

ReactJS

What is React?

- React is a JavaScript library created by Facebook.
- React is a tool for building UI components.

Content

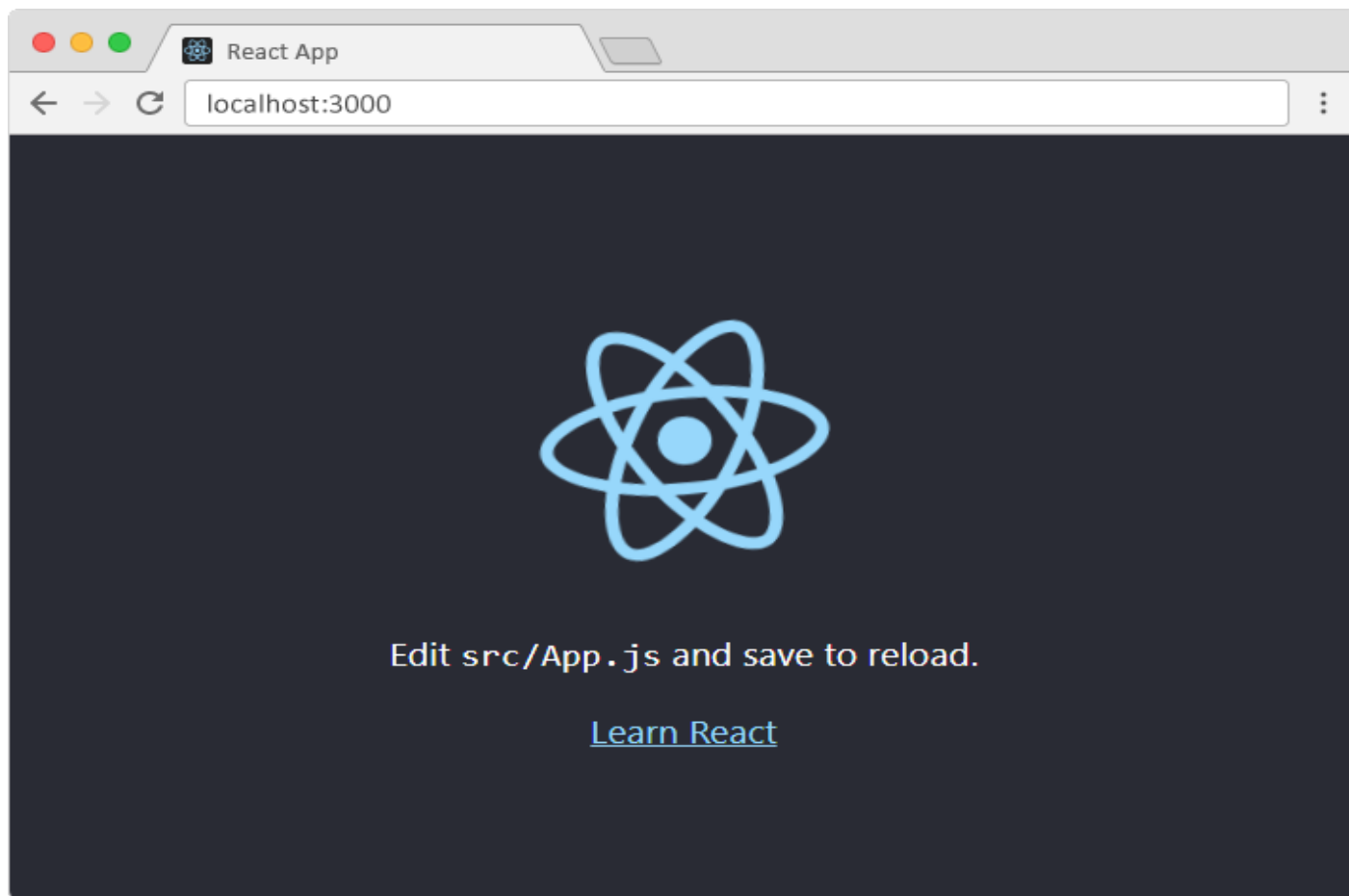
- React Introduction
- Props & State
- State Lifecycle
- Composition
- List & Map
- Lifting Stateup
- Forms
 - Controlled & Uncontrolled Forms

How Does React Work?

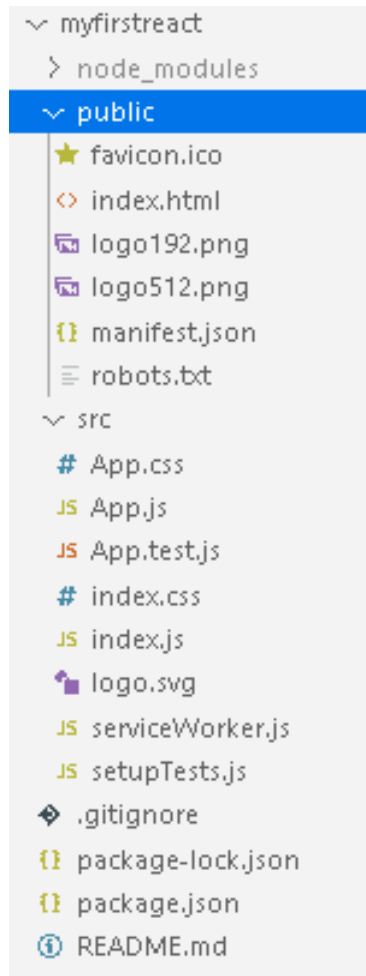
- React creates a VIRTUAL DOM in memory.
 - Instead of manipulating the browser's DOM directly, React creates a virtual DOM in memory, where it does all the necessary manipulating, before making the changes in the browser DOM
- React only changes what needs to be changed!
 - React finds out what changes have been made, and changes **only** what needs to be changed.

React Getting Started Setting up Environment.

- To use React in production, you need **NPM** and [Node.js](#)
 - **Check NPM Installed: `node --version`**
- If you have NPM and Node.js installed, you can create a React application by first installing the create-react-app.
- Install create-react-app by running this command in your terminal:
 - **`npm install -g create-react-app`**
- Run this command to create a React application named reactfirstdemo
 - **`npx create-react-app reactfirstdemo`**
- Now Move to reactfirstdemo folder and run command
 - `npm start`
- Host Application on <http://localhost:3000>.



Folder Structure



package.json – This file contains dependencies and scripts required for the project.

```
{
  "name": "myfirstreact",
  "version": "0.1.0",
  "private": true,
  "dependencies": {
    "@testing-library/jest-dom": "^4.2.4",
    "@testing-library/react": "^9.5.0",
    "@testing-library/user-event": "^7.2.1",
    "react": "^16.13.1",
    "react-dom": "^16.13.1",
    "react-scripts": "3.4.1"
  },
  "scripts": {
    "start": "react-scripts start",
    "build": "react-scripts build",
    "test": "react-scripts test",
    "eject": "react-scripts eject"
  },
  "eslintConfig": {
    "extends": "react-app"
  },
  "browserslist": {
    "production": [
      ">0.2%",
      "not dead",
      "not op_mini all"
    ],
    "development": [
      "last 1 chrome version",
      "last 1 firefox version",
      "last 1 safari version"
    ]
  }
}
```


This file contains all settings for React applications including:

- name - points to name of your app.
- version - refers to the current version that the application is using.
- “private”: true - is a foolproof setting which avoids accidentally publishing of your react app as a public package in npm ecosystem.
- Dependencies will contain all required node modules and versions required for the application. By default, 2 dependencies are added which include React and React-Dom that allow using JavaScript. In our demo, we are using React version 16.13.1.

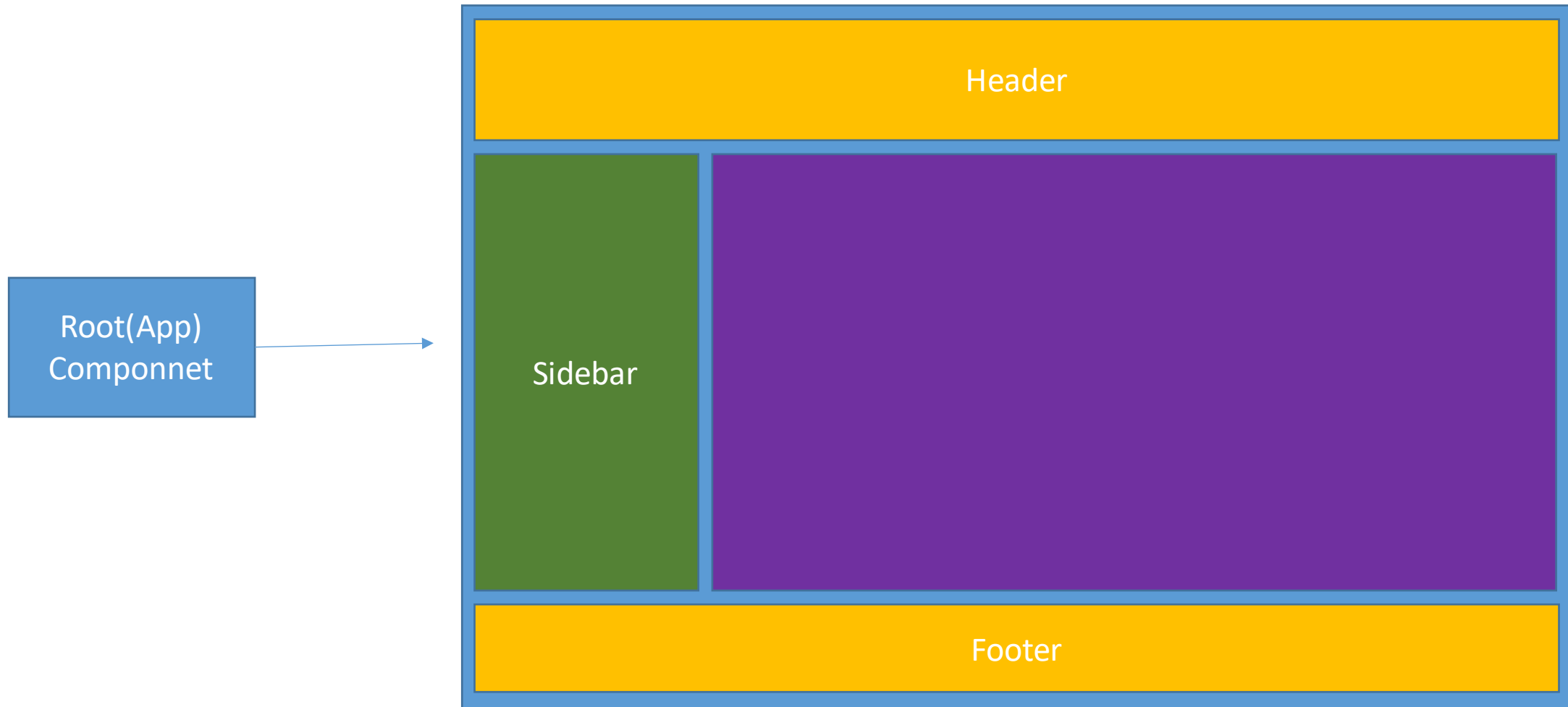
Scripts specify aliases that can be used to access some React command in a more efficient manner.

- package-lock.json contain exact dependency tree to be installed in /node_modules. It helps while a team is working on private apps to ensure that they are working on the same version of dependencies and sub-dependencies. It also maintains a history of changes done in package.json so, that at any point of time, when required previous changes can be looked back in the package-lock.json file.
- node_modules - This folder contains all dependencies and sub-dependencies specified in package.json used by React app. It contains more than 800 subfolders, this folder is automatically added in the .gitignore file.
- public - This folder contains files which don't require additional processing by webpack. The **index.html** file is considered as an entry point for the web application. Here, the favicon is a header icon and manifest.xml file contains configuration when your application is used for Android app.
- src - This folder is the heart of React application as it contains JavaScript which needs to be processed by webpack. In this folder, there is a main component App.js, its related styles (App.css), test suite (App.test.js). index.js, and its style (index.css); which provide an entry point into the App. Lastly, it contains registerServiceWorker.js which takes care of caching and updating files for the end user. It helps in offline capability and faster page loading after the first visit.
- After all this, we add /Component folder in src to add our custom component and its related files and /Views folder for adding React views and its related files.

Components

- Components are the building blocks of any React app and a typical React app will have many of these. Simply put, a component is a JavaScript class or function that optionally accepts inputs i.e. properties(props) and returns a React element that describes how a section of the UI (User Interface) should appear.
- They are re-usable and can be nested inside other component.

Components



Functional components (Stateless)

- These components are purely presentational and are simply represented by a function that optionally takes props and returns a React element to be rendered to the page.
- Generally, it is preferred to use functional components whenever possible because of their predictability and conciseness. Since, they are purely presentational, their output is always the same given the same props.
- You may find functional components referred to as *stateless*, *dumb* or *presentational* in other literature
- **Functional** because they are basically functions
- **Stateless** because they do not hold and/or manage state
- **Presentational** because all they do is output UI elements
- Function name must have first character capital

Functional component syntax

Without Props hello.js component with jsx syntax

```
import React from 'react';
const Greeting = () => <h1>Greeting of the
day!!! </h1>;
export default Greeting;
```

Without Props hello.js component with js syntax

```
import React from 'react';
const Greeting = () =>
{
  return
  React.createElement("div", null, "Greet
ing of the day");}
export default Greeting;
```

App Component

```
import React from 'react';
import './App.css';
import Greeting from './components/hello'
function App() {
  return (
    <div>
    <div>
    <Greeting />
    </div>
    </div>
  );
}
export default App;
```

Functional Component

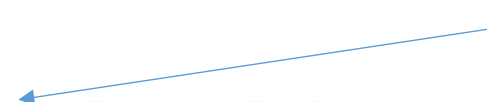
With Props hello.js component

```
import React from 'react';
const Greeting = (props) =>
  <h1>
    Greeting of the day!!! {props.name}
  </h1>;
export default Greeting;
```

Props are read only

```
import React from 'react';
import './App.css';
import Greeting from './components/hello'
function App() {
  return (
    <div>
      <div>
        <Greeting name="Reena" />
      </div>
    </div>
  );
}
export default App;
```

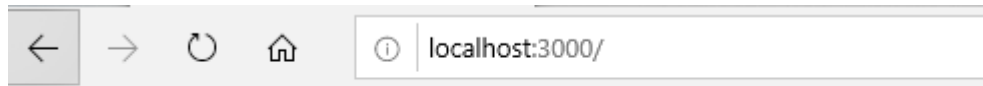
Passing value



output



Greeting of the day!!!



Greeting of the day!!! Reena

Stateful Component

- Class extending component class
- Render method returning HTML
- These components are created using ES6's class syntax. They have some additional features such as the ability to contain logic (for example methods that handle onClick events), local state
- you might find class components referred to as *smart*, *container* or *stateful* components.
- **Class** because they are basically classes
- => **Smart** because they can contain logic
- => **Stateful** because they can hold and/or manage local state
- => **Container** because they usually hold/contain numerous other (mostly functional) components

A class component in its simplest form:

```
import React from 'react';
class Greetingclass extends React.Component
{
  render(){
    return <h1>Greeting of the Day</h1>;
  }
}

export default Greetingclass
```

```
import React from 'react';
import './App.css';
import Greetingclass
  from './components/Greetingclass'
function App() {
  return (
    <div>
      <div>
        <Greetingclass/>
      </div>
    </div>
  );
}
export default App;
```

Class component with props

```
import React from 'react';
class Greetingclass extends React.Component {
  render()
  {
    return
    <h1>
Greeting of the Day
{this.props.fname} {this.props.lname}
</h1>;
  }
}

export default Greetingclass
```

```
import React from 'react';
import './App.css';
import Greetingclass from './components/Greetingclass'
function App() {
  return (
    <div>
      <div>
        <Greetingclass fname="reena"
lname="sharma"></Greetingclass>
      </div>
    </div>
  );
}
export default App;
```

Default props

```
// Greetings Component
```

```
const Greeting = props => <h1>Hello  
{props.name}</h1>;
```

```
// Default Props
```

```
Greeting.defaultProps = {  
  name: "User"  
};
```

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

Functional and Class component

- ***Functional components** can be referenced within **class components** and vice versa. However, it is not often that you will reference a class component within a functional component; class components typically serve as container components.*

Props and State

- **props** are variables passed to it by its parent component
- **State** on the other hand is still variables, but directly initialized and managed by the component.
- The state can be initialized by props.

Props vs State

Props

- Props gets passed to the component
- Function parameters
- props are immutable
- Props-Functional Component
- this.props-Class Component

State

- State is managed within component
- Variables declared in the function body.
- State can be changed
- useState hook-Functional Component
- this.state-class component

Life Cycle of the class component

- **Mounting**(When an instance of a component is being created and inserted into the DOM
 - Constructor
 - static `getDerivedStateFromProps`
 - `render()`
 - `componentDidMount`
- **Updating**(If the state gets changed from within the React or through the api or backend, the component is re-rendered with the new state.)
 - New Props, `setState`, `forceUpdate`
 - `shouldComponentUpdate`
 - static `getDerivedStateFromProps`
 - For new Props, and `setState` no render method will be called, but for `forceUpdate` render method will be invoked
 - `getSnapshotBeforeUpdate`
 - `componentDidUpdate`

Life Cycle of the class component

- **UnMounting**(Components can be removed from the DOM after mounting. The method for this event)
 - `componentWillUnmount`
- **Error Handling**(When there is an error during rendering, in a lifecycle method or in constructor of child component)
 - static `getDerivedStateFromError` and `componentDidCatch`

Component Mounting Lifecycle Method

- **Constructor(props)**
 - A special function that will get called whenever a new component is created.
 - Initialize the State, Binding the event handlers
 - `super(props)` directly overwrite `this.state`
- **static getDerivedStateFromProps(props, state)**
- **render()**
 - Only required method
 - Read props & state and return JSX
 - Do not change state or interact with DOM or make ajax call.
 - Children component life cycle method are also executed
- **componentDidMount()**
 - Invokes immediately after a component and all its children components have been rendered to the DOM
 - Cause side effects. Ex: Interact with DOM or perform any ajax calls to load data

```

import React from 'react'
class LifecycleA extends React.Component
{
  constructor(props)
  {
    super(props);
    this.state={name:''}
    console.log("Constructor Life Cycle A");
  }
  static getDerivedStateFromProps(props,state)
  {
    console.log("getDerivedStateFromPropslife cycle A")
    return null;
  }
  render()
  {
    console.log("Life Cycle A render");
    return(
      <div>
        <div>Lifecycle A </div>

        </div>
      );
  }
  componentDidMount()
  {
    console.log("Component Did Mount A");
  }
}
export default LifecycleA;

```

You can find a complete log in the terminal.

i HTML1300: Navigation occurred.

[HMR] Waiting for update signal from WDS...

2 Constructor Life Cycle A

2 getDerivedStateFromPropslife cycle A

2 Life Cycle A render

Component Did Mount A

```

import React from 'react'
class LifecycleA extends React.Component
{
  constructor(props)
  {
    super(props);
    this.state={name:""}
    console.log("Constructor Life Cycle A");
  }
  static getDerivedStateFromProps(props,state)
  {
    console.log("getDerivedStateFromPropslife cycle A")
    return null;
  }
  render()
  {
    console.log("Life Cycle A render");
    return(
      <div>
        <div>Lifecycle A </div>
        <LifecycleB></LifecycleB>

        </div>
      );
  }
  componentDidMount()
  {
    console.log("Component Did Mount A");
  }
}
export default LifecycleA;

```

```

HTML1300: Navigation occurred.
[HMR] Waiting for update signal from WDS...
2 Constructor Life Cycle A
2 getDerivedStateFromPropslife cycle A
2 Life Cycle A render
2 Constructor Life Cycle B
2 getDerivedStateFromPropslife cycle B
2 Life Cycle B render
Component Did Mount B
Component Did Mount A

```

In case of child component componentDidMount method of Child is called first before the parent

```

import React from 'react';
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>Hello, world!</h1>
        <h2>It is {this.state.date.toLocaleTimeString()}</h2>
      </div>
    );
  }
}
export default Clock;

```

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.

Updating LifeCycle Method

- `static getDerivedStateFromProps(props, state)`
 - Method is called every time component is re-rendered
 - Set the State
- `shouldComponentUpdate(nextProps, nextState)`
 - Dictate if the component should re-render or not
 - Performance Optimization
- `render()`
 - Only required method
 - Read props&state and return JSX.
 - Do not change state or interact with DOM or make ajax call

Updating LifeCycle Method

- `getSnapshotBeforeUpdate(prevProps,prevState)`
 - Called right before the changes from the virtual DOM are to be reflected in the DOM.
 - Capture some information from the DOM.
 - Method will either return null or return a value. Returned value will be passed as the third parameter to the next method.
- `componentDidUpdate(prevProps,prevState,snapshot)`
 - Calls after the render is finished in the re-render cycles.

```
import React from 'react'
import LifecycleB from './LifecycleB';
class LifeCycleA extends React.Component
{
  constructor(props)
  {
    super(props);
    this.state={name:"Greeting of the day"}
    console.log("Constructor Life Cycle A");
  }
  static getDerivedStateFromProps(props,state)
  {
    console.log("getDerivedStateFromPropslife cycle A")
    return null;
  }
  changeState=()=>>
  {
    this.setState({name:"Welcome"});
  }
  render()
  {
    console.log("Life Cycle A render");
    return(
      <div>
        <p>{this.state.name}</p>
        <div>LifeCycle A
          <button onClick={this.changeState}>Change State</button>
        </div>
        <LifecycleB></LifecycleB>
      </div>
    );
  }
}
```



```
componentDidMount()  
{  
  console.log("Component Did Mount A");  
}  
shouldComponentUpdate()  
{  
  console.log("Life Cycle A shouldcomponentUpdate");  
  return true;  
}  
getSnapshotBeforeUpdate(prevProps,prevState)  
{  
  console.log("Life Cycle A getSnapshotBeforeUpdate");  
  return null;  
}  
componentDidUpdate()  
{  
  console.log("Lifecycle A component did update method")  
}  
}  
export default LifeCycleA;
```

Output

Welcome

LifeCycle A

LifeCycle

Whenever a user clicks on change State following output will rendered

2 `getDerivedStateFromProps` life cycle A

2 `Life Cycle A shouldComponentUpdate`

2 `Life Cycle A render`

2 `getDerivedStateFromProps` life cycle B

2 `Life Cycle B render`

`Life Cycle A getSnapshotBeforeUpdate`

`Lifecycle A component did update method`

Using State Correctly

- Do not modify State directly
 - Following will not re-render a component

```
this.state.comment="Hello">//wrong
```

- Instead use useState

```
// Correct  
this.setState({comment: 'Hello'});
```

The only place where you can assign this.state is the constructor

State Updates May Be Asynchronous

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

State Updates are Merged

- When you call `setState()`, React merges the object you provide into the current state.
- For example, your state may contain several independent variables:

```
constructor(props) {  
  super(props);  
  this.state = {  
    posts: [],  
    comments: []  
  };  
}
```

Then you can update them independently with separate `setState()` calls:

```
componentDidMount() {  
  fetchPosts().then(response => {  
    this.setState({  
      posts: response.posts  
    });  
  });  
  
  fetchComments().then(response => {  
    this.setState({  
      comments: response.comments  
    });  
  });  
}
```

The merging is shallow, so `this.setState({comments})` leaves `this.state.posts` intact, but completely replaces `this.state.comments`.

Composing Components

- React allows you to reference components within other components, allowing you to add a level(s) of abstraction to your application.
- Take for example a user profile component on a social network. We could write this component's structure like so:

UserProfile

| -> Avatar

| -> UserName

| -> Bio

Compose Component

```
import React from 'react';  
//Avatar component  
const Avatar = () => ;
```

```
//Username component  
const UserName = () => <h4>janedoe</h4>;
```

```
//Bio component  
const Bio = () =>  
  <p>  
    <strong>Bio: </strong>  
    Lorem ipsum dolor sit amet, justo  
a bibendum phasellus proodio  
    ligula, sit  
  </p>;
```

```
// UserProfile component  
const UserProfile = () =>  
  <div>  
    <Avatar/>  
    <UserName/>  
    <Bio/>  
  </div>;  
export default UserProfile
```

```
import React from 'react';  
import './App.css';  
import UserProfile from './components/composeco  
mponent'  
function App() {  
  return (  
    <div>  
      <UserProfile/>,  
    </div>  
  );  
}  
export default App;
```


Children Composition

- children prop that might be used with every React Component.
- children is a special prop, typically defined by the child tags in the JSX expression rather than in the tag itself.

```
import React from 'react';
function FancyBorder(props) {
  return (
    <div className={'FancyBorder FancyBorder-' + props.color}>
      {props.children}
    </div>
  );
}
export default FancyBorder;
```

Children Composition

```
<FancyBorder color="blue">  
  <h1 className="Dialog-title">  
    Welcome  
  </h1>  
  <p className="Dialog-message">  
    Thank you for visiting our spacecraft!  
  </p>  
</FancyBorder>
```

Lists and Keys

```
function NumberList(props) {  
  const numbers = props.numbers;  
  const listItems = numbers.map((number) =>  
    <li>{number}</li>  
  );  
  return (  
    <ul>{listItems}</ul>  
  );  
}
```

```
const numbers = [1, 2, 3, 4, 5];  
ReactDOM.render(  
  <NumberList numbers={numbers} />,  
  document.getElementById('root')  
);
```

Lists and Keys

We use the [map\(\)](#) function to take an array of numbers and double their values. We assign the new array returned by map() to the variable doubled and log it.

```
const numbers = [1, 2, 3, 4, 5];  
const doubled = numbers.map((number) => number * 2);  
console.log(doubled);
```

```
const numbers = [1, 2, 3, 4, 5];  
const listItems = numbers.map((number) =>  
  <li>{number}</li>  
);
```

```
ReactDOM.render(  
  <ul>{listItems}</ul>,  
  document.getElementById('root')  
);
```

Lists and Keys

```
function NumberList(props) {  
  const numbers = props.numbers;  
  const listItems = numbers.map((number) =>  
    <li key={number.toString()}>  
      {number}  
    </li>  
  );  
  return (  
    <ul>{listItems}</ul>  
  );  
}
```

```
const numbers = [1, 2, 3, 4, 5];  
ReactDOM.render(  
  <NumberList numbers={numbers} />,  
  document.getElementById('root')  
);
```

Lists and Keys

```
import React from 'react'
function GetList(props)
{
  const list = props.list;
  const listItems = list.map((listItem) =>
    <li key={listItem.FirstName.toString()}
  >
{listItem.FirstName}</li>
  );
  return (
    <ul>{listItems}</ul>
  );
}
export default GetList
```

```
import React from 'react';
import './App.css';
import GetList from './components/basiclist'

const peoplelist=[{"FirstName":"Ronit","LastName":"Patel"},
{"FirstName":"Reena","LastName":"Sharma"}]

function App() {
  return (
    <div>
      <GetList list={peoplelist}></GetList>
    </div>
  );
}
export default App;
```

Output



- Ronit
- Reena

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.

```
import React from 'react'
function info(e)
{
  e.preventDefault();
  console.log("Welcome");
}
function Test()
{
  return(
    <div>
      <button onClick={info}>Click</button>
    </div>
  );
}
export default Test
```

Another difference is that you cannot return false to prevent default behavior in React. You must call preventDefault explicitly.


```
class Toggle extends React.Component {
  constructor(props) {
    super(props);
    this.state = {isToggleOn: true};

    // This binding is necessary to make `this` work in the callback
    this.handleClick = this.handleClick.bind(this);
  }

  handleClick() {
    this.setState(state => ({
      isToggleOn: !state.isToggleOn
    }));
  }

  render() {
    return (
      <button onClick={this.handleClick}>
        {this.state.isToggleOn ? 'ON' : 'OFF'}
      </button>
    );
  }
}

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

Passing Arguments to Event Handlers

```
<button onClick={(e) => this.deleteRow(id, e)}></button>  
<button onClick={this.deleteRow.bind(this, id)}></button>
```

Forms

- **Controlled Component**
 - In HTML, form elements such as `<input>`, `<textarea>`, and `<select>` typically maintain their own state and update it based on user input. In React, mutable state is typically kept in the state property of components, and only updated with [`setState\(\)`](#).
- **Uncontrolled Component**
 - uncontrolled components, where form data is handled by the DOM itself.

Controlled Component example

```
import React,{Component} from 'react';
class Signup extends React.Component
{
  constructor(props)
  {
    super(props);
    this.state={
      name:"reena",address:"ahmedabad",stateinfo:"",gender:"",hobbies:[],
      selectedhobbies:""
    }
    this.handleChange=this.handleChange.bind(this);
    this.handleSubmit=this.handleSubmit.bind(this);
  }
  handleChange(event)
  {
    const target=event.target;
    const value=target.value;
    const name=target.name;
    console.log(target.name);
    if(target.name=="hobbies")
    {
      this.setState.selectedhobbies+=target.value+",";
    }
    else{
      //this.setState(name:value)//single property assign
      this.setState({[name]:value})//multiple property assign
    }
  }
}
```

Controlled Component example

```
handleSubmit(event)
{
    event.preventDefault();
    var message="";
    for(var i=0;i<this.state.hobbies.length;i++)
    {
        message+=this.state.hobbies[i]+" ";
    }
    console.log(message);
    alert(this.state.name+" "+this.state.address+" "+this.state.stateinfo+" "+
this.state.gender+this.state.selectedhobbies);
}
render()
{
    return(
        <form onSubmit={this.handleSubmit}>
            <label>
                Name <input type="text" name="name" value={this.state.name}
onChange={this.handleInputChange} />
            </label>
            <label>
                Address
                <textarea name="address" value={this.state.address} onChange={this.handleInputChange} />
            </label>
            <label>
```

Controlled Component example

```
<select name="stateinfo" value={this.state.stateinfo} onChange={this.handleChange}>
  <option value="Ahmedabad">Ahmedabad</option>
  <option selected value="Mumbai">Mumbai</option>
  <option selected value="Delhi">Delhi</option>
  <option value="Chennai">Chennai</option>
</select>
</label>
<label>
Gender
  <input name="gender" type="radio" value="Male" onChange={this.handleChange}/>Male
  <input name="gender" type="radio" value="Female" onChange={this.handleChange}/>Female
</label>
<label>
Hobbies
  <input name="hobbies" type="checkbox" value="0" onChange={this.handleChange}>Hockey</input>
  <input name="hobbies" type="checkbox" value="1" onChange={this.handleChange}>Cricket</input>
  <input name="hobbies" type="checkbox" value="2" onChange={this.handleChange}>Badminton</input>

</label>
<input type="submit" value="Submit" />
</form>

  );
}
}
export default Signup;
```

Uncontrolled Component example

```
import React,{Component} from 'react';
class Signupuncontrolled extends React.Component
{
  constructor(props)
  {
    super(props);
    this.name = React.createRef();
    this.address=React.createRef();
    this.stateinfo=React.createRef();

    this.handleSubmit=this.handleSubmit.bind(this);
  }

  handleSubmit(event)
  {
    event.preventDefault();
    alert(this.name.value+" "+this.address.value+" "+this.stateinfo.value);
    // event.preventDefault();
  }
}
```

```
render()
{
  return(
    <form onSubmit={this.handleSubmit}>
      <label>
        Name <input type="text" ref={input => this.name = input} />
      </label>
      <label>
        Address
        <textarea ref={input => this.address = input} />
      </label>
      <label>[
        <select name="stateinfo" ref={input => this.stateinfo = input}>
          <option value="Ahmedabad">Ahmedabad</option>
          <option value="Mumbai">Mumbai</option>
          <option selected value="Delhi">Delhi</option>
          <option value="Chennai">Chennai</option>
        </select>
      </label>
      <input type="submit" value="Submit" />
    </form>
  );
}
export default Signupuncontrolled;
```


Lifting StateUp

- Often, several components need to reflect the same changing data.

Product Order screen

The diagram illustrates a 'Product Order screen' with three sections: 'Product Information', 'Address Information', and 'Summary Information'. Each section is enclosed in a red border. The 'Product Information' section contains a 'Product Name' dropdown menu with 'Product1' selected and an 'Enter Quantity' input field with the value '56'. The 'Address Information' section contains an 'Enter Address' input field with the value 'fdgdfg'. The 'Summary Information' section contains a 'Product Name' label with the value 'Product-1', an 'Address' label with the value 'fdgdfg', and an 'Enter Quantity' input field with the value '56'. Blue arrows indicate the flow of state changes: a vertical arrow on the right points upwards from the 'Summary Information' section to the 'Product Information' section, labeled 'Sync changes from product information to summary information & from summary Information to Product Information'. A horizontal arrow points from the 'Product Information' section to the right, towards the 'Parent Component contains' list. A horizontal arrow points from the 'Address Information' section to the 'Summary Information' section, labeled 'Address sync'. A horizontal arrow points from the 'Enter Quantity' field in the 'Summary Information' section back to the 'Enter Quantity' field in the 'Product Information' section.

Product Information

Product Name

Enter Quantity :

Address Information

Enter Address :

Summary Information

Product Name **Product-1**

Address fdgdfg

Enter Quantity :

Parent Component contains

- Product Information
- Address Information
- SummarInformation

Sync changes from product information to summary information & from summary Information to Product Information

Order Component

```
import React from 'react'
class Order extends React.Component
{
  constructor(props)
  {
    super(props);
    this.state={quantity:'',Address:''}
  }
  OrderInfoChanged=val=>{
    this.setState({quantity:val})
  }
  AddressChanged=val=>{this.setState({Address:val})}
  render()
  {
    return(
      <div>
        <h1>Product Order screen</h1>
        <ProductInformation quantity={this.state.quantity} onQuantityChange={this.OrderInfoChanged}>
</ProductInformation>
        <AddressInformation Address={this.state.Address} onAddressChange={this.AddressChanged}></Add
ressInformation>
        <SummaryInformation quantity={this.state.quantity} Address={this.state.Address}
onQuantityChange={this.OrderInfoChanged}></SummaryInformation>
      </div>
    )
  }
}
```

Product Information Component

```
class ProductInformation extends React.Component
{
  constructor(props)
  {
    super(props)
  }
  handleChange =(e)=>{
    this.props.onQuantityChange(e.target.value)
  }
  render()
  {
    return(
      <div style={{border: '3px solid red'}}>
        <h2>Product Information</h2>
        <p>
          <label>Product Name
            <select>
              <option value="Product-1">Product1 </option>
              <option value="Product-2">Product2 </option>
              <option value="Product-3">Product </option>
            </select>
          </label>
        </p>
        <label>Enter Quantity :<input type="text" value={this.props.quantity}
onChange={this.handleChange}></input></label>
      </div>
    );
  }
}
```

Address Information

```
class AddressInformation extends React.Component
{
  constructor(props)
  {
    super(props)
  }
  handleChange =(e)=>{
    this.props.onAddressChange(e.target.value)
  }
  render()
  {
    return(

      <div style={{border: '3px solid red'}}>
        <h2>Address Information</h2>

        <label>Enter Address :<textarea value={this.props.Address} onChange={this.handleChange}/>
      </label>
      </div>
    );
  }
}
```

Summary Information

```
class SummaryInformation extends React.Component
{
  constructor(props)
  {
    super(props)
  }
  handleChange = (e) => {
    this.props.onQuantityChange(e.target.value)
  }
  render()
  {
    return(
      <div style={{border: '3px solid red'}}>
        <h1>Summary Information</h1>
        <p>
          <label>Product Name <b>Product-1</b></label>
        </p>
        <p>
          Address <label>{this.props.Address}</label>
        </p>
        <label>Enter Quantity :<textarea value={this.props.quantity} onChange={this.handleChange}/></label>
      </div>
    );
  }
}
```

axios

- Axios helps us to fetch data from webapi.
- `npm install axios --save`

Sending Get Request

- `import axios from 'axios';`