What is react JS ?

React.js or React JS is a Java Script library used for building user interfaces, particularly for web applications with dynamic content. It helps developers create interactive and efficient ULs by allowing them to build reusable components. These components represent different parts of the user interface and can update dynamically as the data changes. React simplifies the process of managing and updating the user interface, making it easier to build complex and responsive web applications

- React.js is widely used in various real-world applications and websites. Some notable examples include:

1. Face book:

React was developed by Facebook, and it's extensively used throughout the platform for creating dynamic and responsive user interfaces.

2. Instagram :

Instagram's web interface is built using React, providing a seamless and interactive experienc for users.

3. Whats App Web :

React is used in the web version of Whats App, contributing to the responsiveness and user interface design.

4. Netflix :

Netflix employs React for parts of its user interface, helping to deliver a smooth and engaging streaming experience.

5. Airbnb :

React is used in Airbnb's user interfaces, making it easier to manage the complexity of their dynamic and interactive web pages.

As we learnt before that React JS is Javascript Library so what basically Library means? And why we compare Library with Framework always?

In real-world terms, think of a library as a collection of pre-written recipes (functions) that you can use when cooking (coding). You decide when and how to use these recipes in your own cooking process (programming).

Comparing a library to a framework is like the difference between getting a recipe book (Library) and attending a cooking class (Frame-work). With the book, you can choose the recipes you want and follow them. In the class, the chef (Framework) provides a structure and tells you when to use certain ingredients (functions).

A library gives you more freedom, allowing you to pick and choose what you need. In contrast,

a framework provides a more structure environment, dictating how your project should be organized and often offering tools to handle common tasks. It's like choosing between a DIY approach (Library) and following a guided recipe (Framework) for your coding project

How to install react js

To install React.js, you typically use the Node.js package manager, npm. Here's a step-by-step guide:

1- Install Node.js:

- Visit the official Node.js website: (https://nodejs.org/).
- Download and run the installer for your operating system.
- Follow the installation prompts.

2. Verify Node.js installation:

- Open a command prompt or terminal.
- Type `node -v` and `npm -v` to check if Node.js and npm are installed.

3. Create a React App:

- Open a command prompt or terminal.
- Run the following command to install the Create React App tool globally: `npm install -g create-react-app`.

4. Generate a React App:

- Run `npx create-react-app my-react-app` (replace "my-react-app" with your preferred project name).
- Change to the project directory: `cd my-react-app`

5- Start the Development Server:

- Run `npm start` to start the develop--ment server.
- This will open a new browser window with your React app running.

React .js Folder structure :

1. **`node_modules`:**

- contains project dependencies installed via npm. This folder is usually Large and is not tracked by version control.

2. **'public':**

- Contains the static assets and the HTML File ('index.html') that serves as the entry point to your React application.

3. **'src':**

- **'index.js':** The main Java Script file where React is rendered into the DOM. It typically imports the `App` component.
- **`App.js`:** The root component of your application, which may include other components.
- **'~~as~~components/':** A directory to organize your React components.

- **'assets/': ** Stores images, fonts, or other static assets used in your project.
- **'styles/': ** Holds CSS Files or other styling solutions for your components.
- **'utils/': ** If applicable; a place for utility functions or helper modules.

4. **'Public' (continued): **

- **'Favicon.ico': ** The icon that appears in the browser tab.
- **'manifest.json': ** A JSON File that provides metadata about the web application.

5. **'.gitignore': **

- Specifies files and directories that should be ignored by version control (e.g., 'node_modules').

6. **'package.json': **
- Contains metadata about the project and lists its dependencies. Also

includes scripts for various tasks like starting the development server or building the project.

7. **`package-lock.json`:**

- Records the exact versions of installed npm packages to ensure consistency across different development environments.

8. **`README.md`:**

- Documentation file providing information about the project, how to set it up, and other relevant details.

9. **`yarn.lock` (optional):**

- Generated by the yarn package manager if you use it instead of npm. It ensures that dependencies are installed consistently across different machines.

React JSx

JSX is like a special language in React that helps us describe how things should look on a website. It's a mix of JavaScript and HTML, making it easier for us to create and understand what our web page will show. So, when we write code with JSX, React can understand and show the right stuff on the screen. It's like giving instructions to React in a way that's both friendly for humans and the computer.

Open src/App.js in your code editor://src/App.js

```
import React from 'react';

Function App () {
 return (
  <div>
   <h1> Hello, JSX ! </h1>
   <p> This is a React app using JSX.</
p>
```

```
  </div>
  );
}

export default App;
```

In this code, observe the 'return' section where we've written '<h1> Hello, JSX! </h1>' and This is React app using JSX. . The syntax resembles HTML, and you might find it familiar. This is JSX, a blend of JavaScript and HTML, allowing us to express UI elements more intuitively within our JavaScript code. So, essentially, we're ~~combing~~ combining the power of JavaScript with the structure and simplicity of HTML to define how our React component should render on the webpage.

We can also add CSS like this:

```
import React from 'react';
function App () {
  return (
  <div style = {{ color : 'blue', Font Size :
'20 px', text Align : 'center' }} >
   <h 1 > Hello , JSX with Inline CSS !</h1>
   <p> This is a React app with inline
   styles . </p>
      </ div >
  );
 }

  export default App ;
```

In this example, the style attribute takes an object where the keys are CSS property names in CamelCase, and the values are the corresponding styles.
Note that the style value is an object, and the double curly braces ({{}}) are use to embed this JavaScript object within JSX.

We can also add external css like this:

```
import React from 'react';
import './Your External Styles.css';

function App () {
 return (
  <div class Name = "hero" >
   <h1 class Name = "hero"> Hello, JSX
with External CSS! </h1>
  <p> This is a React app with external
  styles. </p>
    </div>
 );
 }

  export default App;
```

Make File is src Folder (File name):

```
Your External Styles.css
//write as normal CSS you know before

. hero {
background - color : red;
}
```

"Note":
To include external CSS in a React Component, import the CSS file at the beginning of your component file.

Ensure the path to the CSS file is correct; in this example, it's assumed to be in the same directory.

Let see how to write JavaScript with JSX

In React JSX, you can embed JavaScript expressions and code using curly braces {}. This allows you to dynamically generated content, perform logic, and interact with variables within your JSX code. Here's an example:

```
import React from 'react';

function App () {
 Const name = 'John Doe';
 Const greeting = <p> Hello, {name}! </p>;

 return (
```

```
  <div>
   {greeting}
   <p> Today is {new
Date ().toLocaleDateString()}.</p>
   <p> {2+2} equals 4.</p>
   </div>
  );
}

export default App;
```

In this example:

{name} inserts the value of the name variable into the JSX.
{new Date().toLocaleDateString()} dynamically generates the current date.
{2+2} performs a JavaScript calculatio with in JSX.

Remember to use curly braces only for JavaScript expressions and not for statemen or blocks. If you need to include a block of JavaScript code, you might want to do that outside the JSX, typically within the functional component.

"Components"

In React, a component is a reusable and self-contained piece of code that defines how a part of the user interface should appear and behave.

Lets see with example

Steps to create Components:

1st: Make a file inside src folder or any folder (in our case we are make it inside src)

Write any relatable file name and we usually start component name with capital Letter and add extension js or

jsx with it.
So in our case we write

"My Component.jsx"

2nd: Code it

```
import React from 'react';

function My Component () {
 return (
<p> Hello, World! </p>
);
}

Export default My Component;
```

Here's a breakdown of the code:

import React from 'react'; : Imports the React library, which is necessary for writing React components.

function My Component () { : Defines a functional component named My Component.

return (: Begins the return statement,

indicating what the component will render.

 Hello, World ! : Represents a paragraph element with the text "Hello, World !".

); : Ends the return statement.

} : Closes the function block.

export default My Component; : Exports the My Component function as the default export from this module, making it available for use in other files or components.

3rd : now import it where you want to use it

Certainly ! If you want to use the My Components inside App. js, you would import it and then include it within the App component. Here's an example :

```
import React from 'react';
import My Component from './
My Component';
// As
My Component.js is in the same directory

function App ()
return (
   <div>
<h1> My React App </h1>
 < My Component />
</div>
);
}

export default App;
```

In this example:

import My Component from './ My Component'; imports the My Component functional component from the file where it is defined (My Component.js).

< My Components /> includes the My Componen component withinthe JSX of the App Component

Exercise :

1. Create a `Header` Component :
   - Create a new React functional component called `Header`.
   - inside `Header`, use JSX to display a header with a title of your choice.
   - Export the `Header` component.

2: Create a footer Component :
   - Create another functional component called `Footer`.
   - inside `Footer`, use JSX to display a footer with a message like "© My React App".
   - Export the `Footer` component.

3. Update UserInfo Component :
   - Enhance the `UserInfo` component to include additional information like the user's email and a profile picture.
   - Update the exported `UserInfo` component.

4. Modify App Component :
   - Import the `Header`, `UserInfo`,

and `Footer` components into the `App` component.
- Use JSX to include these components within the `App` Component.
- Export the updated `App` Component.

5. Render `App` inside index.js

What is Props ?

In React, "props" is short for properties, and it's a way to pass data from one component to another. Imagine it as a messager carrying information from a parent to its child.

Let' say you have a Parent component and a Child component:

```
// Parent.js
import React from 'react';
import Child from './Child';

function Parent (){
  const message ="Hello from Parent!";
return (
  <Child greeting ={message}/>
);
}

export default Parent;


// Child.js
import React from 'react';
```

```
function Child (props) {
 return (
  <div>
    <p> {props . greeting } </p>
  < div >
 );
}

export default Child;
```

In this example:

The Parent component has a variable message.
It passes this message to the Child component using the greeting prop.
The Child component then displays the recieved message using props greeting.

Let us create a simple Movie Website with Knowledge which we learn

Lets us first create Components

1st : Movie Card.js
For making card

2nd : Movie List.js
For List of Movies

3rd : Movie.css
For Styling

```
// Movie List.js
import Reach From 'react';
import Movie Card from './Movie Card';

function Movie List () {
 const movies = [
  {
   title: 'Inception',
   genre: 'Sci-fi',
   year: 2010,
   image Url: 'https://example.com
 /inception.jpg',
  },
  {
   title: 'The Dark knight',
   genre: 'Action',
   year: 2008,
   image Url: 'https://example.com/dark
 -knight.jpg',
  },
  // Add more movies as needed
 ];

 return (
  <div>
   <h1> Popular Movies </h1>
   <div class Name = "movie-list">
```

```
        {movies.map ((movie, index) => (
          <MovieCard key={index}{...movie}/
>
        ))}
        </div>
      </div>
    );
}

export default MovieList;


// MovieCard.js
import React from 'react';
import './MovieCard.css';

function MovieCard (props) {
  const {title, genre, year, imageUrl} =
props;

return (
  <div class Name = "movie-card">
    <img src = {imageUrl} alt = {title}
class Name = "movie-image"/>
  <div class Name = "movie-details">
    <h2 class Name = "movie-title">{title}
</h2>
```

```
    <p class Name ="movie-info"> Genre:
{genre} </p>
    <p class Name = "movie-info"> Year:
{year} </p>
    </div>
    </div>
);
}

export default Movie Card;
```

→ /* Movie Card.css */

```
.movie-card {
  border : 1 px solid # ddd;
  borderRadius : 8 px;
  overflow : hidden;
  margin : 16 px;
  box Shadow : 0 4px 8px rgba (0,0,0,0.1)
  transition : transform 0.3 s;
}

.mOvie - card : hover {
  transform : scale (1.05);
}

.movie - image {
```

```
  width : 100% ;
  height : auto;
  border-bottom : 1px solid # ddd;
}

.movie-details {
  padding : 16px;
}

.movie-title {
  margin : 0;
  fontSize : 1.5em;
  color : # 333;
}

.movie-info {
  margin : 8px 0;
  color : # 666 ;
}
```

→ // App.js

```
import React from 'react';
import MovieList from './MovieList';

function App (){
  return (
```

```
    <div class Name = "app" >
      <Movie List />
    </div >
  );
}

export default App;
```

Try this code and make some changes by your self so you can learn better.

Exercises for Props

Here are exercises focusing on under-standing and using props in React:

1. * Simple Greeting Component : *

- Create a functional component named 'Greeting'.
- Pass a prop called 'name' to the component.
- Display a greeting message using the provided name.

2. * User Profile Card : *

- Create a component called 'UserProfile' that recieves props for 'username', 'avatar', and 'followers'.
- Display these user details in a simple card format.

3. * Product Card : *

- Build a component named 'Product Card'.
- Pass props for 'productName', 'price', and 'image'.

- Display these details in a card like format for a product.

4. * List Rendering with Props : *

- Create a components called `To do list`.
- Pass an array of to do items as a prop to this component.
- Render each to do items as a list.

5. * Interactive Button Component : *

- Develop a component named `interactive Button`.
- Pass a prop for the button label, e.g., `label`.
- Include an `on click` prop to handle a click event.
- Display the button with the provided label and trigger the click event.

* React Hooks in Java Script : *

React Hooks are functions that allow you to use state and other React features in functional components. They were introduced in React version 16.8 to provide a more direct way to work with stateful logic in functional components, previously only possible in class components.

* Advantages of React Hooks : *

1. Simplified state Management:
   - Before Hooks: Stateful Logic was managed in class components using `this.state` and `this.setState`.
   - With Hooks: `useState` allows functional components to managed state without needing a class.

2. Reuse of Stateful Logic:
   - Before Hooks: Reusing stateful logic required higher-order components or render props.
   - With Hooks: Custom hooks allow

you to extract and share stateful logic between components easily.

3. Cleaner and Readable Code:
- Before Hooks: Class components could become lengthy and harder to read.
- With Hooks: Functional components with hooks often result in more concise and readeable code.

4. Better Separation of Concerns:
- Before Hooks: Lifecycle methods were used for various concerns, leading to component logic spread across different methods.
- With Hooks: Each hook serves a specific concern (e.g., `useState` for state, `useEffect` for side effects), leading to cleaner separation.

5. No Need for Class Syntax:
- Before Hooks: State and lifecycle features required class components.
- With Hooks: Functional component can now handle all aspects of

state and Lifecycle management.

6. Improved code Reusability:
- Before Hooks : Reusable logic was often tied to class components.
- With Hooks : Custom hooks enable the creation of reusable pieces of logic that can be shared across multiple components.

7. Simplified Component Lifecycle:
- Before Hooks : Life cycle methods like 'component Did Mount' and 'component Did Update' handled side effects.
- With Hooks : 'use Effects' allows for a more unified and readable way to manage side effects.

8. Easier Testing :
- Before Hooks : Testing state ful logic in class components could be complex.
- With Hooks : Since hooks are function testing is often more straightforward with tools like Jest.

9. Improved Performance:
- Before Hooks : Class components might suffer from performance issues due to frequent re-renders.
- With Hooks : 'use Memo' and 'use Callback' help optimize performanc by memoizing values and functions.

React Hooks simplify and enhance the development experience in React by providing a more functional and modular approach to state and lifecycle management.

Don't worry if you don't understand any of these points we will understand each & and everything in easy way.

"use State"

useState is a React Hook that allows functional components to manage state.

It was introduced in React 16.8 as part of the Hooks API.

Importing useState :

Import useState from the 'react' library:

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

Syntax :

useState returns as array with two elements : the current state value and a function to update it.

The first etement is the current state, and the second element is a function to update the state.

```
const [state, setState] =
useState (initial State);
```

Initialization :

useState takes an initial state as an argument.

The initial state can be a primitive value, object, or a function returning the initial state.

```
Const [ count, setCount ] = useState (0);
```

Updating State :

Use the function returned by useState to update the state.

The state is updated asynchronously and React will batch multiple setSta calls for performance.

```
setCount (count + 1);
```

Functional Updates :

Use the functional form of setState when the new state defends on the previous state.

```
setCount (prevCount => prevCount + 1);
```

Multiple State Variables:

You can use useState multiple times to manage different pieces of state in a component.

```
const [name, setName] = useState("");
const [age, setAge] = useState (0);
```

Let's see a simple example of counter app:

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

function Counter (){
  const [count, setCount] = useState (0);

  return (
    <div>
      <p> Count : {count}</p>
      <button onClick = {() =>
      setCount + 1)}> Increment </
```

```
export default Counter;
```

1. Import Statements:
- `import React, { useState } from 'react';` imports the necessary modules from the 'react' library, including the `useState` hook.

2. Functional Component:
- `Function Counter () {` begins the definition of a functional component named 'Counter'.

3. State Initialization:
- `const [count, setCount] = useState (0);` uses the `useState` hook to declare a state variable named 'count' with an initial value of 0. The `setCount` function is a means to update the 'count' state.

4. JSX Structure:
- The `return` statement contains JSX, defining the structure of the component.

- `<div>` encapsulates the entire component.

5. Displaying Count :
- `<p> Count : {count} </p>` displays the current value of the 'count' state variable.

6. Increment Button :
- `<button onClick={ () => setCount (count + 1)} > Increment </button>` creates a button that, when clicked, triggers a function to increment the 'count' state variable by 1.

7. Export :
- `export default Counter;` exports the 'counter' component as the default export of this module, making it available for use in other files.

Exercises

1. * Beginner : Simple Counter *
- Create a functional component that displays a counter.
- Use the `useState` hook to manage the counter state.
- Include buttons on increment and decrement the counter.

2. * Beginner : Toggle Button *
- Build a component with a button that toggles between "ON" and "OFF" when clicked.
- Utilized the `useState` hook to manage the state of the button.

3. * Intermediate : Input Field *
- Develope a form component with an input field.
- Use the `useState` hook to manage the input's value.
- Display the entered value below th input.

4. * Intermediate : List Rendering *
- Create a component that renders a

List of items.
- Use `useState` to manage an array of items.
- Include a button to add new items to the list.

5. * Intermediate : Conditional Rendering *
- Build a component that conditionally renders content based on a boolean state.
- Use `useState` to toggle between two different views.

6. * Advanced : Dynamic forms *
- Develop a dynamic form where users can add and remove input fields.
- Use `useState` to manage an array of input values.

7. * Advanced : Controlled Components *
- Build a controlled from where each input field is directty linked to the component's state using `useState`.

8. * Advanced : Multi-step Wizard *
- Create a multi-step wizard component with navigation buttons.

- Use 'useState' to manage the step number and dynamically render the appropriate content.

9. * Advanced : Use of 'useEffect' *
   - Extend a previously exercise by introducing the 'useEffect' hook.
   - Fetch data from an API and update the component's state with the fetched data.

10. * Advanced : State Management with Reducer *
   - Rewrite an existing component using the 'useReducer' hook instead of 'useState'.
   - Manage more complex state logic using actions and a reducer function.

useEffect

The useEffect hook is used to perform side effects in functional components. Side effects may include data fetching, subscriptions, or manually changing the DOM.

Using `useEffect` to fetch data is a common scenario in React when dealing with asynchronous operations. Here's an example of how you might use `useEffect` to fetch data from an API

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

function DataFetchingExample () {
  const [ data , setData ] = useState (null
  const [ loading , setLoading ] =
useState (true);
  const [error, setError ] = useState (nul

useEffect (() => {
  // Function to fetch data
  const fetchData = async () => {
    try {
```

```
    // Set loading to true while data is
being fetched
    setLoading(true);

    // Fetch data from an API (replace
URL with your API endpoint)
    const response = await
fetch('https://api.example.com/data');
    const result = await response.json();

    // Update state with fetched data
    setData(result);
    setError(null);
  } catch (error) {
    // Handle errors
    setError('Error fetching data: ' +
error.message);
    setData(null);
  } finally {
    // Set loading to false after data
fetch (whether successful or not)
    setLoading(false);
  }
};

// Call the fetch data function
fetchData();
```

```
},[]); // Empty dependency array means
this effect runs once after the initial
 render

 // Render based on data, loading state,
and errors
  return (
    <div>
      {loading && <p> Loading data... </p>}
      {error && <p style = {{ color: 'red'}}
>{error}</p> }
      {data && (
        <div>
          <h1> Data Fetched Successfully <
h1>
        {/* Render your data here */}
        <pre> {JSON.stringify (data, null,
</pre>
      </div>
    )}
    </div>
  );
}

  export default Data Fetching Example;
```

In this example:

- 'useEffect' is used to initiate the data fetching operation after the component is mounted.
- The 'fetchData' function is an asynchronous function that uses the 'fetch' API to make an HTTP request to a specified URL.
- State variable ('data', 'loading', and 'error') are used to manage the component's state during the data fetching process.
- The component renders different UI based on the current state (loading, error, or successful data fetch).

Remember to replace the API URL with the actual endpoint you want to fetch data from. This example demonstrates a basic pattern for fetching data in a React functional component.

Exercises for useEffect

1. * Fetching Data : *
- Create a component that fetches data from an API using `useEffect` and displays it.

2. * Conditional Effects : *
- Implement a component with `useEffect` that runs different side effects based on certain conditions.

3. * Debouncing Input : *
- Build a search component that uses `useEffect` to debounce user input before making an API call.

4. * Real-time Clock : *
- Create a component that displays a real-time clock and updates every second using `useEffect`.

5. * Form Validation : *
- Implement a form with validation logic using `useEffect` to dynamically check and display error messages.

6. * Infinite Scrolling : *
- Develope a component that Loads more data when the user scrolls to the bottom of the page using 'use-Effect'.

7. * Authentication Status : *
- Build a component that checks the authentication status on mount using 'use Effect' and updates the UI accordingly.

8. * Local Storage Interaction : *
- Create a component that saves and retrieves data from 'Local Storage' using ~~'using~~ 'use Effect'.

9. * Web Socket Integration : *
- Implement a chat component that uses 'useEffect' to established a Web Socket connection and handle incoming messages.

10. * Dynamic Theming : *
- Build a theming component that dynamically changes the application theme using 'useEffect' when a user preference is updated.

React.memo

`React.memo` is a higher-order component in React that memoizes the rendering of a functional component. It's used to optimize performance by preventing unnecessary renders.

Syntax:

```
const MemoizedComponent =
React.memo(MyComponent);
```

Memoization:

1. * Definition: *
   - Memoization is an optimization technique where the results of expensive function calls are cached so that the next time the function is called with the same arguments, the cached result is returned instead of recomputing the result.

2. * Use Cases in React: *
   - Memoization is used in various React scenarious, including optimizing

renders with 'React.memo', caching computed values with 'useMemo' hook, and preventing unnecessary re-renders in functional components.

3. * React.useMemo: *
- The 'useMemo' hook in React is used for memoization with functional components. It memoizes the result of a function and re-runs the function only if the dependencies change.

```
const memoized Value
= React.useMemo (() =>
Compute Expensive Value (a,b), [a,b]);
```

4. * Benefits: *
- Memoization helps improve performance by avoiding rebundant calculations or renders, especially in scenarios where computation are resource-intensive.

5. * Considerations: *
- While memoization is powerful for optimization, it should be used judiciously. Unnecessary memoization can lead to increased memory usage.

Let's consider a scenario where 'React.memo' can be beneficial, and then demonstrate how to fix a potential issue with it.

* Scenario without Memoization: *

Suppose you have a simple component that renders a list of items without using 'React.memo':

```
Const ItemList = ({items}) => {
  console.log ('Rendering ItemList...');

 return (
  <ul>
   {items.map ((item) => (
    <li key = {item.id}>{item.name}</li>
  ))}
  </ul>
 );
};

export default ItemList;
```

In this example, every time the parent

component re-renders, the `ItemList` component will also re-render, even if the `items` prop remains unchanged. This can be inefficient, especially if the list is large or if the rendering logic is computationally expensive.

* Using React.memo to Optimize: *

Now, let's apply `React.memo` to the `ItemList` component to prevent unnecessary re-renders:

```
import React from 'react';

const ItemList = React.memo ({items}
=> {
  console.log ('Rendering ItemList...');

  return (
   <ul>
    {items.map (item) => (
     <li key = {item.id}>{item.name}</li>
    ))}
   </ul>
  );
});
```

```
export default ItemList;
```

By wrapping `ItemList` with `React.memo`, the component will only re-render when the `items` prop changes. The console log statement inside the component will only be executed if there's an actual change in the `items` prop.

This optimization is particularly useful when dealing with large lists or components with complex rendering logic. It helps to avoid unneccessary rendering and can improve the overall performance of your React application.

Use Memo

Suppose you have a component that calculates the sum of an array of numbers. Without memoization, the calculation would occur on every render, even if the array remains the same. `use Memo` can be used to memoize the result of the summation.

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

const NumberList = ({numbers}) => {
  const sum = useMemo (() => {
    console.log ('Calculating sum...');
    return numbers.reduce ((acc, num) =>
acc+ num, 0);
  }, [numbers]);

  return (
    <div>
      <p> Numbers: {numbers.join (',')}</p>
      <p> Sum : {sum}</p>
    </div>
  );
};
```

```
const App = () => {
  const [number Array, set Number Array] =
use State ([1, 2, 3, 4, 5]);

const handle Add Number = () => {
  // Simulate adding a number to the
array
    set number Array ((prev Numbers)
=> [... prev Numbers,
 Math. Floor (Math. random () * 10)]);
};

 return (
  <div>
     <h1> Number List </h1>
      < Number List
numbers = {number Array} />
     < button on click = {handle AddNumber}
  > Add Number </button>
    </div>
 );
};

 export default App;
```

In this example :

- The `NumberList` component calculates the sum of an array of numbers using `useMemo`. The calculation is only performed if the `numbers` array changes.
- The `App` component contains a button to add a random number to the array, simulating a change in the `numbers` prop.

By using `useMemo`, the sum is only recalculated when the `numbers` array changes, preventing unnecessary calculations and rendering when the array remains the same. The console log statement inside the `useMemo` callback will only be executed when the `numbers` prop changes.

use Callback

`use Callback` and `useMemo` are both hooks in React that involve memoization, but they serve slightly different purposes.

`Use Callback`:

1. * Purpose : *
   - `use Callback` is used to memoize functions, especially when those functions are passed as props to child components. It helps in preventing unnecessary re-creation of functions on every render.

2. * Syntax : *

```
Const memoized Callback =
use Callback (() => {
  // Function logic
}, [/* dependencies */]);
```

3. * Dependencies : *
   - The second argument to `useCallback` is an array of dependencies. If

any of these dependencies change, the callback function is re-created; otherwise, it returns the memoized version of the function.

`use Memo`:

1. * Purpose: *
   - `use Memo` is used to memoize the result of a computation. It's often used for optimizing expensive calculations or preventing unnecessary re-renders.

2. * Syntax: *
   const memoizedValue = useMemo (() => computeExpensiveValue (a, b), [a, b]);

3. * Dependencies: *
   - Similar to `useCallback`, `useMemo` takes a second argument, an array of dependencies. If any of these dependencies change, the memoized value is recalculated; otherwise, it returns the cached result.

Using `use Callback` Instead of `use Memo`:

If you want to memoize a function and use it in a child component, you can use `use Callback`:

```
import React, { use State, use Callback }
from 'react';

const Child Component = ({ on Click }) => {
 return (
  <button on Click = {on Click} > Click me! </
 button >
  );
};

const Parent Component = () => {
  const [ count, set Count ] = use State (0);

// Using use Callback to memoize the
on Click handler
  const handle Click = use Callback (() => {
   alert ('Button clicked !');
}, []); // No dependencies, so the
  callback won't change
```

```
return (
  <div>
    <p> Count : { count } </p>
    <Child Component
on Click = { handle Click } />
    <button on Click = {() =>
set Count (count + 1)} > Increment Count </
button >
    </ div >
  );
};

export default Parent Component;
```

In this example :

`handle Click` is memoized using `use Call back`. The `on Click` prop passed to `Child Component` won't change unless `handle Click` is recreated (which only happens if dependencies change).

Remember, while both `use Callback` and `use Memo` provide memoization, they are used

in different contexts. `useCallback` is for memoizing functions, and `useMemo` is for memoizing values or the result of computations. Choose the one that suits your specific use case.

Context API

The Context API in React provides a way to pass data through the component tree without having to pass props manually at every level. It's designed to share state or other information globally in a React application.

1. * Context Object: *
   - The Context API consists of the `create Context` function, which returns a context object. This context object has two components: `Provider` and `consumer`.
   - Example:

```
const My Context =
React.create Context ();
```

2. ** Provider Component: **
   - The `Provider` component is used to wrap the part of the component tree where you want to make the context available.
   - It takes a `value` prop, which is the data you want to share.

- Example :

```
<My Context . Provider
Value = { /* some value */ }>
        {/* Your component tree */}
        </My Context . Provider>
```

3. * Consumer Component : *

- The `Consumer` component is used to access the value from the context within a component.
- It can be used either with the `render props` pattern or the `use Context` hook.
- Example with render props:

```
< My Context . Consumer >
{ value => / * Render Something based
on the context value */ }
< / My Context . Consumer >
```

- Example with `use Context` hook:

```
import {used Context} from 'react';

function My Component (){
const value =
```

```
use Context (My Context);
    // Render something based on the
context value
}
```

4. * Default Values : *
- You can provide a default value to the context when creating it using `create Context`. This default value is used when a component does not find a matching `Provider` in its ancestor tree.
  - Example :

```
const My Context =
React.create Context (default Value);
```

5. * Dynamic Context : *
- The `value` prop of the `Provider` can be dynamic and changed over time. When the value changes, all components that consume that context will re-render.
  - Example :

```
const [value, set Value] =
use State (initial Value);
```

```
return (
  < My Context . Provider value = {value} >
  {/* Your component tree */}
  </My Context . Provider >
);
```

6. * Nested Contexts : *

- You can have multiple contexts in a single application, and they can be nested. Components will access the nearest 'Provider' up the tree.
- Example :

```
< Theme Context . Provider value = {/*
theme value */} >
  < User Context . Provider value={/*user
value */} >
    {/* Your component tree */}
  </ User Context . Provider >
</ Theme Context . Provider >
```

7. * Context API vs. Redux : *

- The Context API is often compared to libraries like Redux for state management. While both can be used for global state, the Context API is simpler and more lightweight, making

it a good choice for smaller applications or when Redux might be overkill.

8. * Use Cases: *
— The Context API is useful for scenarios where prop drilling becomes cumbersome, such as passing down authentication information, themes, or language preferences.

9. * Performance Considerations: *
— Be cautious with the performance implications of using the context API, especially if the value changes frequently, as it can lead to unnecessary re-renders.

10. * Hooks Integration: *
— The Context API integrates well with React Hooks, and you can use the 'UseContext' hook to consume the context value directly in functional components.

The Context API simplifies the management of state that needs to be shared across

components without the need for prop drilling. It is a powerful tool for building more modular and maintainable React applications.

Example of useContex

use Context *

Step 1: Create a Context

```
// User Context.js
import { create Context } from 'react';

const User Context = create Context ();

export default User Context ;
```

Step 2: Create Component A

```
// Component A.js
import React, {use State} from 'react';
import User Context from './User Context';
import Component B from './
Component B';

const Component A = () => {
  const [user, set User] = use State ({ name:
'John', age: 20});
```

```
return (
  <UserContext.Provider value={user}>
    <div>
      <h2> Component A </h2>
      < Component B />
    </ div >
  </UserContext.Provider>
);
};

export default Component A;
```

- We create a state variable `user` in Component A and wrap its content with the `User Context . Provider`. The Value prop of the provider is set to the user state.

Step 3: Create Component B

```
// Component B.js
import React from 'react';
import Component C from './
Component C';

const Component B = () => {
```

```
return (
  <div>
    <h3> Component B </h3>
    <Component C />
  </div>
);
};

export default Component B;
```

- Component B doesn't need to do anything with the context directly. It simply renders Component C.

Step 4: Create Component C

```
// Component C.js
import React from 'react';
import Component D from './
Component D';

Const Component C = () => {
  return (
    <div>
    <h4> Component C</h4>
    <Component D />
```

```
    </div>
  );
};

export default Component C;
```

- Component C doesn't need to do anything with the context directly. It simply renders Component D.

Step 5: Create Component D

```
// Component D.js
import React, {useContext} from 'react';
import UserContext from './UserContext';

const ComponentD = () => {
  const user = useContext(UserContext);

  return (
    <div>
      <h5> Component D </h5>
      <p> User Name: {user.name} </p>
      <p> User Age: {user.age} </p>
    </div>
  );
};
```

```
export default Component D;
```

- Component D uses the `useContext` hook to access the value from the `UserContext`. The `user` variable now holds the user information passed down from Component A.

Explanation:

1. * Component A: *
   - Creates a user state and wrap its content with `UserContext.Provider`, passing the `user` state as the value.

2. * Component B: *
   - Renders Component C.

3. * Component C: *
   - Renders Component D.

4. * Component D: *
   - Uses `useContext(UserContext)` to access the user information passed down from Component A.
   - Displays the user's name and age.

With this set up, the user information flows from Component A to Component D through the context. Components B and C act as intermediaries, and we avoid prop drilling. Each component only consumes the context when needed.

Here are 10 exercises that cover a range of difficulty levels using `useContext` in React:

Beginner Level:

1. * Basic Context Setup: *
— Create a simple context for a theme (light / dark mode). Implement a button that toggles the theme, and use `useContext` to apply the selected theme to components.

2. * User Authentication: *
— Create a context for user authentication. Implement login and log out functionalities, and use `useContext` to unconditionally render component based on the user's authentication status.

3. * Shopping Cart: *
— Build a shopping cart using context. Allow users to add and remove item from the cart. Display the total number of items in the cart in a seperate component using `useContext`.

4. * Language Switcher: *
— Implement a language switcher using context. Allow users to change the application language, and use `use Context` to dynamically update the text in components.

5. * Nested Contexts: *
— Create two context (e.g., User Context and Theme Context). Use them in different components and demonstrate how nested Contexts work.

Intermediate Level:

6. * Dynamic Context Value: *
— Create a Context for a dynamic value (e.g., a timer). Allow components to start, stop, and reset the timer. Use `useContext` to display the timer value in different components.

7. * To do List with Context: *
— Implement a simple to do list with context. Allow users to add, remove, and mark todos as complete. Use `use Context` to share the todo list across components.

8. * Multi-Step Form : *
— Build a multi-step form using context to manage the form state. Allow users to navigate between steps, and use `use Context` to collect and display the form data.

Advanced Level :

9. * Real-Time Data with Context : *
— Simulate real-time data updates (e.g., stock prices) using context. Use a timer to update the data at regular intervals, and use `use Context` to display the real-time data in multiple components.

10. * Authentication with API : *
— Implement user authentication using context and a mock API. Allow users to log in and log out, and use `use Context` to make authenticated API request in different components.

useReducer

`useReducer` is a React Hook that provides an alternative way to manage complex state logic in functional components. It is particularly useful when the state transitions are more complex or depend on the previous state. Here are some key points to understand about `useReducer`:

1. * Purpose : *
— `useReducer` is designed for managing state in scenarios where state logic is more involved, and simple state updates using `useState` become impractical.

2. * Syntax : *
— The basic syntax of `useReducer` is as follows :

```
const [state, dispatch] =
useReducer (reducer, initialState);
```

3. * Arguments : *
— `reducer` : A function that takes the

current state and an action as arguments and returns a new state. It is responsible for defining how the state should change in response to different actions.

— `initial Stale`: The initial state of the component, similar to the initial state in `use State`.

4. * State and Dispatch : *

— `state`: The current state of the component, which is managed by `use Reducer`.

— `dispatch`: A function used to dispatch actions to the reducer, triggering state updates. Actions are objects that describe what kind of state change should occur.

5. * Example : *

```
const initial State = 0;

const reducer = (state, action) => {
  switch (action.type) {
    case 'INCREMENT':
    return state + 1;
```

```
    case 'DECREMENT':
      return state - 1;
    default:
      return state;
  }
};

  const [count, dispatch] =
useReducer(reducer, initialState);
```

— In this example, the state is a number, and the reducer handles actions like 'INCREMENT' and 'DECREMENT' to update the state accordingly.

6. * Complex State Logic: *
— `useReducer` is often preferred when state logic involves more complex transitions and computations. It allows you to encapsulate the logic in the reducer function, making it more moduler and easier to test.

7. * Actions: *
— Actions are plain JavaScript objects with a `type` property that describes the type of action being performed. Additional

properties can be included to provide more information for the reducer to process.
— Example:

```
dispatch ({ type: 'INCREMENT'});
```

8. * Dynamic Actions: *
— Actions can be dynamic, and the payload can be passed for more flexible state updates.
— Example:

```
dispatch ({ type: 'SET_COUNT',
payload: 10 });
```

9. * Benefits: *
— Helps in managing complex state logic in a more organized way.
— Reduces the likelihood of bugs in state transitions by encapsulating the logic within the reducer.

10. * Use Cases: *
— `useReducer` is beneficial when managing state becomes complex, or when state updates depends on the previous state, as in the case of counters, forms, or any dynamic UI elements.

->Example using `useReducer` to manage a simple counter with increment and decrement actions:

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

const initialState = {count:0};

const reducer = (state, action) => {
  switch (action.type) {
    case 'INCREMENT':
      return {count : state.count + 1};
    case 'DECREMENT':
      return {count: state.count - 1};
    default :
      return state;
  }
};

const Counter = () => {

  const [state, dispatch] =
useReducer (reducer, initialState);

  return (
    <div>
      <h2> Counter : {state.count}</h2>
```

```
<button onClick = {() =>
dispatch ({ type: 'INCREMENT'})}
>Increment </button >
<button onClick = {() =>
dispatch ({ type: 'DECREMENT' })}
>Decrement </button >
);
};

export default Counter;
```

→ Now, let's break down each step:

1. **Define the initial state (initialState):**

```
const initialState =
{ count : 0 };
```

— We set the initial state of our counter to an object with a 'count' property initialized to 0.

2. **Define the reducer function (reducer):*

```
const reducer = (state , action) => {
switch (action.type) {
```

```
case 'INCREMENT':
  return { count : state.count + 1 };
case 'DECREMENT':
  return { count : state.count - 1 };
default:
  return state;
}
};
```

— The reducer function takes the current state and an action as arguments. Depending on the action type, it returns a new state. In this case, we handle 'INCREMENT' and 'DECREMENT' actions.

3. * Use useReducer to managed state : *

```
const [state, dispatch] =
useReducer (reducer, initialState);
```

— We use the 'use Reducer' hook, passing in our reducer function and the initial state. It returns the current state ('state') and a 'dispatch' function.

4. * Dispatch actions to update the state : *

```
<button onClick={() => dispatch({type:
'INCREMENT'})}> Increment </button>
<button onClick={() => dispatch({type:
'DECREMENT'})}> Decrement </button>
```

- We provide buttons that, when clicked, dispatch actions of type 'INCREMENT' or 'DECREMENT'. ~~})})~~ This triggers the reducer to update the state accordingly.

With this example, clicking the "Increment" or "Decrement" buttons will update the counter displayed on the UI by dispatching the corresponding actions to the reducer. The 'useReducer' hook manages the state transitions in a clean and predictable manner.

Here are ten exercises with increasing complexity, covering various aspects of using the `useReducer` hook in React:

Beginner Level:

1. * Counter: *
   - Create a simple counter application using `useReducer` to manage state. Include buttons for incrementing and decrementing the counter.

2. * Todo List: *
   - Build a basic todo list application where users can add and remove tasks. Use `useReducer` to manage the state of the todo items.

3. * Toggle Dark Mode: *
   - Implement a feature to toggle between light and dark modes in your application. Use `useReducer` to manage the theme state.

4. * Simple Form: *
   - Create a form with input fields for name and email. Use `useReducer`

to manage the form state and validate the inputs.

5. * User Authentication : *
— Develop a simple user authentication system. Use `useReducer` to manage the authentication state (logged in or logged out).

Intermediate Level :

6. * Shopping Cart : *
— Build a shopping cart application where users can add and remove items. Utilize `useReducer` to manage the state of the shopping cart.

7. * Drag and Drop : *
— Implement a drag-and-drop feature for a list of items. Use `useReducer` to manage the order of the items in the list.

8. * Pagination : *
— Create a paginated list of items. Use `useReducer` to manage the current page and display the corresponding

items.

Advanced Level:

9. * Undo / Redo Functionality: *
— Develop a feature that allows users to undo and redo actions. Use 'useReducer' to manage a history of state changes.

10. * Real-Time Collaboration: *
— Build a collaborative editing feature where multiple users can edit a shared document in real-time. Utilize 'useReducer' to synchronize and manage the document state across users.

Higher Order Components

In Java Script, a higher-order function is a function that takes one or more functions as arguments and/or returns a function as its result. In the context of React, higher-order functions are often used to enhance or modify the behavior of components.

Here's a simple example using a higher-order function in React:

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

//Higher-order function
const withCounter =
(WrappedComponent) => {
 // This component enhances the
functionality of the Wrapped Component
  const EnhancedComponent = () => {
    const [count, setCount] = useState(0);

  const increment = () => {
    setCount(count + 1);
  };

  return (
```

```
    <Wrapped Component
      count = {count}
      increment = {increment}
      // You can pass down other props
here if needed
    />
  );
};

  return EnhancedComponent;
};

// Original component
const Counter = ({count, increment}) =>
{
  return (
    <div>
    <p> Count : {count} </p>
    <button onClick = {increment}
> Increment </button>
    </div>
  );
};

// Enhance the original component using
 the higher-order function
const EnhancedCounter =
```

```
withCounter(Counter);

// Now, you can use EnhancedCounter
just like any other React component
const App = () => {
  return (
    <div>
      <h1> Higher-Order Function in React</
h1>
    <EnhancedCounter />
    </div>
  );
};

export default App;
```

In this example:

1. 'withCounter' is a higher-order function that takes a component ('Wrapped Component') and return a new component ('Enhanced Component') with additional state and functionality

2. 'Counter' is a simple React component that recieves 'count' and 'increment'

as props.

3. 'Enhanced Counter' is the result of applying the higher-order function to the 'Counter' component, which enhances it with the ability to track and increment a count.

4. Finally, 'App' renders the enhanced counter component.

This is a basic example, and higher-order functions in React can be more complex based on the use case. They are commonly used for tasks like state management, code reusability, and component composition.

React Router dom

React Router and Client-Side Routing

Overview :

- Client-Side Routing Definition :
  - In traditional websites, the browser fetches a new document from the server every time you click a link, resulting in a full page reload.
  - React Router introduces "client-side routing", allowing your app to update the URL and render new UI without requesting a new document from the server.

How it Works :

- Traditional Website Flow :
  - Browser requests a document from the server.
  - Downloads and evaluates CSS and Java Script assets.
  - Renders HTML sent from the server.
  - Clicking a link starts the process

anew for a new page.

- Client - Side Routing :
  - React Router manages the URL and UI changes without full page reloads.
  - Clicking a link triggers an immediate update to the URL.
  - Your app renders new UI and makes data requests (e.g., using 'fetch') to update the page dynamically.

Benefits :

- Faster User Experiences :
  - No need to request an entirely new document from the server.
  - Skips the process of re-evaluating CSS and JavaScript assets.
  - Result : quicker page transitions.

- Dynamic User Experiences :
  - Allows for smooth animations and transitions between pages.
  - Enhances interactivity without waiting for server responses.

Implementation with React Router:

- React Router:
  - A Library for handling navigation in React applications.
  - Defines routes that map to different components in your app.

- Example:
  - Clicking a link updates the URL using React Router.
  - The app immediately renders new UI without a full page reload.
  - Data can be fetched to update the page dynamically.

Summary:

- Client-side routing with React Router enables a more fluid and responsive user experience.
- It eliminates the need for the browser to fetch an entirely new document, making page transitions faster and allowing for dynamic UI updates without

server round - trips.
— React Router simplifies the implementation of client - side routing in React applications, making in beginner-friendly for building modern, interactive web pages.

Set up:

just write
npm i react - router - dom
on terminal location of working app

Basic Usage:

1. Importing the Router:
— Wrap your application with `Browser Router` from 'react-router-dom'.

```
import { Browser Router as Router }
from 'react - router - dom';

function App () {
  return (
    <Router>
    {/* Your application components go
here */}
```

```
    <App 2/>
    </Router>
  );
}
```

2. Route Component :

— The `Route` component renders UI based on the URL.

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

function Home () {
  return <h2> Home Page </h2>;
}

function About () {
  return <h2> About Us </h2>;
}

function App 2 () {
  return (
    <Routes>
      <Route path = "/" exact
element = {<Home />} />
      <Route path = "/about"
```

```
element = {< About />} />
  </ Routes >
 );
}
```

3. Link Component :

— The 'Link' component is used for navigation.

```
import {Link} from 'react-router-dom';

function Navigation () {
  return (
   <nav>
   < ul >
   < li >
    < Link to = "/" >Home < / Link >
   </ li >
   < li >
    < Link to = "/about" > About </Link >
   < /Li >
   </ul >
  </nav>
 );
}
```

```
    <App 2/>
    </Router>
  );
}
```

2. Route Component:
— The `Route` component renders UI based on the URL.

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

function Home () {
  return <h2> Home Page </h2>;
}

function About () {
  return <h2> About Us </h2>;
}

function App 2 () {
  return (
    <Routes>
      <Route path = "/" exact
element = {<Home />} />
      <Route path = "/about"
```

```
element = { < About /> } />
  </ Routes >
 );
}
```

3. Link Component :

— The 'Link' component is used for navigation.

```
import {Link} from 'react-router-dom';

function Navigation () {
  return (
   <nav>
   <ul>
   <li>
    <Link to = "/" >Home</ Link >
   </li>
   <li>
    <Link to = "/about"> About </Link>
   </li>
   </ul>
  </nav>
 );
}
```

4. Redirect Component:
- The 'Redirect' component is used for programmatically redirecting users.

```
import { Redirect } from
'react-router-dom';

function
ProtectedPage ({ isAuthenticated }) {
  if (isAuthenticated) {
    return <h2> Welcome to the Protected
Page ! </h2>;
  } else {
    return < Redirect to = "/login" />;
  }
}
```

Advanced Features:

1. Nested Routes:
- Allow for nested components to be rendered based on different parts of the URL.

```
function User () {
  return (
```

```
  <div>
    <h2> User Profile </h2>
    <Route path ="/user/posts"
element = {< User Posts />} />
    <Route path ="/user/settings"
element = {< User Settings />} />
    </ div>
  );
}
```

2. Route Parameters :

— Extract dynamic parts from the URL using route parameters.

```
function User Profile ({ match }) {
const {username} = match.params;
return <h2> User Profile : {username} </
h2>;
}

function App () {
return (
  <Router>
    <Route path ="/user/:username"
element = {< User Profile />} />
  </ Router>
```

```
  );
}
```

3. Route Guards:
- Protect routes based on certain conditions, like user authentication.

```
function PrivateRoute ({ component:
component, isAuthenticated, ...rest }) {
  return (
  <Route
   {... rest }
   render = { (props) =>
   isAuthenticated ? < Component
 {... props } /> : < Redirect to="/login" />
    }
   />
  );
}
```

Explanation of Examples:

- Basic Usage Examples:
- Import `BrowserRouter` to enable routing in your app.

- Use `Route` to define what component to render for specific paths.
- `Link` is used for navigation, providing a way to move between different views.
- `Redirect` helps in directing users to specific pages based on conditions.

- Advanced Features Examples:
  - Nested routes allow for more complex page structures.
  - Route parameters help extract dynamic values from the URL.
  - Route guards (Private Route) protect routes based on certain conditions, providing a way to control access to specific pages.

10 project ideas for React.js that range from beginner to advanced levels:

1. ToDo List App (Beginner):
— Tools Used: State management, useState hook.
— Description: Create a simple ToDo list application with the ability to add, delete, and mark tasks as completed.

2. Weather App (Beginner Intermediate):
— Tools Used: API integration (e.g., Open Weather Map API), useState, useEffect hooks.
— Description: Build a weather application that fetches real-time weather data based on user input.

3. Movie Search App (Intermediate):
— Tools Used: API integration (e.g., The Movie Database API), React Router.
— Description: Develop a movie search application allowing users to explore

and search for information about movies.

4. E-commerce Website (Intermediate):
— Tools Used: React Router, Context API (for state management), styled-components (for styling).
— Description: Create a simple e-commerce platform with product listings, a shopping cart, and checkout functionality.

5. Blog Platform (Intermediate):
— Tools Used: React Router, Redux (for state management), Markdown rendering.
— Description: Build a blog platform where users can view and create blog posts, with support for Markdown formatting.

6. Social Media Dashboard (Intermediate - Advanced):
— Tools Used: React Router, Redux, chart libraries (e.g., Chart.js), Firebase (for authentication and data storage).

— Description : Develop a social media dasboard that displays user analytics, engagement metrics, and charts.

7. Real-Time Chat Application (Advanced):
— Tools Used : React Router, Redux, Firebase (for real-time data synchronization and authentication).
— Description : Create a real-time chat application allowing users to send and recieve messages instantly.

8. Task Management System (Advanced):
— Tools Used : React Router, Redux, Firebase (for authentication and data storage), Drag-and-drop libraries (e.g., react-beautiful-dnd).
— Description : Build a comprehensive task management system with features like drag-and-drop task sorting and collaboration.

9. Portfolio Website (Advanced):
- Tools Used: React Router, Styled-component, Contentful (for content management).
- Description: Develop a professional portfolio website showcasing your projects, skills, and resume.

10. Interactive Data Visualization Dashboard (Advanced):
- Tools Used: React Router, D3.js (or other charting libraries), Redux, Firebase (for data storage).
- Description: Create an interactive dashboard that visualizes complex data sets using charts, graphs, and maps.

Remember to choose a project that aligns with your current skill level, and don't hesitate to challenge yourself to learn new tools and libraries as you progress from beginner to advanced projects.

Date: _____ Page: _____

Project based react js notes

Coming Soon