

Unit-5

React JS

1. Install React JS
2. Create-react-app
3. React Router
4. React Components
5. State
6. Props
7. React Forms
8. Component Life-Cycle
9. React Redux
10. Angular vs React JS.

Introduction:-

1.Defination:-

ReactJS is a declarative, efficient, and flexible JavaScript library for building reusable UI components. It is an open-source, component-based front end library responsible only for the view layer of the application. It was created by **Jordan Walke**, who was a software engineer at **Facebook**. It was initially developed and maintained by Facebook and was later used in its products like **WhatsApp & Instagram**. Facebook developed ReactJS in **2011** in its newsfeed section, but it was released to the public in the month of **May 2013**.

Today, most of the websites are built using MVC (model view controller) architecture. In MVC architecture, React is the 'V' which stands for view, whereas the architecture is provided by the Redux or Flux.

- A ReactJS application is made up of multiple components, each component responsible for outputting a small, reusable piece of HTML code. The components are the heart of all React applications. These Components can be nested with other components to allow complex applications to be built of simple building blocks. ReactJS uses virtual DOM based mechanism to fill data in HTML DOM. The virtual DOM works fast as it only changes individual DOM elements instead of reloading complete DOM every time.
- To create React app, we write React components that correspond to various elements. We organize these components inside higher level components which define the application structure. For example, we take a form that consists of many elements like input fields, labels, or buttons. We can write each element of the form as React components, and then we combine it into a higher-level component, i.e., the form component itself. The form components would specify the structure of the form along with elements inside of it.

2. Why learn ReactJS?

- Today, many JavaScript frameworks are available in the market(like angular, node), but still, React came into the market and gained popularity amongst them. The previous frameworks follow the traditional data

flow structure, which uses the DOM (Document Object Model). DOM is an object which is created by the browser each time a web page is loaded. It dynamically adds or removes the data at the back end and when any modifications were done, then each time a new DOM is created for the same page. This repeated creation of DOM makes unnecessary memory wastage and reduces the performance of the application.

- Therefore, a new technology ReactJS framework invented which remove this drawback. ReactJS allows you to divide your entire application into various components. ReactJS still used the same traditional data flow, but it is not directly operating on the browser's Document Object Model (DOM) immediately; instead, it operates on a virtual DOM. It means rather than manipulating the document in a browser after changes to our data, it resolves changes on a DOM built and run entirely in memory. After the virtual DOM has been updated, React determines what changes made to the actual browser's DOM. The React Virtual DOM exists entirely in memory and is a representation of the web browser's DOM. Due to this, when we write a React component, we did not write directly to the DOM; instead, we are writing virtual components that react will turn into the DOM.

3. React.JS History

Current version of React.JS is V18.0.0 (April 2022).

Initial Release to the Public (V0.3.0) was in July 2013.

React.JS was first used in 2011 for Facebook's Newsfeed feature.

Facebook Software Engineer, Jordan Walke, created it.

Current version of create-react-app is v5.0.1 (April 2022).

create-react-app includes built tools such as webpack, Babel, and ESLint.

4. Features of React

React offers some outstanding features that make it the most widely adopted library for frontend app development. Here is the list of those salient features.

- **JSX**



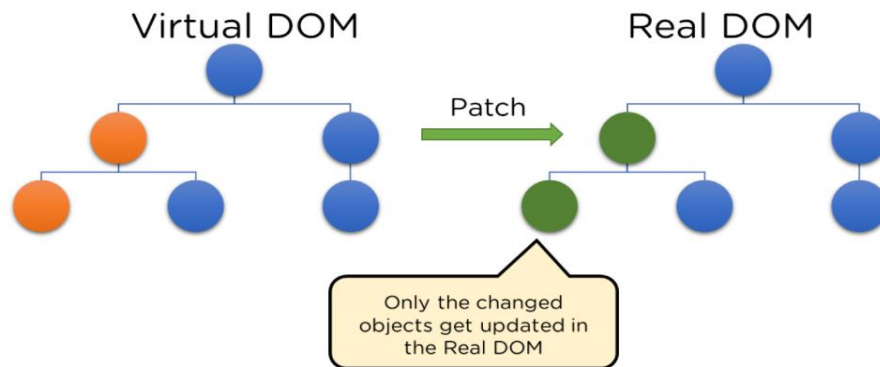
JSX is a JavaScript syntactic extension. It's a term used in React to describe how the user interface should seem. You can write HTML structures in the same file as JavaScript code by utilizing JSX.

```
const name = 'Simplilearn';
```

```
const greet = <h1>Hello, {name}</h1>;
```

The above code shows how JSX is implemented in React. It is neither a string nor HTML. Instead, it embeds HTML into JavaScript code.

- **Virtual Document Object Model (DOM)**



The Virtual DOM is React's lightweight version of the Real DOM. Real DOM manipulation is substantially slower than virtual DOM manipulation. When an object's state changes, Virtual DOM updates only that object in the real DOM rather than all of them.

- What is the Document Object Model (DOM)?

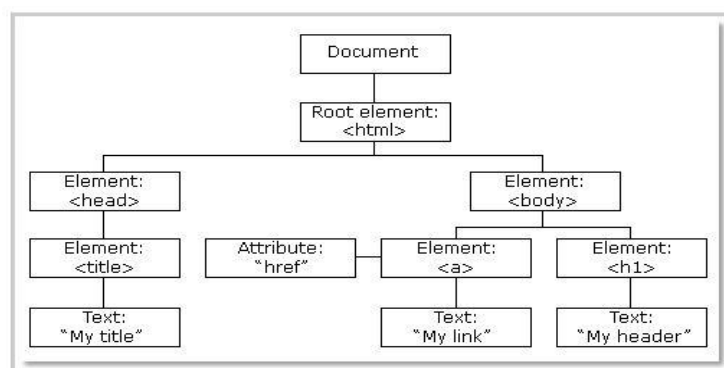


Fig: DOM of a Webpage

DOM (Document Object Model) treats an XML or HTML document as a tree structure in which each node is an object representing a part of the document.

- How do Virtual DOM and React DOM interact with each other?

When the state of an object changes in a React application, VDOM gets updated. It then compares its previous state and then updates only those objects in the real DOM instead of updating all of the objects. This makes things move fast, especially when compared to other front-end technologies that have to update each object even if only a single object changes in the web application.

- **Architecture**

In a Model View Controller(MVC) architecture, React is the 'View' responsible for how the app looks and feels.

MVC is an architectural pattern that splits the application layer into Model, View, and Controller. The model relates to all data-related logic; the view is used for the UI logic of the application, and the controller is an interface between the Model and View.

- **Extensions**



React goes beyond just being a UI framework; it contains many extensions that cover the entire application architecture. It helps the building of mobile apps and provides server-side rendering. Flux and Redux, among other things, can extend React.

- **Data Binding**

Since React employs one-way data binding, all activities stay modular and quick. Moreover, the unidirectional data flow means that it's common to nest child components within parent components when developing a React project.



Fig: One-way data binding

- **Debugging**

Since a broad developer community exists, React applications are straightforward and easy to test. Facebook provides a browser extension that simplifies and expedites React debugging.

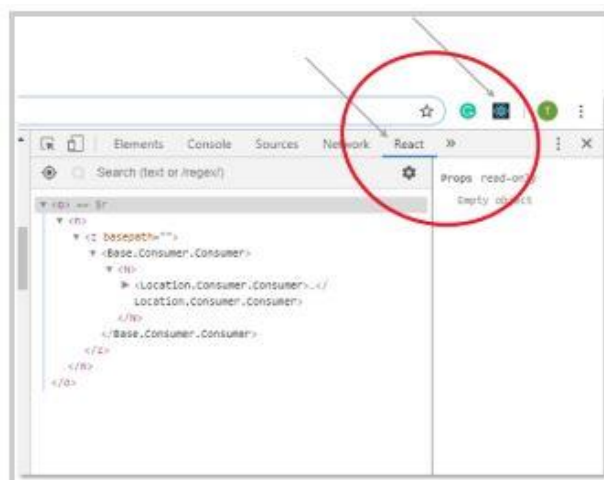
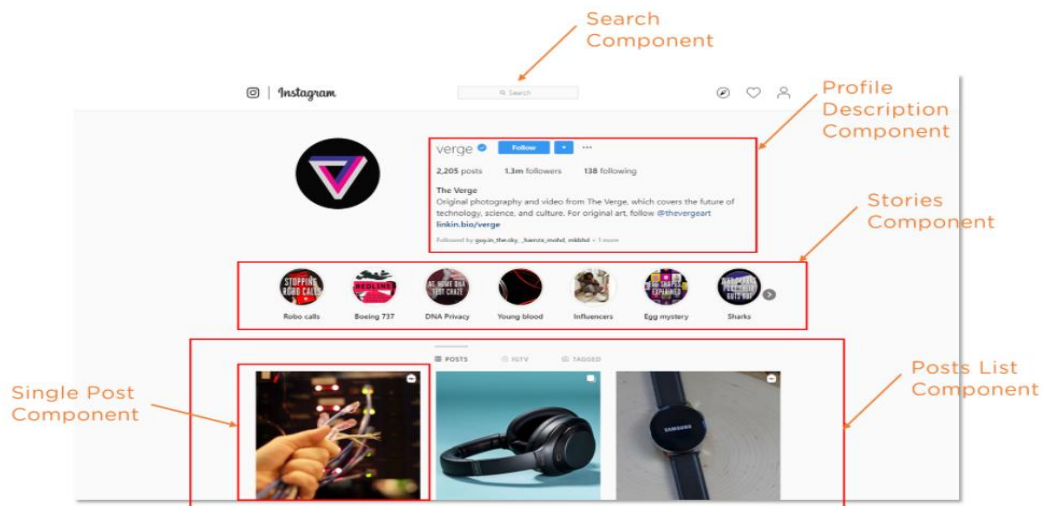


Fig: React Extension

This extension, for example, adds a React tab in the developer tools option within the Chrome web browser. The tab makes it easy to inspect React components directly.

- **Components in React**

Components are the building blocks that comprise a React application representing a part of the user interface.



React separates the user interface into numerous components, making debugging more accessible, and each component has its own set of properties and functions.

- **Single-Page Applications (SPAs)**

React is recommended in creating SPAs, allowing smooth content updates without page reloads. Its focus on reusable components makes it ideal for real-time applications.

1. Install React JS

How To Install React on Windows

In this section, we'll guide you through the process of installing React on a Windows machine.

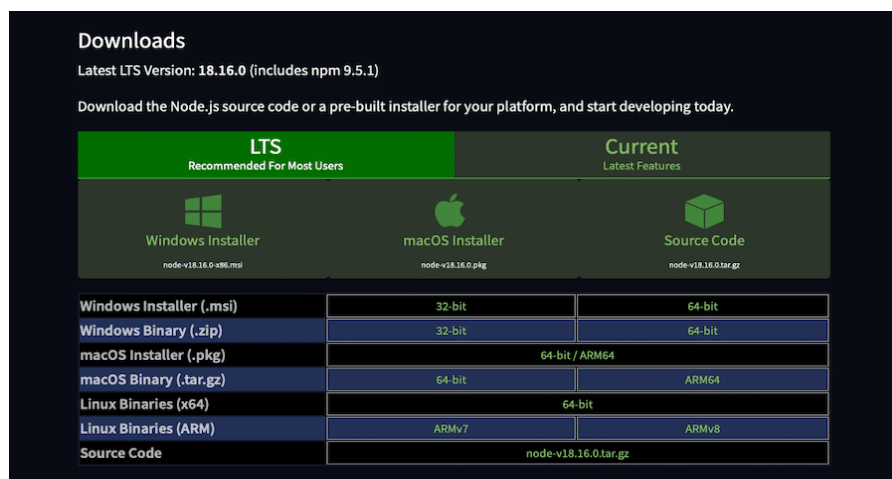
Follow these steps to get started:

1. [Step 1: Install Node.js and npm](#)
2. [Step 2: Install Create React App](#)
3. [Step 3: Create a New React Project](#)
4. [Step 4: Go To the Project Directory and Start the Development Server](#)

Step 1: Install Node.js and npm

Before installing React, you need to have Node.js and npm (Node Package Manager) installed on your system. If you haven't already installed them, follow these steps:

1. Visit the Node.js download page at: <https://nodejs.org/en/download/>
2. Download the installer for your Windows system (either the LTS or Current version is fine, but the LTS version is recommended for most users)
3. To install Node.js and npm, please run the installer and carefully follow the provided prompts.



Downloading the Node.js installer for Windows.

After the installation is complete, you can verify that Node.js and npm are installed by opening a command prompt and running the following commands:

- `node -v`
- `npm -v`

These commands should display the version numbers for Node.js and npm, respectively.

Step 2: Install Create React App

Create React App is a command-line tool that simplifies the process of setting up a new React project with a recommended project structure and configuration. To install Create React App globally, open a command prompt and run the following command:

- `npm install -g create-react-app`

This command installs Create React App on your system, making it available to use in any directory.

Step 3: Create a New React Project

Now that you have Create React App installed, you can use it to create a new React project. To do this, open a command prompt, go to the directory where you want the project to live, and run the following command:

- `create-react-app my-app`

Replace “my-app” with the desired name for your project. Create React App will create a new directory with the specified name and generate a new React project with a recommended project structure and configuration.

Step 4: Go To the Project Directory and Start the Development Server

Once the project is created, head over to the project directory by running the following command in the command prompt:

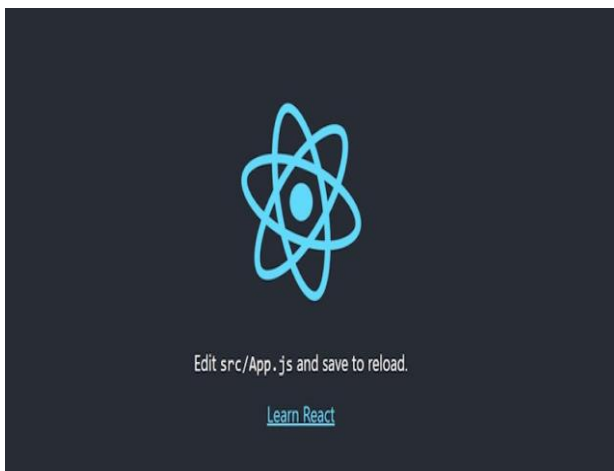
```
cd my-app
```

Replace “my-app” with the name of your project directory. Now, start the development server by running the following command:

```
npm start
```

This command launches the development server, which watches for changes to your project files and automatically reloads the browser when changes are detected.

A new browser window should open with your React application running at <http://localhost:3000/> that looks like this:



React has been successfully installed on Windows.

Congratulations! You have successfully installed React on your Windows machine and created a new React project. You can now begin building your user interfaces with React.

2. React create-react-app

Starting a new React project is very complicated, with so many build tools. It uses many dependencies, configuration files, and other requirements such as Babel, Webpack, ESLint before writing a single line of React code. Create React App CLI tool removes all that complexities and makes React app simple. For this, you need to install the package using NPM, and then run a few simple commands to get a new React project.

The **create-react-app** is an excellent tool for beginners, which allows you to create and run React project very quickly. It does not take any configuration manually. This tool is wrapping all of the required dependencies like **Webpack**, **Babel** for React project itself and then you need to focus on writing React code only. This tool sets up the development environment, provides an excellent developer experience, and optimizes the app for production.

Create React App is a command-line tool that simplifies the process of setting up a new React project with a recommended project structure and configuration.

Requirements

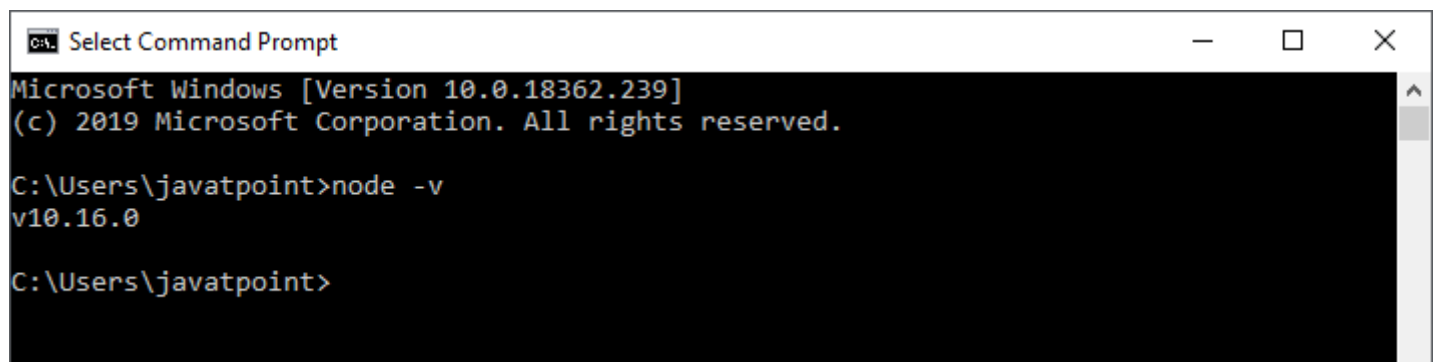
The Create React App is maintained by **Facebook** and can works on any **platform**, for example, macOS, Windows, Linux, etc. To create a React Project using create-react-app, you need to have installed the following things in your system.

1. Node version
2. NPM version

Let us check the current version of **Node** and **NPM** in the system.

Run the following command to check the Node version in the command prompt.

1. \$ node -v



```
Microsoft Windows [Version 10.0.18362.239]
(c) 2019 Microsoft Corporation. All rights reserved.

C:\Users\javatpoint>node -v
v10.16.0

C:\Users\javatpoint>
```

Run the following command to check the NPM version in the command prompt.

1. \$ npm -v



```
C:\Users\javatpoint>npm -v
6.9.0

C:\Users\javatpoint>
```


Install React

We can install React using npm package manager by using the following command. There is no need to worry about the complexity of React installation. The create-react-app npm package manager will manage everything, which needed for React project.

1. C:\Users\javatpoint> **npm install -g create-react-app**

Note:-If you've previously installed create-react-app globally, it is recommended that you uninstall the package to ensure npx always uses the latest version of create-react-app.

To uninstall, run this command: `npm uninstall -g create-react-app`.

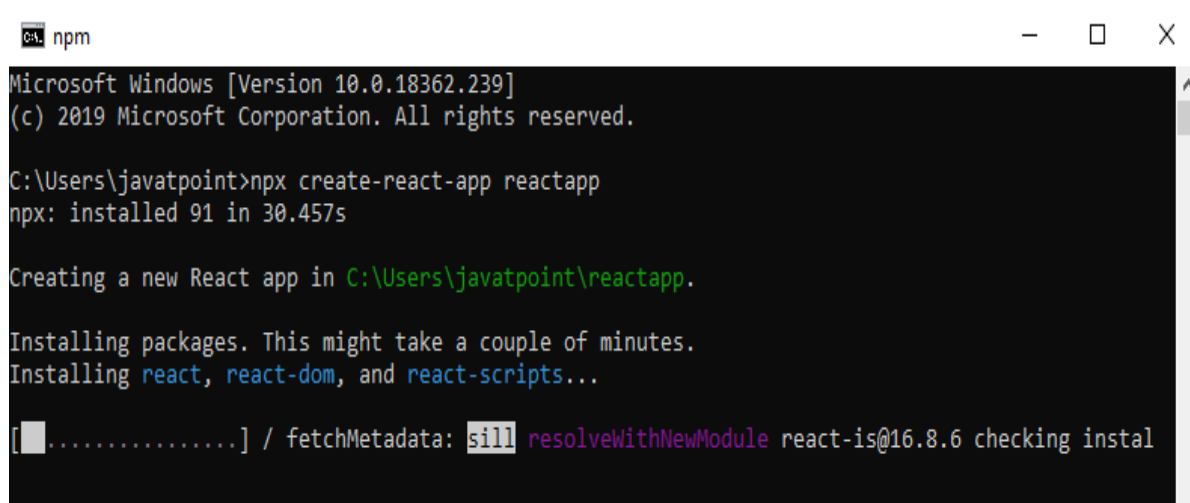
Create a new React project

Once the React installation is successful, we can create a new React project using create-react-app command. Here, I choose "reactproject" name for my project.

1. C:\Users\javatpoint> **create-react-app reactproject**

NOTE: We can combine the above two steps in a single command using npx. The npx(Node Package eXecute) is a package runner tool which comes with npm 5.2 and above version.

1. C:\Users\javatpoint> **npx create-react-app reactproject**



```
Microsoft Windows [Version 10.0.18362.239]
(c) 2019 Microsoft Corporation. All rights reserved.

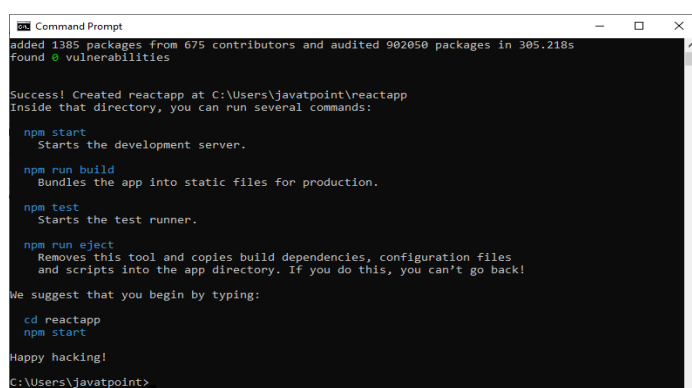
C:\Users\javatpoint>npx create-react-app reactapp
npx: installed 91 in 30.457s

Creating a new React app in C:\Users\javatpoint\reactapp.

Installing packages. This might take a couple of minutes.
Installing react, react-dom, and react-scripts...

[.....] / fetchMetadata: sill resolveWithNewModule react-is@16.8.6 checking instal
```

The above command will take some time to install the React and create a new project with the name "reactproject." Now, we can see the terminal as like below.



```
Command Prompt
added 1385 packages from 675 contributors and audited 902050 packages in 305.218s
Found 0 vulnerabilities

Success! Created reactapp at C:\Users\javatpoint\reactapp
Inside that directory, you can run several commands:

  npm start
    Starts the development server.

  npm run build
    Bundles the app into static files for production.

  npm test
    Starts the test runner.

  npm run eject
    Removes this tool and copies build dependencies, configuration files
    and scripts into the app directory. If you do this, you can't go back!

We suggest that you begin by typing:

  cd reactapp
  npm start

Happy hacking!
C:\Users\javatpoint>
```

The above screen tells that the React project is created successfully on our system. Now, we need to start the server so that we can access the application on the browser. Type the following command in the terminal window.

Run the React Application

Now you are ready to run your first *real* React application!

Run this command to move to the my-react-app directory:

```
cd my-react-app
```

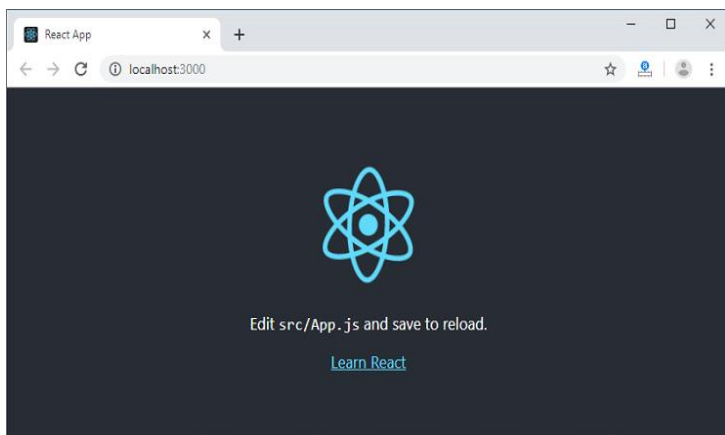
Run this command to run the React application my-react-app:

```
npm start
```

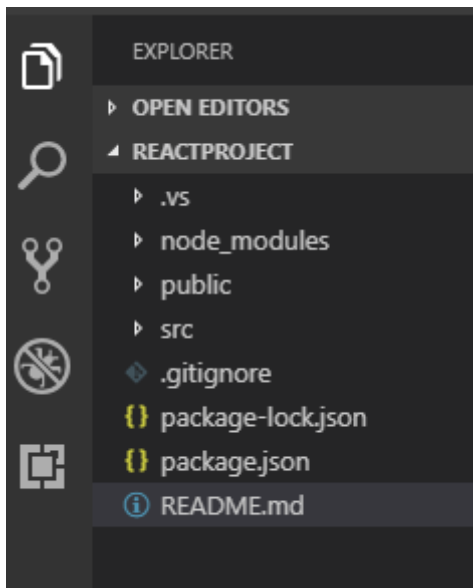
A new browser window will pop up with your newly created React App! If not, open your browser and type localhost:3000 in the address bar.

The result:

NPM is a package manager which starts the server and access the application at default server <http://localhost:3000>. Now, we will get the following screen.



Next, open the project on Code editor. Here, I am using Visual Studio Code. Our project's default structure looks like as below image.



In React application, there are several files and folders in the root directory. Some of them are as follows:

1. **node_modules:** It contains the React library and any other third party libraries needed.
2. **public:** It holds the public assets of the application. It contains the index.html where React will mount the application by default on the `<div id="root"></div>` element.
3. **src:** It contains the App.css, App.js, App.test.js, index.css, index.js, and serviceWorker.js files. Here, the App.js file always responsible for displaying the output screen in React.
4. **package-lock.json:** It is generated automatically for any operations where npm package modifies either the node_modules tree or package.json. It cannot be published. It will be ignored if it finds any other place rather than the top-level package.
5. **package.json:** It holds various metadata required for the project. It gives information to npm, which allows to identify the project as well as handle the project's dependencies.
6. **README.md:** It provides the documentation to read about React topics.

Modify the React Application

Now, open the **src >> App.js** file and make changes which you want to display on the screen. After making desired changes, **save** the file. As soon as we save the file, Webpack recompiles the code, and the page will refresh automatically, and changes are reflected on the browser screen. Now, we can create as many components as we want, import the newly created component inside the **App.js** file and that file will be included in our main **index.html** file after compiling by Webpack.

Look in the my-react-app directory, and you will find a src folder. Inside the src folder there is a file called App.js, open it and it will look like this:

```
/myReactApp/src/App.js:
```

```
import logo from './logo.svg';
```

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

```
function App() {
```

```
  return (
```

```
    <div className="App">
```

```
      <header className="App-header">
```

```
        <img src={logo} className="App-logo" alt="logo" />
```

```
        <p>
```

```
          Edit <code>src/App.js</code> and save to reload.
```

```
        </p>
```

```
        <a
```

```
          className="App-link"
```

```
          href="https://reactjs.org"
```

```
          target="_blank"
```

```
    rel="noopener noreferrer"
  >
  Learn React
</a>
</header>
</div>
);
}
export default App;
```

Try changing the HTML content and save the file.

Notice that the changes are visible immediately after you save the file, you do not have to reload the browser!

Example

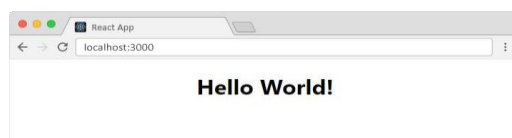
Replace all the content inside the `<div className="App">` with a `<h1>` element.

See the changes in the browser when you click Save.

```
function App() {
  return (
    <div className="App">
      <h1>Hello World!</h1>
    </div>
  );
}
export default App;
```

Notice that we have removed the imports we do not need (`logo.svg` and `App.css`).

The result:



Next, if we want to make the project for the production mode, type the following command. This command will generate the production build, which is best optimized.

1. `$ npm build`

3. React Router

a. What is a React Router?

React Router is a standard library for routing in React. It enables the navigation among views of various components in a React Application, allows changing the browser URL, and keeps the UI in sync with the URL. Let us create a simple application to React to understand how the React Router works. The application will contain three components the **home component**, the **Blogs component**, and the **contact component**. We will use React Router to navigate between these components.

b. Steps to Use React Router

Step 1: Initialize a react project.

```
npx create-react-app Reactrouterprogram
```

Step 2: Install react-router in your application write the following command in your terminal

```
npm i react-router-dom
```

Step 3: Importing React Router

```
import { BrowserRouter, Routes, Route } from "react-router-dom";
```

Folder Structure:

The updated dependencies in **package.json** file.

```
"dependencies": {  
  "@testing-library/jest-dom": "^5.17.0",  
  "@testing-library/react": "^13.4.0",  
  "@testing-library/user-event": "^13.5.0",  
  "react": "^18.2.0",  
  "react-dom": "^18.2.0",  
  "react-router-dom": "^6.22.1",  
  "react-scripts": "5.0.1",  
  "web-vitals": "^2.1.4"  
},
```

c. React Router Components

The Main Components of React Router are:

- **BrowserRouter:** BrowserRouter is a router implementation that uses the HTML5 history API(pushState, replaceState, and the popstate event) to keep your UI in sync with the URL. It is the parent component that is used to store all of the other components.
- **Routes:** It's a new component introduced in the v6 and an upgrade of the component. The main advantages of Routes over Switch are:
 - Relative s and s
 - Routes are chosen based on the best match instead of being traversed in order.
- **Route:** Route is the conditionally shown component that renders some UI when its path matches the current URL.
- **Link:** The link component is used to create links to different routes and implement navigation around the application. It works like an HTML anchor tag.

d. Implementing React Router

Example: This example shows navigation using react-router-dom to Home, About and Contact Components.

Within the **src** folder, we'll create a folder named **pages** with several files:

src\pages\:

- **Layout.js**

- Home.js
- Blogs.js
- Contact.js
- NoPage.js

Each file will contain a very basic React component.

Now we will use our Router in our `index.js` file.

Example:-

Use React Router to route to pages based on URL:

- **index.js:**

```
import ReactDOM from "react-dom/client";
import { BrowserRouter, Routes, Route } from "react-router-dom";
import Layout from "../pages/Layout";
import Home from "../pages/Home";
import Blogs from "../pages/Blogs";
import Contact from "../pages/Contact";
import NoPage from "../pages/NoPage";
```

```
export default function App() {
  return (
    <BrowserRouter>
      <Routes>
        <Route path="/" element={<Layout />} />
        <Route index element={<Home />} />
        <Route path="blogs" element={<Blogs />} />
        <Route path="contact" element={<Contact />} />
        <Route path="*" element={<NoPage />} />
      </Routes>
    </BrowserRouter>
  );
}
```

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

Example Explained

We wrap our content first with `<BrowserRouter>`.

Then we define our `<Routes>`. An application can have multiple `<Routes>`. Our basic example only uses one.

`<Route>`s can be nested. The first `<Route>` has a path of `/` and renders the `Layout` component.

The nested `<Route>`s inherit and add to the parent route. So the `blogs` path is combined with the parent and becomes `/blogs`.

The `Home` component route does not have a path but has an `index` attribute. That specifies this route as the default route for the parent route, which is `/`.

Setting the `path` to `*` will act as a catch-all for any undefined URLs. This is great for a 404 error page.

Pages / Components

The **Layout** component has `<Outlet>` and `<Link>` elements.

The `<Outlet>` renders the current route selected.

`<Link>` is used to set the URL and keep track of browsing history.

Anytime we link to an internal path, we will use `<Link>` instead of ``.

The "layout route" is a shared component that inserts common content on all pages, such as a navigation menu.

- **Layout.js:**

```
import { Outlet, Link } from "react-router-dom";
```

```
const Layout = () => {
```

```
  return (
```

```
    <>
```

```
    <nav>
```

```
      <ul>
```

```
        <li>
```

```
          <Link to="/">Home</Link>
```

```
        </li>
```

```
        <li>
```

```
          <Link to="/blogs">Blogs</Link>
```

```
        </li>
```

```
        <li>
```

```
          <Link to="/contact">Contact</Link>
```

```
        </li>
```

```
      </ul>
```

```
    </nav>
```

```
    <Outlet />
```

```
  </> )
```

```
};
```

```
export default Layout;
```

- **Home.js:**

```
const Home = () => {  
  
  return <h1>Home</h1>;  
  
};  
  
export default Home;
```

- **Blogs.js:**

```
const Blogs = () => {  
  
  return <h1>Blog Articles</h1>;  
  
};  
  
export default Blogs;
```

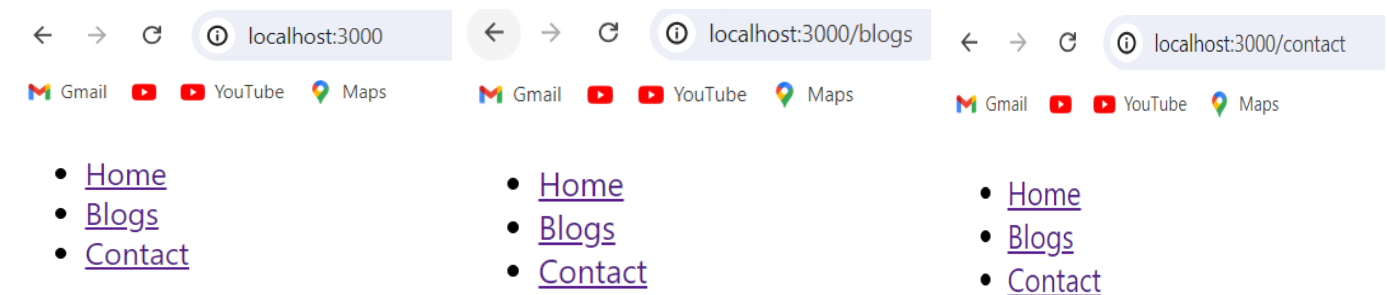
- **Contact.js:**

```
const Contact = () => {  
  
  return <h1>Contact Me</h1>;  
  
};  
  
export default Contact;
```

- **NoPage.js:**

```
const NoPage = () => {  
  
  return <h1>404</h1>;  
  
};  
  
export default NoPage;
```

Output:-



Home

Blog Articles

Contact Me

4. React Components

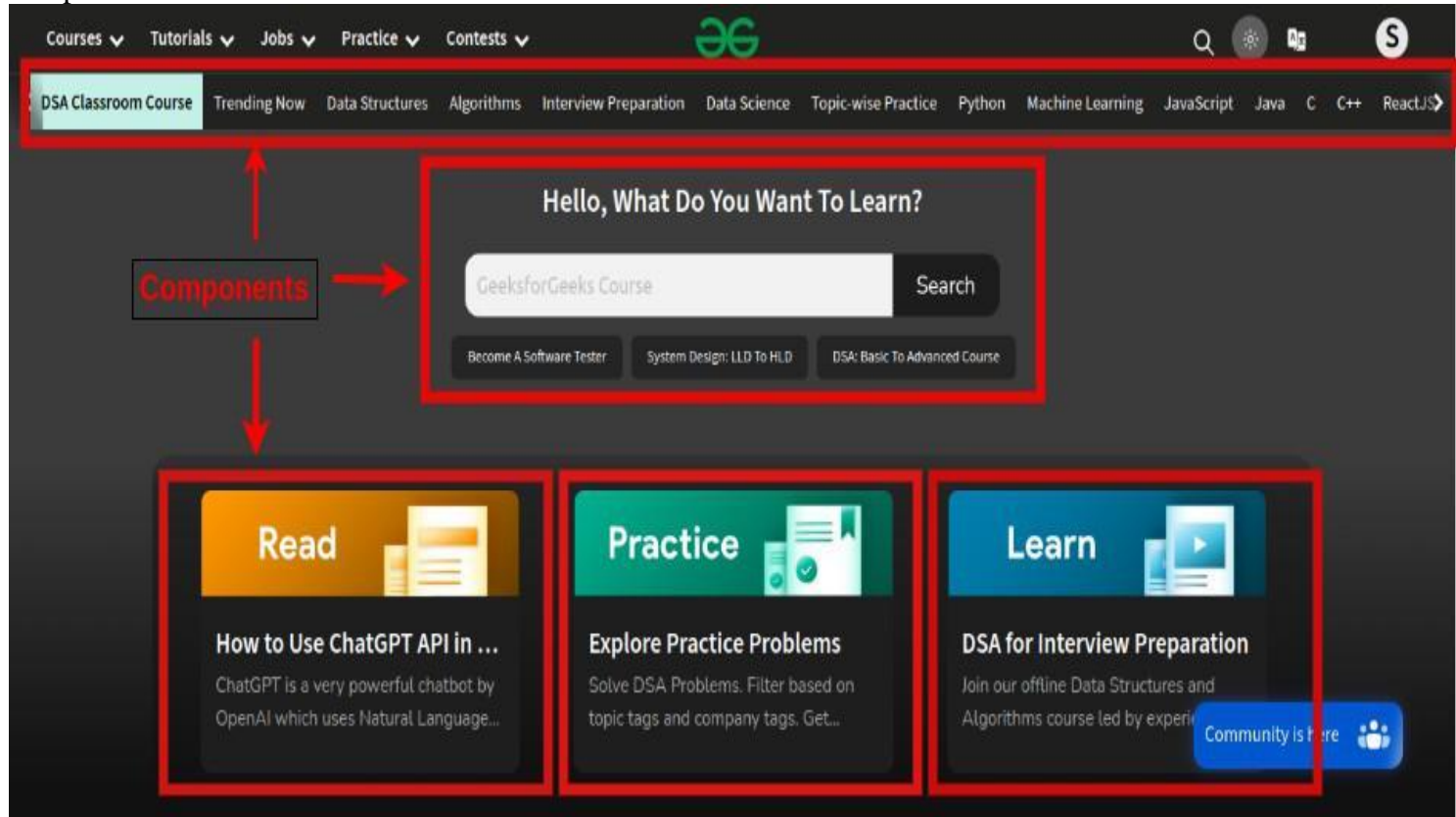
1. What are React Components?

React Components are the building block of React Application. They are the reusable code blocks containing logics and UI elements. They have the same purpose as **JavaScript functions** and return **HTML**. Components make the task of building UI much easier.

A UI is broken down into multiple individual pieces called components. You can work on components independently and then merge them all into a **parent component** which will be your final UI.

Components promote **efficiency** and **scalability** in web development by allowing developers to compose, combine, and customize them as needed.

You can see in the below image we have broken down the UI of GeeksforGeeks's homepage into individual components.



Components in React return a piece of JSX code that tells what should be **rendered on the screen**.

Types of Components in React

In React, we mainly have two types of components:

- **Functional Components**
- **Class Components**
- **Functional Component**

Functional components are just like JavaScript functions that accept properties and return a React element. We can create a functional component in React by writing a JavaScript function. These functions may or may not receive data as parameters. The below example shows a valid functional component in React:

Syntax:

```
function demoComponent() {  
  return (<h1>  
    Welcome Message!  
  </h1>);  
}
```

Example: Create a function component called welcome.
Javascript

```
function welcome() {
  return <h1>Hello, Welcome to GeeksforGeeks!</h1>;
}
```

• Class Component

The [class components](#) are a little more complex than the functional components. A class component can show **inheritance** and access data of other components.

Class Component must include the line “**extends React.Component**” to pass data from one class component to another class component. We can use JavaScript ES6 classes to create class-based components in React.

Syntax:

```
class Democomponent extends React.Component {
  render() {
    return <h1>Welcome Message!</h1>;
  }
}
```

The below example shows a valid class-based component in React:

Example: Create a class component called welcome.

Javascript

```
class Welcome extends Component {
  render() {
    return <h1>Hello, Welcome to GeeksforGeeks!</h1>;
  }
}
```

The components we created in the above two examples are equivalent, and we also have stated the basic difference between a functional component and a class component.

2.Functional Component vs Class Component

- A functional component is best suited for cases where the component doesn't need to interact with other components or manage complex states.
- Functional components are ideal for **presenting static UI elements** or composing multiple simple components together under a single parent component.
- While class-based components can achieve the same result, they are generally **less efficient** compared to functional components. Therefore, it's recommended to not use class components for general use.

3.Rendering React Components

Rendering Components means turning your component code into the UI(User Interface) that users see on the screen.

React is capable of rendering user-defined components. To render a component in React we can initialize an element with a user-defined component and pass this element as the first parameter to [ReactDOM.render\(\)](#) or directly pass the component as the first argument to the ReactDOM.render() method.

The below syntax shows how to initialize a component to an element:

```
const elementName = <ComponentName />;
```

In the above syntax, the *ComponentName* is the name of the user-defined component.

Note: The name of a component should always start with a capital letter. This is done to differentiate a component tag from an [HTML tag](#).

Example: This example renders a component named Welcome to the Screen.

Javascript

```
// Filename - src/index.js:
```

```
import React from "react";
import ReactDOM from "react-dom";
```

```
// This is a functional component
const Welcome = () => {
  return (
    <h1>Hello World!</h1>;
  </>
)
};

ReactDOM.render(
  <Welcome />,
  document.getElementById("root")
);
```

Output: This output will be visible on the **http://localhost:3000/** on the browser window.

Hello World!

Explanation:

Let us see step-wise what is happening in the above example:

- We call the ReactDOM.render() as the first parameter.
- React then calls the component Welcome, which returns <h1>Hello World!</h1>; as the result.
- Then the ReactDOM efficiently updates the DOM to match with the returned element and renders that element to the DOM element with id as “root”.

4.Components in Components

We can call components inside another component

Example:

Javascript

```
// Filename - src/index.js:

import React from "react";
import ReactDOM from "react-dom";

const Greet = () => {
  return <h1>Hello Geek</h1>
}

// This is a functional component
const Welcome = () => {
  return <Greet />;
};

ReactDOM.render(
  <Welcome />,
  document.getElementById("root")
);
```

The above code will give the same output as other examples but here we have called the Greet component inside the Welcome Component.

5. React State:-

a. What Is 'State' in ReactJS?

The state is a built-in React object that is used to contain data or information about the component. A component's state can change over time; whenever it changes, the component re-renders. The change in state can happen as a response to user action or system-generated events and these changes determine the behavior of the component and how it will render.

b. Creating State Object

Creating a state is essential to building dynamic and interactive components. We can create a state object within the **constructor** of the class component.

Example:-

```
import React from 'react';
import ReactDOM from 'react-dom/client';
```

```
class Car extends React.Component {
  constructor(props) {
    super(props);
    this.state = {
      brand: "Ford",
      model: "Mustang",
      color: "red",
      year: 1964
    };
  }
  render() {
    return (
      <div>
        <h1>My {this.state.brand}</h1>
        <p>
          It is a {this.state.color}
            {this.state.model}
            from {this.state.year}.
        </p>
      </div>
    );
  }
}
```

Output:-

My Ford

It is a red Mustang from 1964.

c. Changing the **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 re-render, meaning that the output will change according to the new value(s).

Example:

Add a button with an onClick event that will change the color property:

```
import React from 'react';
import ReactDOM from 'react-dom/client';
```

```
class Car extends React.Component {
  constructor(props) {
    super(props);
    this.state = {
      brand: "Ford",
      model: "Mustang",
      color: "red",
      year: 1964
    };
  }
  changeColor = () => {
    this.setState({ color: "blue" });
  }
  render() {
    return (
      <div>
        <h1>My {this.state.brand}</h1>
        <p>
          It is a {this.state.color}
            {this.state.model}
            from {this.state.year}.
        </p>
        <button
          type="button"
          onClick={this.changeColor}
        >Change color</button>
      </div>
    );
  }
}
```

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

Output:-

My Ford

It is a red Mustang from 1964.

Change color

My Ford

It is a blue Mustang from 1964.

Change color

d. React Hook

Hooks were added to React in version 16.8.

Hooks allow function components to have access to state and other React features. Because of this, class components are generally no longer needed.

Although Hooks generally replace class components, there are no plans to remove classes from React.

What is a Hook?

Hooks allow us to "hook" into React features such as state and lifecycle methods.

Example:

Here is an example of a Hook.

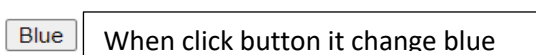
```
import React, { useState } from "react";
import ReactDOM from "react-dom/client";

function FavoriteColor() {
  const [color, setColor] = useState("red");
  return (
    <div>
      <h1>My favorite color is {color}!</h1>
      <button
        type="button"
        onClick={() => setColor("blue")}
      >Blue</button>
    </div>
  );
}

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

Output:-

My favorite color is green!



You must **import** Hooks from **react**.

Here we are using the **useState** Hook to keep track of the application state.

State generally refers to application data or properties that need to be tracked.

Hook Rules

There are 3 rules for hooks:

- Hooks can only be called inside React function components.
- Hooks can only be called at the top level of a component.
- Hooks cannot be conditional

Note: Hooks will not work in React class components.

6. React Props:-

Defination:-

Props stand for "**Properties**." They are **read-only** components. It is an object which stores the value of attributes of a tag and work similar to the HTML attributes. It gives a way to pass data from one component to other components. It is similar to function arguments. Props are passed to the component in the same way as arguments passed in a function.

Props are **immutable** so we cannot modify the props from inside the component. Inside the components, we can add attributes called props. These attributes are available in the component as **this.props** and can be used to render dynamic data in our render method.

When you need immutable data in the component, you have to add props to **ReactDOM.render()** method in the **index.js** file of your ReactJS project and used it inside the component in which you need.

Syntax:

// Passing Props

```
<DemoComponent sampleProp = "HelloProp" />
```

Syntax:

//Accessing props

```
this.props.propName;
```

Example:-

```
import React from 'react';
```

```
import ReactDOM from 'react-dom/client';
```

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

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

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

Output:-

Who lives in my Garage?

I am a Mustang!

7. React Forms

a. Defination:-

React Forms are the components **used to collect and manage the user inputs**. These components includes the input elements like text field, check box, date input, dropdowns etc. In [HTML forms](#) the data is usually handled by the DOM itself but in the case of React Forms data is handled by the react components.

React forms are the way to collect the user data in a React application. React typically utilize controlled components to manage form state and handle user input changes efficiently. It provides additional functionality such as preventing the default behavior of the form which refreshes the browser after the form is submitted.

In React Forms, all the form data is stored in the React's component state, so it can handle the form submission and retrieve data that the user entered. To do this we use controlled components.

Controlled Components

In simple HTML elements like input tags, the value of the input field is changed whenever the user type. But, In React, whatever the value the user types we save it in state and pass the same value to the input tag as its value, so here DOM does not change its value, it is controlled by react state. These are known as **Controlled Components**.

b. Creating Form:-

React offers a stateful, reactive approach to build a form. The component rather than the DOM usually handles the React form. In React, the form is usually implemented by using controlled components.

There are mainly two types of form input in React.

1. Uncontrolled component
2. Controlled component

1.Uncontrolled component

The uncontrolled input is similar to the traditional HTML form inputs. The DOM itself handles the form data. Here, the HTML elements maintain their own state that will be updated when the input value changes. To write an uncontrolled component, you need to use a ref to get form values from the DOM. In other words, there is no need to write an event handler for every state update. You can use a ref to access the input field value of the form from the DOM.

Example

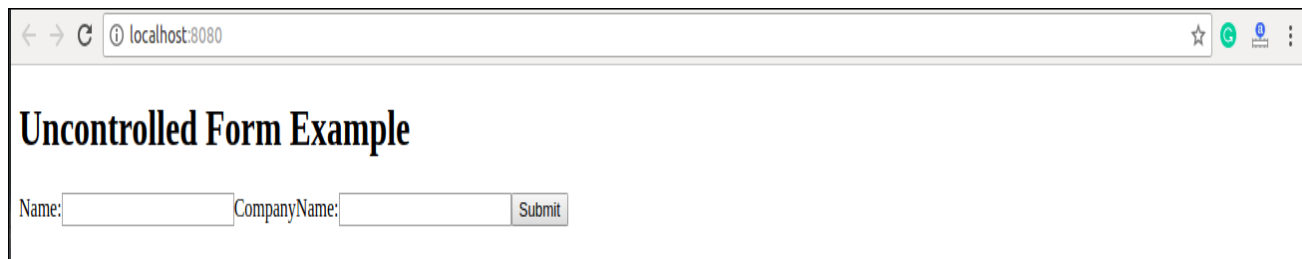
In this example, the code accepts a field **username** and **company name** in an uncontrolled component.

1. `import React, { Component } from 'react';`
2. `class App extends React.Component {`
3. `constructor(props) {`
4. `super(props);`
5. `this.updateSubmit = this.updateSubmit.bind(this);`
6. `this.input = React.createRef();`
7. `}`
8. `updateSubmit(event) {`
9. `alert('You have entered the UserName and CompanyName successfully.');`

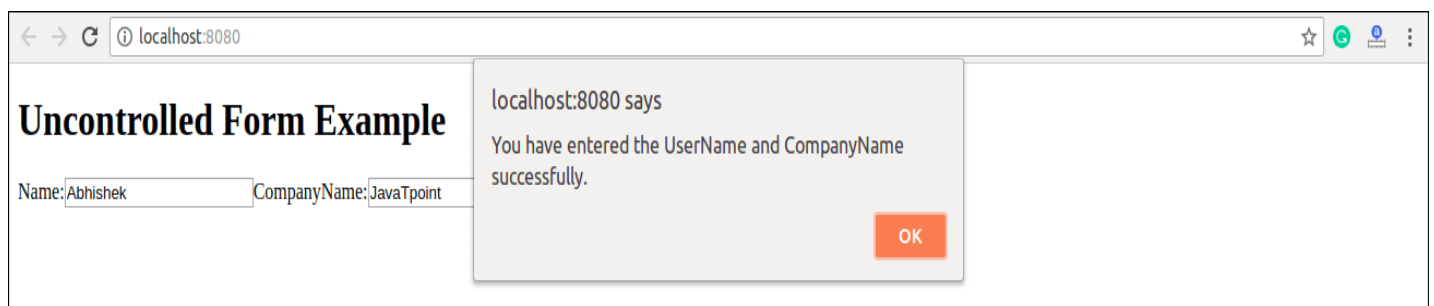
```
10.     event.preventDefault();
11. }
12. render() {
13.     return (
14.         <form onSubmit={this.updateSubmit}>
15.             <h1>Uncontrolled Form Example</h1>
16.             <label>Name:
17.                 <input type="text" ref={this.input} />
18.             </label>
19.             <label>
20.                 CompanyName:
21.                 <input type="text" ref={this.input} />
22.             </label>
23.             <input type="submit" value="Submit" />
24.         </form>
25.     );
26. }
27. }
28. export default App;
```

Output

When you execute the above code, you will see the following screen.



After filling the data in the field, you get the message that can be seen in the below screen.



2.Controlled Component

In HTML, form elements typically maintain their own state and update it according to the user input. In the controlled component, the input form element is handled by the component rather than the DOM. Here, the mutable state is kept in the state property and will be updated only with **setState()** method.

Controlled components have functions that govern the data passing into them on every **onChange event**, rather than grabbing the data only once, e.g., when you click a **submit button**. This data is then saved to state and updated with `setState()` method. This makes component have better control over the form elements and data.

A controlled component takes its current value through **props** and notifies the changes through **callbacks** like an `onChange` event. A parent component "controls" this changes by handling the callback and managing its own state and then passing the new values as props to the controlled component. It is also called as a "dumb component."

Example

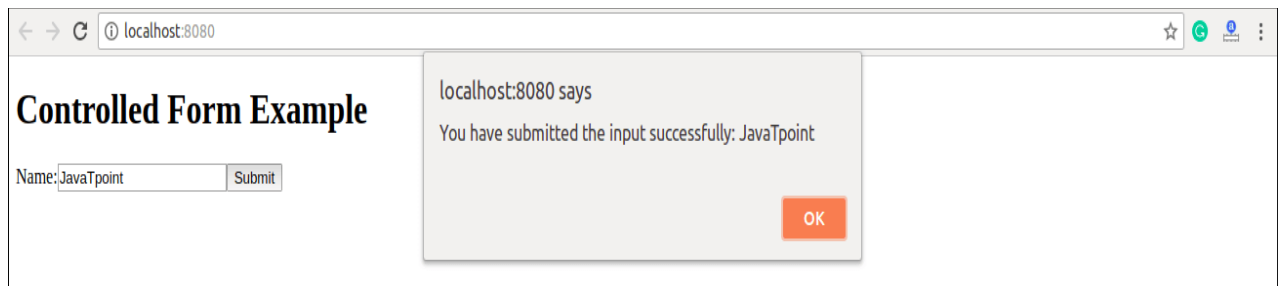
```
1. import React, { Component } from 'react';
2. class App extends React.Component {
3.   constructor(props) {
4.     super(props);
5.     this.state = { value: "" };
6.     this.handleChange = this.handleChange.bind(this);
7.     this.handleSubmit = this.handleSubmit.bind(this);
8.   }
9.   handleChange(event) {
10.    this.setState({ value: event.target.value });
11.  }
12.  handleSubmit(event) {
13.    alert('You have submitted the input successfully: ' + this.state.value);
14.    event.preventDefault();
15.  }
16.  render() {
17.    return (
18.      <form onSubmit={this.handleSubmit}>
19.        <h1>Controlled Form Example</h1>
20.        <label>
21.          Name:
22.          <input type="text" value={this.state.value} onChange={this.handleChange} />
23.        </label>
24.        <input type="submit" value="Submit" />
25.      </form>
26.    );
27.  }
28. }
29. export default App;
```

Output:-

When you execute the above code, you will see the following screen.



After filling the data in the field, you get the message that can be seen in the below screen.



(or) controlled form

Submitting Forms:-

We can use the `useState` Hook to keep track of each inputs value and provide a "single source of truth" for the entire application.

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

Example:-

```
import { useState } from "react";
import ReactDOM from 'react-dom/client';

function MyForm() {
  const [name, setName] = useState("");

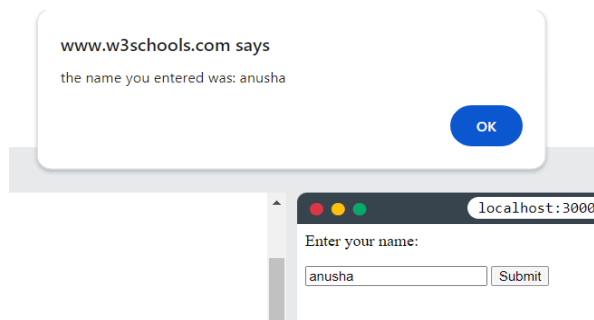
  const handleSubmit = (event) => {
    event.preventDefault();
    alert(`The name you entered was: ${name}`);
  }

  return (
    <form onSubmit={handleSubmit}>
      <label>Enter your name:
        <input
          type="text"
          value={name}
          onChange={(e) => setName(e.target.value)}
        />
      </label>
      <input type="submit" />
    </form>
  )
}

const root = ReactDOM.createRoot(document.getElementById('root'));
```

```
root.render(<MyForm />);
```

Output:-



c. Multiple Input Fields:-

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

We will initialize our state with an empty object.

To access the fields in the event handler use the `event.target.name` and `event.target.value` syntax.

To update the state, use square brackets [bracket notation] around the property name.

Example:

```
import { useState } from "react";
import ReactDOM from "react-dom/client";

function MyForm() {
  const [inputs, setInputs] = useState({});

  const handleChange = (event) => {
    const name = event.target.name;
    const value = event.target.value;
    setInputs(values => ({...values, [name]: value}))
  }

  const handleSubmit = (event) => {
    event.preventDefault();
    console.log(inputs);
  }

  return (
    <form onSubmit={handleSubmit}>
      <label>Enter your name:
      <input
        type="text"
        name="username"
        value={inputs.username || ""}
        onChange={handleChange}
      />
    </label>
    <label>Enter your age:
    <input
```

```

        type="number"
        name="age"
        value={inputs.age || ""}
        onChange={handleChange}
      />
    </label>
    <input type="submit" />
  </form>
)
}

```

```

const root = ReactDOM.createRoot(document.getElementById('root'));
root.render(<MyForm />);

```

```

/*
Click F12 and navigate to the "Console view"
to see the result when you submit the form.
*/

```

Output:-

Enter your name: Enter your age:

Note: We use the same event handler function for both input fields, we could write one event handler for each, but this gives us much cleaner code and is the preferred way in React.

d. Textarea:-

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

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

```

<textarea>

  Content of the textarea.

</textarea>

```

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

Example:

```

import { useState } from "react";
import ReactDOM from "react-dom/client";

function MyForm() {
  const [textarea, setTextarea] = useState(
    "The content of a textarea goes in the value attribute"
  );

  const handleChange = (event) => {
    setTextarea(event.target.value)
  }

  return (

```

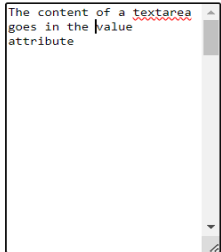
```

    <form>
      <textarea value={textarea} onChange={handleChange} />
    </form>
  )
}

const root = ReactDOM.createRoot(document.getElementById('root'));
root.render(<MyForm />);

```

Output:-



e. Select:-

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

in HTML, the selected value in the drop down list was defined with the `selected` attribute:

HTML:

```

<select>
  <option value="Ford">Ford</option>
  <option value="Volvo" selected>Volvo</option>
  <option value="Fiat">Fiat</option>
</select>

```

In React, the selected value is defined with a `value` attribute on the `select` tag:

Example:

A simple select box, where the selected value "Volvo" is initialized in the constructor:

```

import { useState } from "react";
import ReactDOM from "react-dom/client";

function MyForm() {
  const [myCar, setMyCar] = useState("Volvo");

  const handleChange = (event) => {
    setMyCar(event.target.value)
  }
}

```

```
return (  
  <form>  
    <select value={myCar} onChange={handleChange}>  
      <option value="Ford">Ford</option>  
      <option value="Volvo">Volvo</option>  
      <option value="Fiat">Fiat</option>  
    </select>  
  </form>  
)  
}  
  
const root = ReactDOM.createRoot(document.getElementById('root'));  
root.render(<MyForm />);
```

By making these slight changes to `<textarea>` and `<select>`, React is able to handle all input elements in the same way.

Output:-



Volvo ▼

Ford

Volvo

Fiat

8. Component Life-Cycle:-

In React, components have a lifecycle that consists of different phases. Each phase has a set of lifecycle methods that are called at specific points in the component's lifecycle. These methods allow you to control the component's behavior and perform specific actions at different stages of its lifecycle.

The lifecycle of the component is divided into **four phases**. They are:

1. Initial Phase
2. Mounting Phase
3. Updating Phase
4. Unmounting Phase

1. Initial Phase

It is the **birth** phase of the lifecycle of a ReactJS component. Here, the component starts its journey on a way to the DOM. In this phase, a component contains the default Props and initial State. These default properties are done in the constructor of a component. The initial phase only occurs once and consists of the following methods.

Example:-

```
class Clock extends React.Component {
  constructor(props)
  {
    // Calling the constructor of
    // Parent Class React.Component
    super(props);

    // Setting the initial state
    this.state = { date : new Date() };
  }
}
```

2. Mounting

Mounting means putting elements into the DOM.

React has four built-in methods that gets called, in this order, when mounting a component:

- **constructor()**
- **getDerivedStateFromProps()**
- **render()**
- **componentDidMount()**

The **render()** method is required and will always be called, the others are optional and will be called if you define them.

- **Constructor:-**

Method to initialize state and bind methods. Executed before the component is mounted.

Example:-

```
import React from 'react';
import ReactDOM from 'react-dom/client';

class Header extends React.Component {
  constructor(props) {
    super(props);
    this.state = { favoritecolor: "red" };
  }
  render() {
    return (
      <h1>My Favorite Color is {this.state.favoritecolor}</h1>
    );
  }
}

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

Output:-

My Favorite Color is red

- **Static getDerivedStateFromProps:**

Used for updating the state based on props. Executed before every render.

Example:-

```
import React from 'react';
import ReactDOM from 'react-dom/client';

class Header extends React.Component {
  constructor(props) {
    super(props);
    this.state = { favoritecolor: "red" };
  }
  static getDerivedStateFromProps(props, state) {
    return { favoritecolor: props.favcol };
  }
  render() {
    return (
      <h1>My Favorite Color is {this.state.favoritecolor}</h1>
    );
  }
}

const root = ReactDOM.createRoot(document.getElementById('root'));
root.render(<Header favcol="yellow"/>);
```

- **Render**

Responsible for rendering JSX and updating the DOM.

Example:-

```
import React from 'react';
import ReactDOM from 'react-dom/client';

class Header extends React.Component {
```

```

render() {
  return (
    <h1>This is the content of the Header component</h1>
  );
}
}

```

```

const root = ReactDOM.createRoot(document.getElementById('root'));
root.render(<Header />);

```

Output:-

This is the content of the Header component

• componentDidMount

The `componentDidMount()` method is called after the component is rendered.

This is where you run statements that requires that the component is already placed in the DOM.

Example:-

```

import React from 'react';
import ReactDOM from 'react-dom/client';

class Header extends React.Component {
  constructor(props) {
    super(props);
    this.state = { favoritecolor: "red" };
  }
  componentDidMount() {
    setTimeout(() => {
      this.setState({ favoritecolor: "yellow" })
    }, 1000)
  }
  render() {
    return (
      <h1>My Favorite Color is {this.state.favoritecolor}</h1>
    );
  }
}

```

```

const root = ReactDOM.createRoot(document.getElementById('root'));

root.render(<Header />);

```

Output:-

My Favorite Color is yellow

3. Updating

The next phase in the lifecycle is when a component is *updated*.

A component is updated whenever there is a change in the component's **state** or **props**.

React has five built-in methods that gets called, in this order, when a component is updated:

1. **getDerivedStateFromProps()**
2. **shouldComponentUpdate()**
3. **render()**
4. **getSnapshotBeforeUpdate()**
5. **componentDidUpdate()**

The **render()** method is required and will always be called, the others are optional and will be called if you define them.

- **getDerivedStateFromProps:-**

`getDerivedStateFromProps(props, state)` is a static method that is called just before `render()` method in both mounting and updating phase in React. It takes updated props and the current state as arguments.

Example:-

```
import React from 'react';
import ReactDOM from 'react-dom/client';

class Header extends React.Component {
  constructor(props) {
    super(props);
    this.state = { favoritecolor: "red" };
  }
  static getDerivedStateFromProps(props, state) {
    return { favoritecolor: props.favcol };
  }
  changeColor = () => {
    this.setState({ favoritecolor: "blue" });
  }
  render() {
    return (
      <div>
        <h1>My Favorite Color is { this.state.favoritecolor }</h1>
        <button type="button" onClick={this.changeColor}>Change color</button>
      </div>
    );
  }
}

const root = ReactDOM.createRoot(document.getElementById('root'));
root.render(<Header favcol="yellow" />);
```

/*

This example has a button that changes the favorite color to blue, but since the `getDerivedStateFromProps()` method is called, the favorite color is still rendered as yellow (because the method updates the state

with the color from the favcol attribute).

*/

Output:-

My Favorite Color is yellow

Change color

• shouldComponentUpdate:-

In the `shouldComponentUpdate()` method you can return a Boolean value that specifies whether React should continue with the rendering or not.

The default value is `true`.

Example:-

```
import React from 'react';
import ReactDOM from 'react-dom/client';

class Header extends React.Component {
  constructor(props) {
    super(props);
    this.state = { favoritecolor: "red" };
  }
  shouldComponentUpdate() {
    return true;
  }
  changeColor = () => {
    this.setState({ favoritecolor: "blue" });
  }
  render() {
    return (
      <div>
        <h1>My Favorite Color is {this.state.favoritecolor}</h1>
        <button type="button" onClick={this.changeColor}>Change color</button>
      </div>
    );
  }
}

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

Output:-

My Favorite Color is red

Change color

Fig.Before updation

My Favorite Color is blue

Change color

Fig.After updation

Note:-if shouldComponent is return false means no execution.

- **getSnapshotBeforeUpdate:-**

In the `getSnapshotBeforeUpdate()` method you have access to the `props` and `state` *before* the update, meaning that even after the update, you can check what the values were *before* the update.

If the `getSnapshotBeforeUpdate()` method is present, you should also include the `componentDidUpdate()` method, otherwise you will get an error.

The example below might seem complicated, but all it does is this:

When the component is *mounting* it is rendered with the favorite color "red".

When the component *has been mounted*, a timer changes the state, and after one second, the favorite color becomes "yellow".

This action triggers the *update* phase, and since this component has a `getSnapshotBeforeUpdate()` method, this method is executed, and writes a message to the empty DIV1 element.

Then the `componentDidUpdate()` method is executed and writes a message in the empty DIV2 element:

Example:-

```
import React from 'react';
import ReactDOM from 'react-dom/client';

class Header extends React.Component {
  constructor(props) {
    super(props);
    this.state = { favoritecolor: "red" };
  }
  componentDidMount() {
    setTimeout(() => {
      this.setState({ favoritecolor: "yellow" })
    }, 1000)
  }
  getSnapshotBeforeUpdate(prevProps, prevState) {
    document.getElementById("div1").innerHTML =
      "Before the update, the favorite was " + prevState.favoritecolor;
  }
  componentDidUpdate() {
    document.getElementById("div2").innerHTML =
      "The updated favorite is " + this.state.favoritecolor;
  }
  render() {
    return (
      <div>
        <h1>My Favorite Color is {this.state.favoritecolor}</h1>
        <div id="div1"></div>
        <div id="div2"></div>
      </div>
    );
  }
}

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

Output:-

My Favorite Color is yellow

Before the update, the favorite was red

The updated favorite is yellow

- **componentDidUpdate**

The `componentDidUpdate` method is called after the component is updated in the DOM.

The example below might seem complicated, but all it does is this:

When the component is *mounting* it is rendered with the favorite color "red".

When the component *has been mounted*, a timer changes the state, and the color becomes "yellow".

This action triggers the *update* phase, and since this component has a `componentDidUpdate` method, this method is executed and writes a message in the empty DIV element:

Example:-

```
import React from 'react';
import ReactDOM from 'react-dom/client';

class Header extends React.Component {
  constructor(props) {
    super(props);
    this.state = { favoritecolor: "red" };
  }
  componentDidMount() {
    setTimeout(() => {
      this.setState({ favoritecolor: "yellow" })
    }, 1000)
  }
  componentDidUpdate() {
    document.getElementById("mydiv").innerHTML =
      "The updated favorite is " + this.state.favoritecolor;
  }
  render() {
    return (
      <div>
        <h1>My Favorite Color is {this.state.favoritecolor}</h1>
        <div id="mydiv"></div>
      </div>
    );
  }
}

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

Output:-

My Favorite Color is yellow

The updated favorite is yellow

4. Unmounting

The next phase in the lifecycle is when a component is removed from the DOM, or *unmounting* as React likes to call it.

React has only one built-in method that gets called when a component is unmounted:

- `componentWillUnmount()`
- `componentWillUnmount`

The `componentWillUnmount` method is called when the component is about to be removed from the DOM.

Example:-

```
import React from 'react';
import ReactDOM from 'react-dom/client';

class Container extends React.Component {
  constructor(props) {
    super(props);
    this.state = { show: true };
  }
  delHeader = () => {
    this.setState({ show: false });
  }
  render() {
    let myheader;
    if (this.state.show) {
      myheader = <Child />;
    };
    return (
      <div>
        {myheader}
        <button type="button" onClick={this.delHeader}>Delete Header</button>
      </div>
    );
  }
}

class Child extends React.Component {
  componentWillUnmount() {
    alert("The component named Header is about to be unmounted.");
  }
  render() {
    return (
      <h1>Hello World!</h1>
    );
  }
}
```



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

Output:-

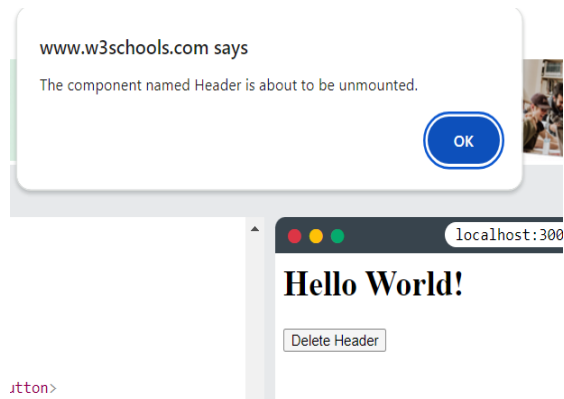
Hello World!

Delete Header

Fig.Before deleting and after deleting alert messege

Delete Header

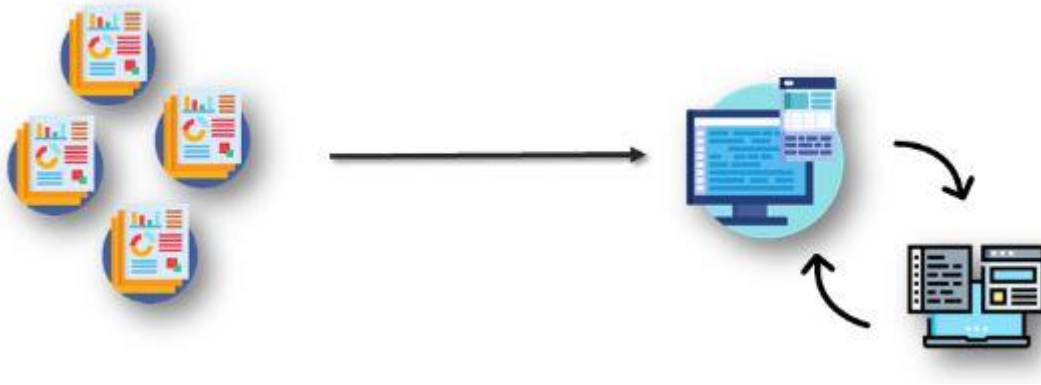
Fig.After Deleting data removed



9. React Redux:-

a. Why Redux?

State transfer between components is pretty messy in React since it is hard to keep track of which component the data is coming from. It becomes really complicated if users are working with a large number of states within an application.



Redux solves the state transfer problem by storing all of the states in a single place called a store. So, managing and transferring states becomes easier as all the states are stored in the same convenient store. Every component in the application can then directly access the required state from that store.

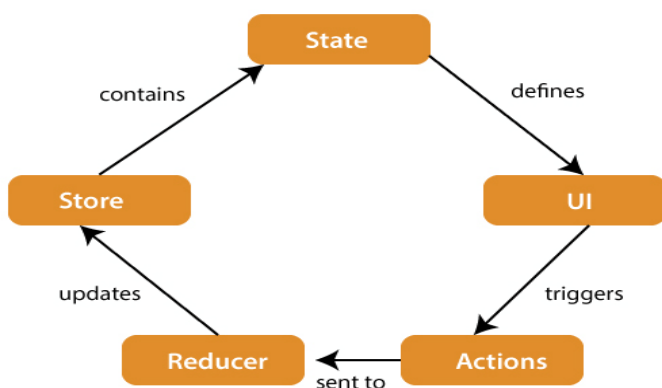
b. Defination:-

Redux is an open-source JavaScript library used to manage application state. React uses Redux for building the user interface. It was first introduced by **Dan Abramov** and **Andrew Clark** in **2015**.

React Redux is the official React binding for Redux. It allows React components to read data from a Redux Store, and dispatch **Actions** to the **Store** to update data. Redux helps apps to scale by providing a sensible way to manage state through a unidirectional data flow model. React Redux is conceptually simple. It subscribes to the Redux store, checks to see if the data which your component wants have changed, and re-renders your component.

Redux was inspired by Flux. Redux studied the Flux architecture and omitted unnecessary complexity.

c. Redux Architecture



The components of Redux architecture are explained below.

STORE: A Store is a place where the entire state of your application lists. It manages the status of the application and has a `dispatch(action)` function. It is like a brain responsible for all moving parts in Redux.

ACTION: Action is sent or dispatched from the view which are payloads that can be read by Reducers. It is a pure object created to store the information of the user's event. It includes information such as type of action, time of occurrence, location of occurrence, its coordinates, and which state it aims to change.

REDUCER: Reducer read the payloads from the actions and then updates the store via the state accordingly. It is a pure function to return a new state from the initial state.

d. Redux Installation

Requirements: React Redux requires React 16.8.3 or later version.

To use React Redux with React application, you need to install the below command.

- `npm install redux`
- `npm install react-redux`

Example Code For Redux Store:-

```
import {createStore} from "redux";
const initialState={
  balance:0,
  fullName:"",
  mobile:,
}
function accountReducer(state=initialState,action){
  if(action.type==="deposit")
  {
    return {...state,balance:state.balance+ +action.payload};
  }
  else if(action.type==="withdraw")
  {
    return {...state,balance:state.balance- +action.payload};
  }
  else if(action.type==="mobileUpdate")
  {
    return {...state,mobile:action.payload};
  }
  else if(action.type==="nameUpdate")
  {
    return {...state,fullName:action.payload};
  }
  else
  return state
}

const store=createStore(accountReducer)
```

```

console.log(store.getState());
store.dispatch({type:"deposit",payload:1000})
console.log(store.getState());
store.dispatch({type:"withdraw",payload:100})
console.log(store.getState());
store.dispatch({type:"mobileUpdate",payload:8547961256})
console.log(store.getState());
store.dispatch({type:"nameUpdate",payload:"swathi"})
console.log(store.getState());

```

Output:-

```

balance:900,
  fullName:" swathi ",
  mobile: 8547961256,

```

e. Example Code:-Creating store and components Accessing data from store

React Redux includes a `<Provider />` component, which makes the Redux store available to the rest of your app components:

- **Index.js**

```

import React from 'react';
import ReactDOM from 'react-dom/client';
import './index.css';
import App from './App';
import { Provider } from 'react-redux';
import store from './store';
import reportWebVitals from './reportWebVitals';

const root = ReactDOM.createRoot(document.getElementById('root'));
root.render(
  <React.StrictMode>
    <Provider store={store}>
      <App />
    </Provider>

  </React.StrictMode>
);

```

- Rendering form and account components
- **App.js**

```

import Form from './Forms';
import Account from './Account';
function App() {
  return (
    <>

```

```

    <Form/>
    <Account/>
  </>
);
}

```

export default App;

- Creating state Store data
- **Store.js**

```

import {createStore} from "redux";
const initialState={
  balance:0,
  fullName:"Anusha",
  mobile:7854789689,
}
function accountReducer(state=initialState,action){
  if(action.type==="deposit")
  {
    return {...state,balance:state.balance+ +action.payload};
  }
  else if(action.type==="withdraw")
  {
    return {...state,balance:state.balance- +action.payload};
  }
  else if(action.type==="mobileUpdate")
  {
    return {...state,mobile:action.payload};
  }
  else if(action.type==="nameUpdate")
  {
    return {...state,fullName:action.payload};
  }
  else
  return state
}

const store=createStore(accountReducer)

export default store;

```

- **Hooks**

React Redux provides a pair of custom React hooks that allow your React components to interact with the Redux store.

`useSelector` reads a value from the store state and subscribes to updates,
while `useDispatch` returns the store's `dispatch` method to let you dispatch actions.

- **Account.js**

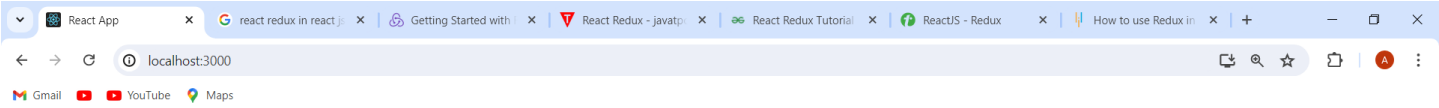
```
import { useSelector } from "react-redux";

function Account(){
  let data= useSelector(
    (state)=>{
return state;
    }
  )
  return <>
    <div>
      <h1>Account Details</h1>
      <h3>{data.balance}</h3>
      <h3>{data.fullName}</h3>
      <h3>{data.mobile}</h3>
    </div>
  </>;
}
export default Account;
```

- **Form.js**

```
import { useState } from "react";
import { useDispatch } from "react-redux";
function Form(){
  let dispatch = useDispatch();
  const [amount, setAmount]=useState("");
  return <>
    <div>
      <h1>Form Component</h1>
      <input type="number" placeholder="Enter Amount"
        value={amount}
        onChange={(e)=>{
          let data=e.target.value;
          setAmount(data);
        }}
      />
      <button onClick={()=>{
        dispatch({type:"deposit",payload:amount})
      }}>change amount</button>
    </div>
  </>;
}
export default Form;
```

Output:-



Form Component

change amount

Account Details

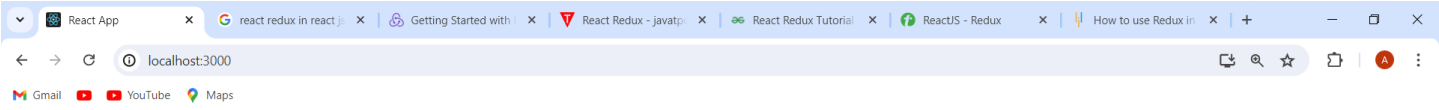
0

Anusha

7854789689



Fig. Before Updating Data



Form Component

change amount

Account Details

1000

Anusha

7854789689



Fig. After Updating Data

10. Angular vs React JS:-

What is React?

React is a front-end JavaScript library that allows you to build user interfaces from reusable UI components. React uses server-side rendering to provide a flexible and performance-based solution. It allows developers to create seamless UX and complex UI.

What is Angular?

Angular is an open-source JavaScript front-end framework developed and managed by Google's Angular team. Angular is the most popular client-side framework for developing scalable and high-performing mobile and web apps using HTML, CSS, and TypeScript. The latest version of Angular is Angular 13, which offers enterprise-ready web app development solutions and is widely used by companies for web development.

Parameters	Angular	React
Developed By	Google	Facebook
Release Year	2009	2013
Written In	TypeScript	JavaScript
Technology Type	Full-fledged MVC framework written in JavaScriptJavaScript	library (View in MVC; requires Flux to implement architecture)
Concept	Brings JavaScript into HTML Works with the real DOM Client-side rendering	Brings HTML into JavaScript Works with the virtual DOM Server-side rendering
Data Binding	Two-way data binding	One-way data binding
Language	JavaScript + HTML	JavaScript + JSX
Learning Curve	Steep	Moderate
UI Rendering	Client/Server-Side	Client/Server-Side
Best Suited For	Highly active and interactive web apps	Larger apps with recurrent variable data
App Structure	Fixes and complicated MVC	Flexible component-based view
Dependency Injection	Fully supported	Not supported
Performance	High	High
DOM Type	Real	Virtual
Popular Apps	IBM, PayPal, Freelancer, Upwork	Facebook, Skype, Instagram, Walmart