any component other than the one has created it.**

This is commonly called a "top-down" or "unidirectional" data flow. Any state is always owned by some specific class component, and any data or UI derived from that state can only affect components "below" them in the tree.

Handling Events

Handling events with React elements is very similar to handling events on DOM elements. There are some syntax differences:

- React events are named using camelCase, rather than lowercase.
- With JSX you pass a function as the event handler, rather than a string.

For example, the HTML:

```
<button onclick="activateLasers()"> Activate Lasers</button>
```

is slightly different in React:

```
<button onClick={this.activateLasers}>Activate Lasers</button>
```

Another difference is that you cannot return `false` to prevent default behavior in React. You must call `preventDefault` explicitly. For example, with plain HTML, to prevent the default form behavior of submitting, you can write:

```
<form onsubmit="console.log('You clicked submit.'); return false">
  <button type="submit">Submit</button>
</form>
```

In React, this could instead be:

```
function Form() {
  function handleSubmit(event)
  {
    event.preventDefault();    //form will not be submitted
    console.log('You clicked submit.');
  }

  return (
    <form onSubmit={this.handleSubmit}>
      <button type="submit">Submit</button>
    </form>
  );
}
```

Here, `e` is a synthetic event. React defines these synthetic events according to the W3C (world wide web consitorium) spec, so you don't need to worry about cross-browser compatibility. **React events do not work exactly the same as native events.**

When using React, you generally don't need to call `addEventListener` to add listeners to a DOM element after it is created. Instead, just provide a listener when the element is initially rendered.

```
import React from 'react';
import ReactDOM from 'react-dom/client';
class Toggle extends React.Component {
    constructor(props) {
      super(props); //calling parent class constructor
      this.state = {isToggleOn: true};
      this.handleClick = this.handleClick.bind(this);
    }
    handleClick()
     {
        this.setState((prevState) => ({
          isToggleOn: !prevState.isToggleOn
        }));
      }
```

```
    render() {
      return (
        <button onClick={this.handleClick}>
        {this.state.isToggleOn==true ? 'Like' : 'DisLike'}
        </button>
      );
    }
  }
const root = ReactDOM.createRoot(document.getElementById('root'));
root.render(<Toggle />);
```

OR

```
import React from 'react';
import ReactDOM from 'react-dom/client';
class Toggle extends React.Component {
    constructor(props) {
      super(props);
      this.state = {isToggleOn: true};
    }
    handleClick = () => {
        this.setState(prevState => ({
          isToggleOn: !prevState.isToggleOn
        }));
      }
    render() {
      return (
        <button onClick={this.handleClick}>
        {this.state.isToggleOn ? 'Like' : 'DisLike'}
        </button>
      );
    }
  }
const root = ReactDOM.createRoot(document.getElementById('root'));
root.render(<Toggle />);
```

Passing Arguments to Event Handlers

It is possible to pass one or more argument into event handlers. See following example

```
import React from 'react';
import ReactDOM from 'react-dom/client';
class LoginButton extends React.Component {
  render() {
    // This syntax ensures `this` is bound within handleClick
    return (
    <button onClick={() => this.handleClick(101,'Ankit')}>        Click me
    </button>
    );
  }
  handleClick = (rollno,name) =>
  {
       alert(rollno,name);
  }
}
const root = ReactDOM.createRoot(document.getElementById('root'));
root.render(<LoginButton />);
```

Conditional Rendering

In react, we can conditionally display component(s). means we can check one or more condition, if condition is evaluate as true then we can display one components otherwise we can display other components. This is called conditional rendering. Conditional rendering in React works the same way conditions work in JavaScript. We can use any relational & logical operators to check conditions. Let us see an example.

```
import React from 'react';
import ReactDOM from 'react-dom/client';
import './index.css';
var root = ReactDOM.createRoot(document.getElementById('root'));
class LoginControl extends React.Component
{
  constructor(props)
  {
    super(props);
    this.state = {
      isLoggedIn: false
    }
  }
  updateLogin = () => {
    this.setState({
      isLoggedIn: ! this.state.isLoggedIn
    });
```

```
  }
  render()
  {
    var output; //local variable
    if(this.state.isLoggedIn==false)
    {
      output=<button className='btn btn-primary' onClick={this.updateLogin}>Login</button>
    }
    else
    {
      output=<button className='btn btn-danger' onClick={this.updateLogin}>Logout</button>
    }
    return output;
  }
}

function UserSection()
{
  return (
    <div className='container'>
      <div className='row'>
          <div className='col-12 mb-3'>
              <h1>Conditional Rendering</h1> <hr/>
          </div>
      </div>
      <div className='row'>
          <div className='col-12'>
            <LoginControl />
          </div>
      </div>
    </div>
  )
}
root.render(<UserSection />);
```

Inline If with Logical && Operator

We may embed expressions in JSX by wrapping them in curly braces. This includes the JavaScript logical `&&` operator. It can be handy for conditionally including an element: let us see an example.

```
import React from 'react';
import ReactDOM from 'react-dom/client';
function Mailbox(props) {
  var messages = props.messages;
  return (
    <div>
      <h1>Conditional Rendering</h1> <hr/>
```

```
      {messages.length > 0 &&        <h2>          You have {messages.length} unread messages.
</h2>}
       {messages.length==0 && <h2> you have no new message </h2>}
    </div>
  );
}
var messages = ['hello', 'how are you', 'let us meet to learn coding']; //array/list in js
const root = ReactDOM.createRoot(document.getElementById('root'));
root.render(<Mailbox messages={messages} />);
```

It works because in JavaScript, true && expression always evaluates to expression, and false && expression always evaluates to false.

Therefore, if the condition is true, the element right after && will appear in the output. If it is false, React will ignore and skip it.

Inline If-Else with Conditional Operator

Another method for conditionally rendering elements inline, is to use the JavaScript conditional operator `condition ? true: false`.

```
import React from 'react';
import ReactDOM from 'react-dom/client';
function User(props) {
    var isLoggedIn = props.isLoggedIn;
    return (
      <div>
        The user is <b>{isLoggedIn==1 ? 'currently' : 'not'}</b> logged in.    </div>
    );
}
const root = ReactDOM.createRoot(document.getElementById('root'));
root.render(<User isLoggedIn='1'  />);
```

Lists and Keys

Lists are used to display data in an ordered format and mainly used to display menus on websites. In React, Lists can be created in a similar way as we create lists in JavaScript.

Let us see how we create a list in React. Usually we would render lists inside a component.

To do this, we will use the map() function for traversing the list element, and for updates, we enclosed them between **curly braces {}**. Finally, we assign the array elements to list Items. Now, include this new list inside ** ** elements and render it to the DOM.

```
import React from 'react';
import ReactDOM from 'react-dom/client';
function StateList(props) {
  var states = props.states; //array or list
  //create variable that has all the states inside li tag
  var listItems = states.map((CurrentState) => {
       return <li>{CurrentState}</li>
    }
  );
 //OR
    // var statelist = []; //empty array
    // for(var i=0;i<states.length;i++)
    // {
    //     statelist[i]= <li className='list-group-item'>{states[i]}</li>
    // }

  return (
    <div>
    <h3>Some Indian States</h3> <hr/>
    <ul type='square'>{listItems}</ul>
    </div>
  );
}
//define list/array
var states = ['Gujarat','Maharastra','Karanataka','Andhra-pradesh','Tamil nadu']; //array or list
var root = ReactDOM.createRoot(document.getElementById('root'));
root.render(<StateList states={states} />);
```

Each item in list should be created with keys. So react can easily track which item in list have been updated or removed. Keys should be given to the elements inside the array to give the elements a stable identity: The best way to pick a key is to use a string that uniquely identifies a list item among its siblings. Most often you would use IDs from your data as keys:

let us see an example

```
import React from 'react';
import ReactDOM from 'react-dom/client';
function NumberList(props) {
  var states = props.states; //create local variable state
  //create variable that has all the states inside li tag
  var listItems = states.map((CurrentState) =><li key={CurrentState.id}>{CurrentState.name}</li>
  );

  return (
    <div>
    <h3>Some Indian States</h3> <hr/>
    <ul type='square'>{listItems}</ul>
    </div>
  );
}
var states =
[{name:'Gujarat',id:1},{name:'Maharastra',id:2},{name:'Karanataka',id:3},{name:'Andrha-
pradesh',id:4},{name:'Tamil nadu',id:5}];
var root = ReactDOM.createRoot(document.getElementById('root'));
root.render(<NumberList states={states} />);
```

rule of thumb is that elements inside the `map()` call need keys.

Forms

HTML form elements work a differently from other DOM elements in React, because form elements naturally keep some internal state. There are two ways to handle forms.

Controlled Input Approach / Controlled Components

As the name says, in the controlled component the form input element's values and changes in it are totally driven by event handlers and the value of the input element is always retrieved from the state. For example each form control is attached with one common event called this.onInputChange(). if you need to update a view with the change of an input then using a controlled input will save a lot of time and will ensure a cleaner code. Let us see an example

Multiple input example using Controlled approach

```
import ReactDOM from 'react-dom/client';
import React, { Component } from "react";
class SimpleForm extends Component {
  constructor() {
    super();
    this.state = {}; //empty state object
  }
  onInputchange = (event) => {
    this.setState({
      [event.target.name]: event.target.value
       //this will add key value into state, if key already exist then it's will value will be updated
    });
  }
  onSubmitForm = (event) => {
    event.preventDefault();
    console.log(this.state)
  }

  render() {
    return (
      <form method="post" onSubmit={this.onSubmitForm}>
        <div>
          <label>
            First Name :
            <input name="fname" type="text" value={this.state.fname} onChange={this.onInputchange}
required  />
          </label>
        </div>
        <div>
          <label>
            Last Name :
            <input name="lname" type="text" value={this.state.lname} onChange={this.onInputchange}
required   />
          </label>
        </div>
```

```
        <div>
          <label>
            Email :
            <input name="email" type="email" value={this.state.email} onChange={this.onInputchange}
required   />
          </label>
        </div>
        <div>
            <button type='submit'>Submit</button>
        </div>
      </form>
    );
  }
}
const root = ReactDOM.createRoot(document.getElementById('root'));
root.render(<SimpleForm />);
```

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

After getting values from the form control, it stores the value based on the name of key like this:

```
[event.target.name]: event.target.value;

i.e. fname : value
```

Any modifications the user makes to the initial or an empty value are reflected in the state object of the components.

Once you submit the form, the state object should look like this:

```
{
    email: "test@test.com"
    fname: "Test first name"
    lname: "Test last name"
}
```

Handling forms with the useState Hook

What is usestate?

`useState` is a Hook that allows you to create/change/remove means access state variables in functional components. You pass the initial state to this function and it returns a variable with the current state value (not necessarily the initial state) and another function to update this value.

Whereas the state in a class is always an object, with Hooks, the state can be any type. Each piece of state holds a single value, which can be an object, an array, a Boolean, or any other type you can use.

The useState Hook allows you to declare only one state variable (of any type) at a time, like this:

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

const Message= () => {
   const messageState = useState( '' ); //normal variable which can hold single
value
   const listState = useState( [] ); //array variable
}
```

`useState` takes the initial value of the state variable as an argument.

Now let us see example

```
import ReactDOM from 'react-dom/client';
import React,{ useState } from "react";
function LoginForm() {
  //create state variables using useState hook
  var [email, setEmail] = useState("");
  var [password, setPassword] = useState("");

  const onSubmitForm = (event) => {
    event.preventDefault();
    console.log(email);
    console.log(password);
  }
-------------------------------------------------------------------------------------------
  return (
    <form onSubmit={onSubmitForm}>
      <div>
        <label htmlFor="email">Email</label>
        <input type="email" id="email" value={email} onChange={(e) => setEmail(e.target.value)}
required />
      </div>
      <div>
        <label htmlFor="password">Password</label>
        <input type="password" id="password" value={password} onChange={(e) =>
setPassword(e.target.value)} required
        />
      </div>
      <input type='submit' value='Login' />
```

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

Uncontrolled input approach / uncontrolled components

In this approach, form data is handled by the DOM itself. This approach doesn't use any states on input elements or any event handler**. If you use this approach you cannot get input element's real-time value changes.** In the uncontrolled components, we use Refs to access the values of input elements. Let us see example

```
import ReactDOM from 'react-dom/client';
import React, {Component} from "react";
class UncontrolledComponent extends Component {
  constructor(props) {
    super(props);
    this.email = React.createRef();
    this.password = React.createRef();
  }
  handleSubmit = (event) => {
    console.log('email ' + this.email.current.value);
    console.log('Password ' + this.password.current.value);
    event.preventDefault();
  }
  render() {
    return (
      <form onSubmit={this.handleSubmit}>
        <div>
          <label>Email: <input type="email" ref={this.email} />        </label>
        </div>
        <div>
          <label>Password: <input type="text" ref={this.password} />        </label>
        </div>
        <input type="submit" value="Submit" />
      </form>
    );
  }
}
const root = ReactDOM.createRoot(document.getElementById('root'));
root.render(<UncontrolledComponent />);
```

React Recommend use of controlled components.