React js Notes

1.Fundamentals Of React

- ReactJS is a **component-based** JavaScript library used to build dynamic and interactive user interfaces. It simplifies the creation of single-page applications (SPAs) with a focus on performance and maintainability.
- E.g

```
Import React from 'react';
function App(){
    return(
            <div>
                <h1>Hello World</h1>
            </div>
           );
    }
   Export default App;
```

1.1.How does React work?

- React operates by creating an in-memory virtual-DOM rather than directly manipulating the browser's DOM
- It performs necessary manipulations within this virtual representation before applying changes to the actual browser DOM
- **1. Actual DOM and Virtual DOM**
- Initially, there is an Actual DOM(Real DOM) containing a div with two child elements: h1 and h2.
- React maintains a previous Virtual DOM to track the UI state before any updates.
- **2. Detecting Changes**
- When a change occurs (e.g., adding a new h3 element), React generates a New Virtual DOM.
- React compares the previous Virtual DOM with the New Virtual DOM using a process called reconciliation.

- **3. Efficient DOM Update**
- React identifies the differences (in this case, the new h3 element).
- Instead of updating the entire DOM, React updates only the changed part in the New Actual DOM, making the update process more efficient.
- **Key Features of React**
- React is one of the most demanding JavaScript libraries because it is equipped with a ton of features which makes it faster and production-ready. Below are the few features of React.
- **1. Virtual DOM**
- React uses a **Virtual DOM** to optimize UI rendering. Instead of updating the entire real DOM directly, React:
- Creates a lightweight copy of the DOM (Virtual DOM).
- Compares it with the previous version to detect changes (diffing).
- Updates only the changed parts in the actual DOM (reconciliation), improving performance.
- **2. Component-Based Architecture**
- React follows a **component-based approach**, where the UI is broken down into reusable components. These components:
- Can be functional or class-based.
- It allows code reusability, maintainability, and scalability.
- **3. JSX (JavaScript XML)**
- React uses**JSX**, a syntax extension that allows developers to write HTML inside JavaScript. JSX makes the code:
- More readable and expressive.
- Easier to understand and debug.
- **4. One-Way Data Binding**
- React uses one-way data binding, meaning data flows in a single direction from parent components to child

components via props. This provides better control over data and helps maintain predictable behavior.

- **5. State Management**
- React manages component state efficiently using the useState hook (for functional components) or this.state (for class components). State allows dynamic updates without reloading the page.
- **6. React Hooks**
- Hooks allow functional components to use state and lifecycle features without needing class components. Common hooks include:
- **useState:** for managing local state.
- **useEffect:** for handling side effects like API calls.
- **useContext:** for global state management.
- **7. React Router**
- React provides React Router for managing navigation in single-page applications (SPAs). It enables dynamic routing without requiring full-page reloads.
- **ReactJS Lifecycle**
- Every React Component has a lifecycle of its own, the lifecycle of a component can be defined as the series of methods that are invoked in different stages of the component's existence. React automatically calls these methods at different points in a component's life cycle. Understanding these phases helps manage the state, perform side effects, and optimize components effectively.

- ReactJS lifecycle
-
- **1. Initialization**
- This is the stage where the component is constructed with the given Props and default state. This is done in the constructor of a Component Class.
- **2. Mounting Phase**
- **Constructor**: The constructor method initializes the component. It's where you set up the initial state and bind event handlers.
- **render():** This method returns the JSX representation of the component. It's called during initial rendering and subsequent updates.
- **componentDidMount():** After the component is inserted into the DOM, this method is invoked. Use it for side effects like data fetching or setting timers.
- **3. Updating Phase**

- **componentDidUpdate(prevProps, prevState)**: Called after the component updates due to new props or state changes. Handle side effects here.
- **shouldComponentUpdate(nextProps, nextState):** Determines if the component should re-render. Optimize performance by customizing this method.
- **render():** Again, the render() method reflects changes in state or props during updates.
- **4. Unmounting Phase**
- **componentWillUnmount()**: Invoked just before the component is removed from the DOM. Clean up resources (e.g., event listeners, timers).
- *For More detail read the article - React Lifecycle Methods*
- **Applications of React**
- **Web Development:** React is used to build dynamic and responsive web applications, including social media platforms, e-commerce sites, and blogs.
- **Mobile Apps:** React Native allows developers to build mobile apps for iOS and Android using the same codebase.
- **Enterprise Applications**: React is used in building large-scale enterprise applications that require a highly interactive UI.
- **Dashboards and Data Visualizations:** React is great for building real-time dashboards and data visualization tools due to its high performance.
- **New Features Added in React 19**
- **Server-Side Rendering Improvements**: React 19 improves server-side rendering (SSR) performance, allowing web apps to render faster and be more SEO-friendly.
- **React Suspense Advancements**: Suspense has been further enhanced, making it easier to manage asynchronous data loading and enabling better UX for handling component rendering while waiting for data.

- **Concurrent Mode:** Enhancements to Concurrent Mode allow React apps to remain responsive and smooth, especially during complex updates or when handling large amounts of data.
- **Automatic Batching Enhancements**: Automatic Batching improvements provide better performance for asynchronous updates, ensuring that multiple state updates are batched together for improved efficiency.
- Better Integration with Modern Web Standards: React 19 improves integration with the latest web standards like Web Vitals, Intersection Observer, and CSS Grid, enhancing responsiveness and overall performance.
- **New Hooks API:** New hooks were introduced to improve state management and lifecycle control, making functional components even more powerful.

2.Components

In React, components are reusable, independent code blocks (A function or a class) that define the structure and behavior of the UI. They accept inputs (props or properties) and return elements that describe what should appear on the screen.

Key Concepts of React Components:

- Each component handles its own logic and UI rendering.
- Components can be reused throughout the app for consistency.
- Components accept inputs via props and manage dynamic data using state.
- Only the changed component re-renders, not the entire page.

Example:

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

```
// Creating a simple functional component
function Greeting() {
  return (
    <h1>Hello, welcome to React!</h1>
  );
}

export default Greeting;
```

Output:

Hello, welcome to React!

Explanation:

- Greeting is a **React functional component**.
- It returns JSX: <h1>Hello, welcome to React!</h1>.
- ReactDOM.render mounts it to the DOM at an element with id root.

Types of React Components

There are two primary types of React components:

1. Functional Components

Functional components are simpler and preferred for most use cases. They are JavaScript functions that return React elements. With the introduction of React Hooks, functional components can also manage state and lifecycle events.

- **Stateless or Stateful:** Can manage state using React Hooks.
- **Simpler Syntax:** Ideal for small and reusable components.
- **Performance:** Generally faster since they don't require a 'this' keyword.

```
function Greet(props) {
```

```
  return <h1>Hello, {props.name}!</h1>;
}
```

2. Class Components

Class components are ES6 classes that extend React.Component. They include additional features like state management and lifecycle methods.

- **State Management:** State is managed using the this.state property.
- **Lifecycle Methods:** Includes methods like componentDidMount,componentDidUpdate, etc.

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

Props in React Components

Props (short for properties) are read-only inputs passed from a parent component to a child component. They enable dynamic data flow and reusability.

- Props are immutable.
- They enable communication between components.

```
function Greet(props) {
  return <h2>Welcome, {props.username}!</h2>;
}

// Usage
<Greet username="Anil" />;
```

State in React Components

The state is a JavaScript object managed within a component, allowing it to maintain and update its own data over time. Unlike props, state is mutable and controlled entirely by the component.

- State updates trigger re-renders.
- Functional components use the useState hook to manage state.

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


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

Rendering a Component

Rendering a component refers to displaying it on the browser. React components can be rendered using the ReactDOM.render() method or by embedding them inside other components.

- Ensure the component is imported before rendering.
- The ReactDOM.render method is generally used in the root file.

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

Components in Components

In React, you can nest components inside other components to build a modular and hierarchical structure.

- Components can be reused multiple times within the same or different components.
- Props can be passed to nested components for dynamic content.

```
function Header() {
  return <h1>Welcome to My Site</h1>;
}

function Footer() {
  return <p>© 2024 My Company</p>;
}

function App() {
  return (
    <div>
      <Header />
      <p>This is the main content.</p>
      <Footer />
    </div>
  );
}

export default App;
```

Best Practices for React Components

- **Keep Components Small:** Each component should do one thing well.

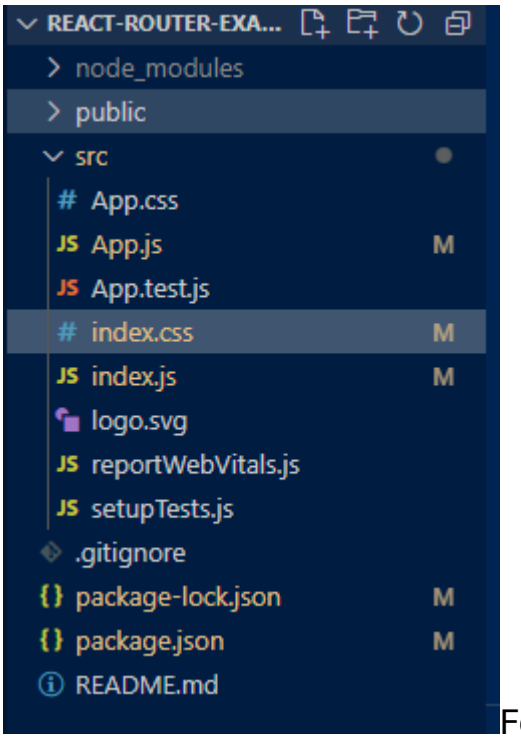
- **Use Functional Components:** Unless lifecycle methods or error boundaries are required.
- **Prop Validation:** Use PropTypes to enforce correct prop types.
- **State Management:** Lift state to the nearest common ancestor when multiple components need access.

3.React Router

- **React Router** is a JavaScript library designed specifically for React to handle client-side routing. It maps specific URL paths to React components, allowing users to navigate between different pages or sections without refreshing the entire page.
- **Types of React Routers**
- There are three types of routers in React:
- **BrowserRouter:** The BrowserRouter is the most commonly used router for modern React applications. It uses the HTML5 History API to manage routing, which allows the URL to be dynamically updated while ensuring the browser's address bar and history are in sync.
- **HashRouter:** The HashRouter is useful when you want to use a URL hash (#) for routing, rather than the HTML5 history API. It doesn't require server configuration and works even if the server doesn't support URL rewriting.
- **MemoryRouter:** The MemoryRouter is used in non-browser environments, such as in React Native or when running tests.
- **Features of React Router**
- **Declarative Routing:** React Router v6 uses the Routes and Route components to define routes declaratively, making the routing configuration simple and easy to read.
- **Nested Routes:** It supports nested routes, allowing for complex and hierarchical routing structures, which helps in organizing the application better.

- **Programmatic Navigation:** The useNavigate hook enables programmatic navigation, allowing developers to navigate between routes based on certain conditions or user actions.
- **Route Parameters:** It provides dynamic routing with route parameters, enabling the creation of routes that can match multiple URL patterns.
- **Improved TypeScript Support:** Enhanced TypeScript support ensures that developers can build type-safe applications, improving development efficiency and reducing errors.
- **Components of React Router**
- Here are the main components used in React Router:
- **1. BrowserRouter and HashRouter**
- **BrowserRouter**: Uses the HTML5 history API to keep your UI in sync with the URL.
- **HashRouter**: Uses the hash portion of the URL (i.e., window.location.hash) to keep your UI in sync with the URL.
- <BrowserRouter>
  (/* Your routes go here */}
  </BrowserRouter>
- **2. Routes and Route**
- **Routes**: A container for all your route definitions.
- **Route**: Defines a single route with a path and the component to render.
- <Routes>
  <Route path="/" element={<Home />} />
  <Route path="/about" element={<About />} />
  </Routes>
- **3. Link and NavLink**
- **Link**: Creates navigational links in your application.
- **NavLink**: Similar to Link but provides additional styling attributes when the link is active.
- <nav>
  <NavLink to="/"
  activeClassName="active">Home</NavLink>

<Link to="/about">About</Link>
</nav>

- **Steps to Create Routes using React Router**
- **Step 1: Initialize React Project**
- Run the following command to create a new React application:
- npx create-react-app react-router-example
  cd react-router-example
- **Step 2: Install React Router**
- Install react-router in your application write the following command in your terminal
- npm install react-router-dom@6
- **Project Structure**
- 

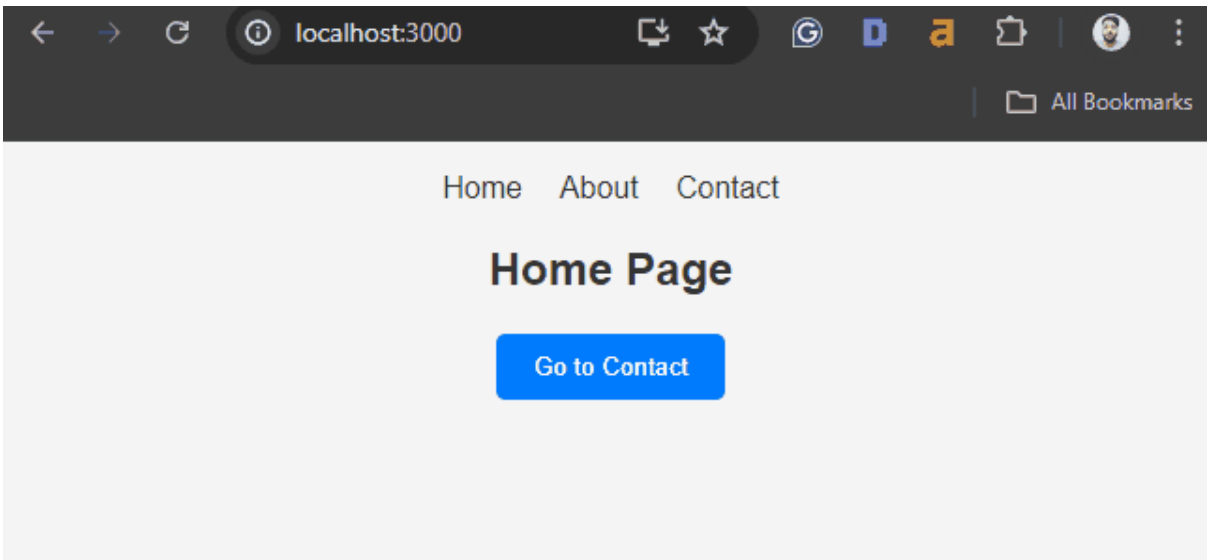
Folder Structure
- **Depenedencies list** after installing **react router**
- "dependencies": {
  "@testing-library/jest-dom": "^5.17.0",
  "@testing-library/react": "^13.4.0",
  "@testing-library/user-event": "^13.5.0",
  "react": "^18.3.1",
  "react-dom": "^18.3.1",
  "react-router-dom": "^6.24.1",

```
    "react-scripts": "5.0.1",
    "web-vitals": "^2.1.4"
  }
```

- **Example:** This example demonstrates implemeting basic routes in a React App.
- *```/* src/index.css */```*
-
- ```body {```
- ``` font-family: Arial, sans-serif;```
- ``` background-color: #f4f4f4;```
- ``` margin: 0;```
- ``` padding: 0;```
- ```}```
-
- ```h2 {```
- ``` text-align: center;```
- ``` color: #333;```
- ```}```
-
- ```nav ul {```
- ``` display: flex;```
- ``` justify-content: center;```
- ``` list-style: none;```
- ``` padding: 0;```
- ```}```
-
- ```nav li {```
- ``` margin: 0 10px;```
- ```}```
-
- ```nav a {```
- ``` text-decoration: none;```
- ``` color: #333;```

- }
-
- **button** {
- **display**: **block**;
- **margin**: 20px **auto**;
- **padding**: 10px 20px;
- **background-color**: #007BFF;
- **color**: **white**;
- **border**: **none**;
- **border-radius**: 5px;
- **cursor**: **pointer**;
- }
-
- **button**:hover {
- **background-color**: #0056b3;
- }
- **Step 4:** Run the application using the following command.
- npm start

Output:

- 

  React Router
- **Uses of React Router**
- **Navigation and Routing:** React Router provides a declarative way to navigate between different views or pages in a React

application. It allows users to switch between views without refreshing the entire page.

- **Dynamic Routing:** React Router supports dynamic routing, which means routes can change based on the application's state or data, making it possible to handle complex navigation scenarios.
- **URL Management:** React Router helps manage the URLs in your application, allowing for deep linking, bookmarkable URLs, and maintaining the browser's history stack.
- **Component-Based Approach:** Routing is handled through components, making it easy to compose routes and navigation in a modular and reusable way.
- **Handling Nested Routes**: React Router allows the creation of nested routes, which enables a more organized and structured approach to rendering content. This is particularly useful for larger applications with a complex structure.

4.React Lifecycle:

In React, the lifecycle refers to the various stages a component goes through. These stages allow developers to run specific code at key moments, such as when the component is created, updated, or removed. By understanding the React lifecycle, you can better manage resources, side effects, and performance. These phases allow you to run specific code at key moments in a component's life, such as when it’s created, updated, or removed from the screen.

Here are the three main phases of the React component lifecycle

React Lifecycle Method

- **Mounting:** Initializes, renders, and mounts the component (componentDidMount()).
- **Updating:** Handles state/prop changes, re-renders, and updates (componentDidUpdate()).

- **Unmounting:** Cleans up before removal (componentWillUnmount()).

Phases of Lifecycle in React Components

1. Mounting

Mounting refers to the process of creating and inserting a component into the DOM for the first time in a React application. During mounting, React initializes the component, sets up its internal state (if any), and inserts it into the DOM.

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

constructor()

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

```
constructor(props) {
  super(props); // Always call super(props) before using this.props
  this.state = {
    count: 0, // Initial state
  };
  console.log("Constructor called");
}
```

getDerivedStateFromProps(props, state)

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

```
static getDerivedStateFromProps(props, state) {
  if (props.value !== state.value) {
    return { value: props.value }; // Update state based on new props
```

```
  }
  return null; // No changes to state
}
```

render() method

Responsible for rendering JSX and updating the DOM.

```
render() {
  return (
    <div>
      <h1>Hello, React Lifecycle!</h1>
    </div>
  );
}
```

componentDidMount() Function

This function is invoked right after the component is mounted on the DOM, i.e. this function gets invoked once after the render() function is executed for the first time.

```
componentDidMount() {
  console.log("Component has been mounted");

  // Example: Fetch data from an API
  fetch("https://api.example.com/data")
    .then(response => response.json())
    .then(data => this.setState({ data }));
}
```

2. Updation

Updating refers to the process of a component being re-rendered due to changes in its state or props. This phase occurs whenever a component's internal state is modified or its parent component passes new props. When an update happens, React re-renders the component to reflect the changes and ensures that the DOM is updated accordingly.

- getDerivedStateFromProps
- setState() Function
- shouldComponentUpdate()
- getSnapshotBeforeUpdate() Method
- componentDidUpdate()

getDerivedStateFromProps

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

```
static getDerivedStateFromProps(props, state) {
  if (props.name !== state.name) {
    return { name: props.name }; // Update state with new props
  }
  return null; // No state change
}
```

setState()

This is not particularly a Lifecycle function and can be invoked explicitly at any instant. This function is used to update the state of a component. You may refer to this article for detailed information.

```
this.setState((prevState, props) => ({
    counter: prevState.count + props.diff
}));
```

shouldComponentUpdate()

shouldComponentUpdate() Is a lifecycle method in React class components that determines whether a component should re-render. It compares the current and next props/states and returns true if the component should update or false if it should not.

shouldComponentUpdate(nextProps, nextState)

It returns true or false, if false, then
render(), componentWillUpdate(), and componentDidUpdate() method does not get invoked.

getSnapshotBeforeUpdate() Method

The getSnapshotBeforeUpdate() method is invoked just before the DOM is being rendered. It is used to store the previous values of the state after the DOM is updated.

getSnapshotBeforeUpdate(prevProps, prevState)

componentDidUpdate()

Similarly, this function is invoked after the component is rendered, i.e., this function gets invoked once after the render() function is executed after the updation of State or Props.

componentDidUpdate(prevProps, prevState, snapshot)

3. Unmounting

This is the final phase of the lifecycle of the component, which is the phase of unmounting the component from the DOM. The following function is the sole member of this phase.

componentWillUnmount()

This function is invoked before the component is finally unmounted from the DOM, i.e., this function gets invoked once before the component is removed from the page, and this denotes the end of the lifecycle.

Implementing the Component Lifecycle methods

Let us now see one final example to finish the article while revising what's discussed above.

First, create a react app and edit your **index.js** file from the src folder.

// Filename - src/index.js:

```
import React from "react";
import ReactDOM from 'react-dom';
class Test extends React.Component {
  constructor(props) {
    super(props);
    this.state = { hello: "World!" };
  }
  componentDidMount() {
    console.log("componentDidMount()");
  }
  changeState() {
    this.setState({ hello: "Geek!" });
  }
  render() {
    return (
      <div>
        <h1>
          GeeksForGeeks.org, Hello
          {this.state.hello}
        </h1>
        <h2>
```

```
            <a
              onClick={this.changeState.bind(
                this
              )}
            >
              Press Here!
            </a>
          </h2>
        </div>
      );
  }
  shouldComponentUpdate(nextProps, nextState) {
    console.log("shouldComponentUpdate()");
    return true;
  }
  componentDidUpdate() {
    console.log("componentDidUpdate()");
  }
}
const root = ReactDOM.createRoot(
  document.getElementById("root")
);
root.render(<Test />);
```

Output

In this example

- The Test class is a React component with a state property hello initially set to “World!”.
- The componentDidMount() the method runs after the component is added to the DOM, logging “componentDidMount().”
- The changeState() method updates the state to change hello to “Geek!” when called.
- In the render() method, the component displays the hello state inside an <h1> tag and includes a link to trigger the changeState() function.
- The shouldComponentUpdate() method logs “shouldComponentUpdate()” and allows the component to re-render while componentDidUpdate() logs “componentDidUpdate()” after the update.

React Lifecycle Methods Table

Lifecycle Method	Phase	Purpose
constructor()	Mounting	Initialize state, bind methods
getDerivedStateFromProps()	Mounting & Updating	Sync state with props
render()	Mounting & Updating	Returns JSX to be displayed
componentDidMount()	Mounting	Run side effects (API calls)

Lifecycle Method	Phase	Purpose
shouldComponentUpdate()	Updating	Control re-rendering
getSnapshotBeforeUpdate()	Updating	Capture values before update
componentDidUpdate()	Updating	Run side effects after the update
componentWillUnmount()	Unmounting	Cleanup before unmounting

Significance of React Component Lifecycle

It provides a structured way to handle specific tasks at various points in a component's life, such as when it is created, updated, or destroyed.

- **Data Fetching:** Lifecycle methods like **componentDidMount()** and **componentDidUpdate()** allow you to fetch data, subscribe to services, or update external resources at the appropriate time.
- **Performance Optimization:** By controlling when certain actions occur (e.g., using shouldComponentUpdate() to prevent unnecessary re-renders), you can optimize performance and ensure your app runs efficiently.
- **Resource Management**: The lifecycle helps with cleanup tasks, such as removing event listeners, canceling network requests, or clearing timers (using methods like componentWillUnmount()) to prevent memory leaks.
- **Handling State and Props Changes**: The lifecycle provides hooks to update and respond to state or props changes in a controlled way, allowing for dynamic behavior and interaction in your components.

- **Handling Component Removal**: When a component is no longer needed, the lifecycle helps you clean up resources and ensure that nothing unnecessary continues to run.

React lifecycle method in class component vs functional component

- In **class components**, we need different methods for handling state and side effects, which can be more complex.
- In **functional components**, hooks like useState() and useEffect() make it easier to manage state and side effects, making the code shorter and simpler. Functional components are often preferred because they're cleaner and easier to understand.

Features	Class Components	Functional Components
State Initialization	constructor()	useState()
Lifecycle Methods	componentDidMount(), shouldComponentUpdate(), componentDidUpdate(), componentWillUnmount()	useEffect() handles mounting, updating, and unmounting
Handling Updates	shouldComponentUpdate(), componentDidUpdate()	useEffect() with dependency array
Cleanup	componentWillUnmount()	Return cleanup function in useEffect()

Features	Class Components	Functional Components
Functionality	Component methods tied to lifecycle phases	Hooks like useState, useEffect, useCallback, useMemo

5.React Hooks

ReactJS Hooks, introduced in React 16.8, are among the most impactful updates to the library, with over 80% of modern React **projects** adopting them for state and lifecycle management. They let developers use state, side effects, and other React features without writing class components. Hooks streamline code, improve readability, and promote a functional programming style, making functional components as powerful—if not more—than their class-based counterparts.

Types of React Hooks

React offers various hooks to handle state, side effects, and other functionalities in functional components. Below are some of the most commonly used types of React hooks:

1. State Hooks

State hooks, specifically useState and useReducer, allow functional components to manage state in a more efficient and modular way. They provide an easier and cleaner approach to managing component-level states in comparison to class components.

useState: The useState hook is used to declare state variables in functional components. It allows us to read and update the state within the component.

Syntax

const [state, setState] = useState(initialState);

- **state:** The current value of the state.
- **setState:** A function used to update the state.
- **initialState:** The initial value of the state, which can be a primitive type or an object/array

useReducer: The useReducer hook is a more advanced state management hook used for handling more complex state logic, often involving multiple sub-values or more intricate state transitions.

Syntax

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

- **state:** The current state value.
- **dispatch:** A function used to dispatch actions that will update the state.
- **reducer:** A function that defines how the state should change based on the dispatched action.
- **initialState:** The initial state value.

Now let's understand how state hook works using this example

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


function App() {
  const [count, setCount] = useState(0);
  const increment = () => setCount(count + 1);
  const decrement = () => setCount(count - 1);
```

```
  return (
    <div>
      <h1>Count: {count}</h1> {/* Display the current count */}
      <button onClick={increment}>Increment</button> {/* Increment the count */}
      <button onClick={decrement}>Decrement</button> {/* Decrement the count */}
    </div>
  );
}

export default App;
```

In this code

- useState is used to declare state variables in functional components.
- The state variable (count) and the updater function (setCount) allow you to read and update the state.

2. Context Hooks

The useContext hook in React is a powerful and convenient way to consume values from the React Context API in functional components. It allows functional components to access context values directly, without the need to manually pass props down through the component tree
const contextValue = useContext(MyContext);

const contextValue = useContext(MyContext);

- The useContext hook takes a context object (MyContext) as an argument and returns the current value of that context.

- The contextValue will hold the value provided by the nearest <MyContext.Provider> in the component tree.

Now let's understand how context hook works using this example

```
import React, { createContext, useContext, useState } from "react";


const ThemeContext = createContext();


function App() {
  const [theme, setTheme] = useState("light");


  const toggleTheme = () => {
    setTheme((prevTheme) => (prevTheme === "light" ? "dark" : "light"));
  };


  return (
    <ThemeContext.Provider value={theme}>
      <div>
        <h1>Current Theme: {theme}</h1>
        <button onClick={toggleTheme}>Toggle Theme</button>
        <ThemeDisplay />
      </div>
    </ThemeContext.Provider>
  );
}
```

```
function ThemeDisplay() {
  const theme = useContext(ThemeContext);

  return <h2>Theme from Context: {theme}</h2>;
}

export default App;
```

In this code:

- useContext allows you to consume context values, making it easier to share data across components without prop drilling.
- The Provider makes the context value accessible to all components below it in the component tree.

3. Effect Hooks

Effect hooks, specifically useEffect,useLayoutEffect, and useInsertionEffect, enable functional components to handle side effects in a more efficient and modular way.

useEffect: The useEffect hook in React is used to handle side effects in functional components. It allows you to perform actions such as data fetching, DOM manipulation, and setting up subscriptions, which are typically handled in lifecycle methods like componentDidMount or componentDidUpdate in class components.

Syntax

```
useEffect(() => {
  // Side effect logic here
}, [dependencies]);
```

- useEffect(() => { ... }, [dependencies]); runs side effects after rendering.

- The effect runs based on changes in the specified dependencies.

useLayoutEffect: The useLayoutEffect is used when we need to measure or manipulate the lawet before the browser paints, ensuring smooth transitions and no flickering.

Syntax

```
useLayoutEffect(() => {
  // Logic to manipulate layout or measure DOM elements
}, [dependencies]);
```

useInsertionEffect: The useInsertionEffect is designed for injecting styles early, especially useful for server-side rendering (SSR) or styling libraries, ensuring styles are in place before the component is rendered visually.

Syntax

```
useInsertionEffect(() => {
   // Logic to inject styles or manipulate stylesheets
}, [dependencies]);
```

Now let's understand how effect hook works using this example:

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


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


   useEffect(() => {
      document.title = `Count: ${count}`;
      console.log(`Effect ran. Count is: ${count}`);


      return () => {
```

```
      console.log("Cleanup for previous effect");
      document.title = "React App";
    };
  }, [count]);

  return (
    <div>
      <h1>Count: {count}</h1>
      <button onClick={() => setCount(count + 1)}>Increment Count</button>
    </div>
  );
}

export default App;
```

In this code

- useEffect is great for handling side effects like data fetching, subscriptions, or manually manipulating the DOM (like changing the document title).
- The cleanup function is useful to clean up resources (like timers or event listeners) when the component is unmounted or before the effect runs again.

4. Performance Hook

Performance Hooks in React, like useMemo and useCallback, are used to optimize performance by avoiding unnecessary re-renders or recalculations.

useMemo: useMemo is a React hook that memoizes the result of an expensive calculation, preventing it from being recalculated on every render unless its dependencies change. This is particularly useful when we have a computation that is expensive in terms of performance, and we want to avoid recalculating it on every render cycle.

Syntax

const memoizedValue = useMemo(() => computeExpensiveValue(a, b), [a, b]);

useCallback: The useCallback is a React hook that helps to memoize functions, ensuring that a function is not redefined on every render unless its dependencies change. This is particularly useful when passing functions as props to child components, as it prevents unnecessary re-renders of those child components.

Syntax

const memoizedCallback = useCallback(() => { doSomething(a, b); }, [a, b]);

- useMemo caches the computed value (num * 2), recalculating it only when num changes.
- This prevents unnecessary calculations on every render.

Now let's understand how performance hook works using this example

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


function App() {
  const [count, setCount] = useState(0);
  const [text, setText] = useState("");

  const expensiveCalculation = useMemo(() => {
    console.log("Expensive calculation...");
    return count * 2;
```

```
  }, [count]);

  return (
    <div>
      <h1>Count: {count}</h1>
      <h2>Expensive Calculation: {expensiveCalculation}</h2>
      <button onClick={() => setCount(count + 1)}>Increment
Count</button>

      <input
        type="text"
        value={text}
        onChange={(e) => setText(e.target.value)}
        placeholder="Type something"
      />
    </div>
  );
}

export default App;
```

In this code

- useMemo memoizes the result of expensiveCalculation.
- It only recomputes when count changes.
- When text changes, the calculation is not re-run, optimizing performance.

- console.log appears only when count changes, showing memoization works.

5. Resource Hooks(useFetch)

The useFetch is typically a custom hook used for fetching data from an API. It is implemented with useEffect to fetch data when the component mounts or when dependencies change.

Synatx

const { data, loading, error } = useFetch(url);

- useFetch is a custom hook for fetching data from a given URL.
- It uses useEffect to fetch data when the URL changes and updates the data state.

Now let's understand how resource hook works using this example

```
import React, { useState, useRef } from "react";


function App() {
  const countRef = useRef(0);
  const [forceRender, setForceRender] = useState(false);


  const increment = () => {
    countRef.current += 1;
    setForceRender(!forceRender);
  };


  return (
    <div>
      <h1>Count: {countRef.current}</h1> {/* Display count value */}
```

```
        <button onClick={increment}>Increment</button>
      </div>
    );
}

export default App;
```

In this code

- countRef holds the mutable count value.
- useState (forceRender) triggers re-renders to reflect changes in the UI.
- When the Increment button is clicked, countRef is updated.
- setForceRender forces a re-render to update the UI.
- The updated count is displayed in an <h1> tag, not in a prompt.

6. Other Hooks

React offers additional hooks for specific use cases

- **useReducer:** For complex state management.
- **useImperativeHandle:** Customizes the instance value exposed by useRef.
- **useLayoutEffect:** Like useEffect but fires synchronously after DOM updates.

7. Custom Hooks

Custom Hooks are user-defined functions that encapsulate reusable logic. They enhance code reusability and readability by sharing behavior between components.

//useWidth.js

```
import { useState, useEffect } from "react";

function useWidth() {
  const [width, setWidth] = useState(window.innerWidth);

  useEffect(() => {
    const handleResize = () => setWidth(window.innerWidth);
    window.addEventListener("resize", handleResize);
    return () => window.removeEventListener("resize", handleResize);
  }, []);

  return width;
}

export default useWidth;
```

Using a Custom Hook

```
import React from "react";
import useWidth from "./useWidth";

function App() {
  const width = useWidth();
  return <h1>Window Width: {width}px</h1>;
}
```

export default App;

- The custom hook useWidth encapsulates the logic for tracking the window's width.
- It reduces redundancy by reusing the logic across components.

*we can see a Complete list of React Hooks in **ReactJS Hooks Complete Reference**.*

Difference Between Hooks and Class Components

Feature	Class Components	React Hooks
State Management	this.state and lifecycle methods	useState and useEffect
Code Structure	Spread across methods, can be complex	Smaller, focused functions
Reusability	Difficult to reuse logic	Easy to create and reuse custom hooks
Learning Curve	Familiar to OOP developers	Requires different mindset than classes
Error Boundaries	Supported	Not currently supported
Third-party Libraries	Some libraries rely on them	May not all be compatible yet

6.Advanced React Concepts

6.1 Lazy loading in React:

Lazy Loading in React is used to initially load and render limited data on the webpage. It helps to optimize the performance of React applications. The data is only rendered when visited or scrolled it can be images, scripts, etc. Lazy loading helps to load the web page quickly and presents the limited content to the user that is needed for the interaction lazy loading can be more helpful in applications that have high-resolution images or data that alters the loading time of the application.

Table of Content

Lazy Loading in React

In React, Lazy loading is a technique that allows you to load components, modules, or assets asynchronously, improving the loading time of your application. It can be achieved by using the built-in **React.lazy()** method and **Suspense component**.

Syntax:

```
// Implement Lazy Loding with React.Lazy method
const MyComponent = React.lazy(() => import('./MyComponent'));
```

Approach

To implement the lazyloading in react follow the steps given below:

- Firstly, Recognize the component you want to Lazy Load. These are mostly Large or complex which is not necessary for all the users when the page loads.
- Import the **lazy()** and Suspense components from the React package

- Use the **lazy()** function to dynamically import the component you want to lazy load:
  Note that the argument to the **lazy()** function should be a function that returns the result of the import() function.
- Wrap the lazy-loaded component in a **Suspense** component, which will display a fallback UI while the component is being loaded:

```
<Suspense fallback={<div>Loading...</div>}>
  <MyComponent />
</Suspense>
```

Example: This example uses the Suspense component and lazy method to implement the lazyloading for specific components.

```
import React from "react";

import { Suspense, lazy } from "react";

const Component1 = lazy(() =>

    import("../src/LazyContent/myComponent1"));

const Component2 = lazy(() =>

    import("../src/LazyContent/myComponent2"));

function App() {

    return (

        <>

            <h1> Lazy Load</h1>

            <Suspense

                fallback={<div>Component1 are loading please wait...</div>}

            >

                <Component1 />

            </Suspense>
```

```
                    <Suspense
                        fallback={<div>Component2 are loading please wait...</div>}
                    >
                        <Component2 />
                    </Suspense>
                </>
        );
}

export default App;
```

Explanation

Lazy Loading in React can be implemented with the help of the built-in function **React. lazy().** This is also known as **code splitting**, In which **React.lazy** along with webpack bundler divides the code into separate chunks, when the component is requested the chunk is loaded on demand. The use of React Suspense is to define fallback content to be displayed during asynchronous components or data loading, as shown above.

React Suspense provides better feedback to the user and improves the user experience as a user is not facing any blank screen or space while the content is being loaded. React Suspense is designed to handle the loading of the components that make asynchronous API requests. React Suspense can be used by wrapping the **<Suspense>** component and specifying the fallback content displayed while the component or data is loading.

Code-splitting: It is an effective technique for optimizing the performance and efficiency of web applications, especially those with large code bases. By reducing the amount of code that needs to be loaded when a page first

loads, code splitting can improve the user experience and make your application more responsive and fast.

Advantages

- Lazy loading allows you to use server resources more efficiently by loading only the resources you need. This is very important for high-traffic applications or when server resources are tight.
- A quicker initial load time can be achieved by using lazy loading, which minimizes the amount of code that must be downloaded and parsed when the page first loads. This can speed up your application's first load time
  greatly.

Conclusion

React Lazy Loading is a powerful technique that significantly improves the performance of web applications built with React. One of the key benefits of lazy loading is that it can help improve the Time to Interactive (TTI) metric, which is the time it takes for a page to become interactive and responsive. By delaying the loading of non-critical components until the page has finished loading, lazy loading reduces TTI and provides a more engaging user experience.

6.2 Higher order component:

Higher-order components (HOC) are an advanced technique in React that is used for reusing component logic. It is the function that takes the original component and returns the new enhanced component.

- It doesn’t modify the input component directly. Instead, they return a new component with enhanced behavior.
- They allow you to reuse component logic across multiple components without duplicating it.
- They are pure functions that accept a component and return a new component.

Syntax:

```
const EnhancedComponent =
higherOrderComponent(OriginalComponent);
```

In this syntax:

- higherOrderComponent is a function that takes an existing component (OriginalComponent) as an argument.
- It returns a new component (EnhancedComponent) with additional functionality or behavior.
- The EnhancedComponent behaves like the original component but with enhanced features provided by the HOC.

Implementation of the Higher-Order Components

Step 1: Create a React application

Create a React application by using the following command.

```
npm create vite@latest foldername
```

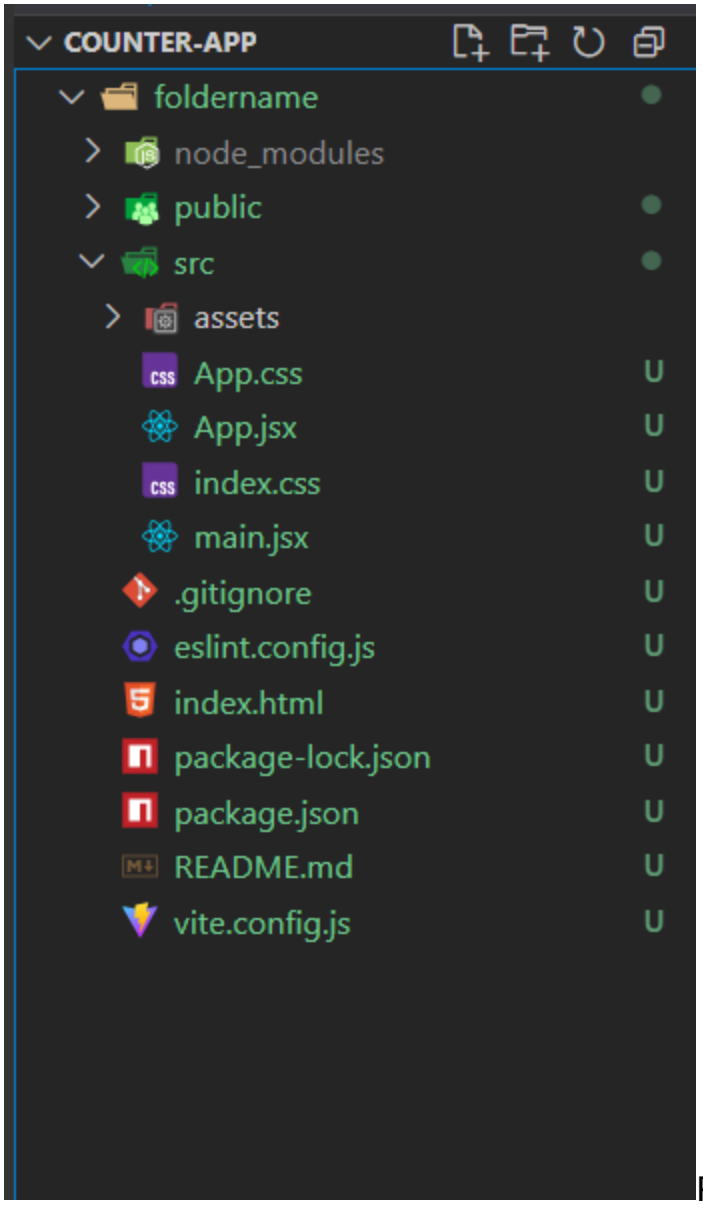
- where foldername is the name of your project. You can change it to any name you prefer.
- **npm create vite@latest foldername:** This command initializes a new Vite project with a React template. Replace foldername with your desired project name.

Step 2: Move in the Folder

After creating your project folder i.e. foldername, move to it using the following command.

```
cd foldername
```

Project Structure:



Project Structure

Example 1: Let say, we need to reuse the same logic, like passing on the name to every component.

```
import React from 'react';

// Higher-Order Component (HOC) as a functional component

const withName = (OriginalComponent) => {

  const NewComponent = (props) => {

    return <OriginalComponent {...props} name="GeeksforGeeks" />;

  };

  return NewComponent;
```

```
};

export default withName;
```

In this example:

- **HOC Definition**: withName is a Higher-Order Component (HOC) that adds a name prop with the value "GeeksforGeeks" to any component passed into it.
- **Original Component:** The App component simply renders the name prop inside an <h1> element.
- **Applying the HOC:** In App.js, the App component is passed to the withName HOC, creating a new component, EnhancedComponent.
- **Enhanced Component:** The EnhancedComponent now has the name prop and will display "GeeksforGeeks" when rendered.
- **Export:** The EnhancedComponent is exported and used to display the final output in the browser.

Example 2: In this example let's implement some logic. Let's make a counter app. In HighOrder.js, we pass the **handleclick** and **show** props for calling the functionality of the component.

```
body {
  margin: 0;
  font-family: sans-serif;
  background: #f0f4f8;
  display: flex;
  justify-content: center;
  align-items: center;
  height: 100vh;
}
```

```
.container {
  background: #ffffff;
  padding: 40px;
  border-radius: 16px;
  box-shadow: 0 8px 16px rgba(0, 0, 0, 0.1);
  text-align: center;
}

.title {
  margin-bottom: 20px;
  font-size: 1.8rem;
  color: #333;
}

.count {
  font-size: 4rem;
  color: #007bff;
  margin-bottom: 20px;
}

.buttons {
  display: flex;
  gap: 12px;
  justify-content: center;
```

```
}

.btn {
  font-size: 1.5rem;
  padding: 10px 16px;
  border: none;
  border-radius: 8px;
  background-color: #007bff;
  color: white;
  cursor: pointer;
  transition: 0.3s;
}

.btn:hover {
  background-color: #0056b3;
}

.reset {
  background-color: #ff4d4f;
}

.reset:hover {
  background-color: #d9363e;
}
```

In this example:

- **App.jsx**: The App component imports and uses the EnhancedCounter, which is the Counter component wrapped by the withCounter HOC to add counter logic.
- **index.js**: It renders the App component to the root DOM element using ReactDOM.createRoot.
- **withCounter.jsx**: The withCounter HOC takes a component (WrappedComponent) and adds state logic (increment, decrement, and reset) for counting.
- **Counter.jsx**: The Counter component displays the current count and provides buttons to increment, decrement, or reset the counter value.
- **State Management**: The withCounter HOC uses useState to manage the count and passes the count and its control functions as props to the Counter component.

Reason to Use Higher-Order Components

- **Code Reusability**: HOCs allow you to reuse logic across multiple components without repeating the same code in each one.
- **Separation of Concerns**: They help separate the logic and UI, making components easier to manage and maintain.
- **Enhances Readability**: By abstracting shared logic into HOCs, your components remain clean and focused solely on rendering UI.
- **Easy to Maintain**: Centralizing shared behavior in a HOC reduces code duplication, making it easier to fix bugs and add new features.

Best Practices for Using (HOC)

- **Don't Overuse HOCs:** Use HOCs only when necessary. Too many HOCs can make your code complex and harder to manage.
- **Use for Reusable Logic:** HOCs are good for adding common features (like authentication or loading states) across multiple components.

- **Pass All Props**: Make sure the HOC passes all the props from the original component to the new one, unless you specifically want to modify or add something.
- **Name Components Clearly:** Always give meaningful names to wrapped components, which helps in debugging and readability.

Conclusion

Higher-Order Components (HOCs) are a powerful tool in React for reusing component logic and enhancing components without changing their original behavior. By wrapping a component with an HOC, you can add extra functionality such as authentication, data fetching, or logging.

6.3 Code Splitting:

Code-Splitting is a feature supported by bundlers like Webpack, Rollup, and Browserify which can create multiple bundles that can be dynamically loaded at runtime.

As websites grow larger and go deeper into components, it becomes heavier. This is especially the case when libraries from third parties are included. Code Splitting is a method that helps to generate bundles that are able to run dynamically. It also helps to make the code efficient because the bundle contains all required imports and files.

Bundling and its efficiency: Bundling is the method of combining imported files with a single file. It is done with the help of **Webpack, Rollup, and Browserify** as they can create many bundles that can be loaded dynamically at runtime.
With the help of code splitting, 'lazy load' can be implemented, which means just using the code which is currently needed.

- **The default way of importing is as follows:**

```
import { add } from './math';

console.log(add(x, y));  // Here x, y are two numbers
```

- **Using code-splitting this can be done as follows:**

```
import("./math").then(math => {
  console.log(math.add(x, y));
});
// Here x, y are two numbers
```

As soon as Webpack gets this type of syntax, code-splitting is started automatically. When using the Create React App, it is already set up and can be used immediately.
The Webpack guide on code splitting should be followed if using Webpack. The instructions can be found here.
When Babel is being used, it has to be made sure that Babel is not transforming the import syntax, but can parse it dynamically. This can be done using babel-plugin-syntax-dynamic-import.

React.lazy and Suspense

As both React.lazy and Suspense are not available for rendering on the server yet now, it is recommended to use loadable-components for code-splitting in a server-rendered app. React.lazy is helpful for rendering dynamic import as a regular component.
Before:

```
import Component from './Component';
```

After:

```
const Component = React.lazy(() => import('./Component'));
```

The Bundle will be loaded on its own which contains the Component when this component is rendered first.
The Lazy component should then be rendered inside Suspense Component which helps to reflect some fallback content meanwhile the lazy component loads.

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

const Component = React.lazy(() => import('./Component'));

function MyComponent() {
```

```
  return (
    <div>
      <Suspense fallback={<div>Loading...</div>}>
    </div>);
}
```

The fallback prop can accept any element of React which will be rendered while waiting for the loading of the Component. The Suspense Component can be placed anywhere above the lazy component. Moreover, multiple lazy components can be wrapped with a single Suspense Component.

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


const ComponentOne =
    React.lazy(() => import('./ComponentOne'));
const ComponentTwo =
    React.lazy(() => import('./ComponentTwo'));
function MyComponent() {
     return (
       <div>
          <Suspense fallback={<div>Loading...</div>}>
       </div>);
}
```

Error Boundaries

Error Boundaries are React components that help when some modules fail to load due to any issue, an error will be triggered. These errors can be handled properly and provide a good experience to the user by the use of a suitable error page.

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

```
import ErrorBoundary from './ErrorBoundary';


const ComponentOne = React.lazy(() =>
    import('./ComponentOne'));
const ComponentTwo = React.lazy(() =>
    import('./ComponentTwo'));
const MyComponent = () => (
  <div>
   <Suspense fallback={<div>Loading...</div>}>
  </div>
);
```

Route-Based Code Splitting

It can be difficult to implement code-splitting in code, the bundles can be split evenly, which will improve the experience for the user.
Here you can see the example code for this.

```
import React,{Suspense, lazy} from 'react';
import {Route, Routes, BrowserRouter } from 'react-router-dom';


const HomePage = lazy(() =>
   import('./routes/HomePage'));
const AboutUs = lazy(() =>
    import('./routes/AboutUs'));
const App = () =>
   (<Suspense fallback={<div>Loading...</div>}>);
```

Named Exports

React.lazy currently supports only default exports. An intermediate module that re-exports as default has to be created if one wants to import a module that uses named exports. This ensures the working of tree shaking and prevents the pulling in of unused components.

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


// Components.js

export const Component = /* ... */;

export const MyUnusedComponent = /* ... */;


// Component.js

export { Component as default } from "./Components.js";


// MyApp.js

const Component = lazy(() => import("./Component.js"));
```