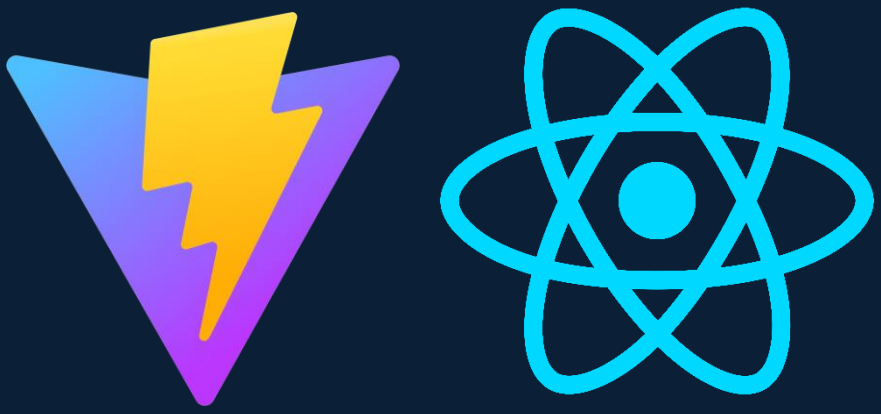
React JS

with

Vite



INTRODUCTION

React

React is a popular JavaScript library for building user interfaces, particularly single-page applications (SPAs).

Vite

Vite is a modern build tool that provides a super-fast development environment

Why use Vite with React?

- Fast development with instant updates.
- Simple setup with zero configuration.
- Better performance with optimized builds.
- Support for modern JavaScript features.

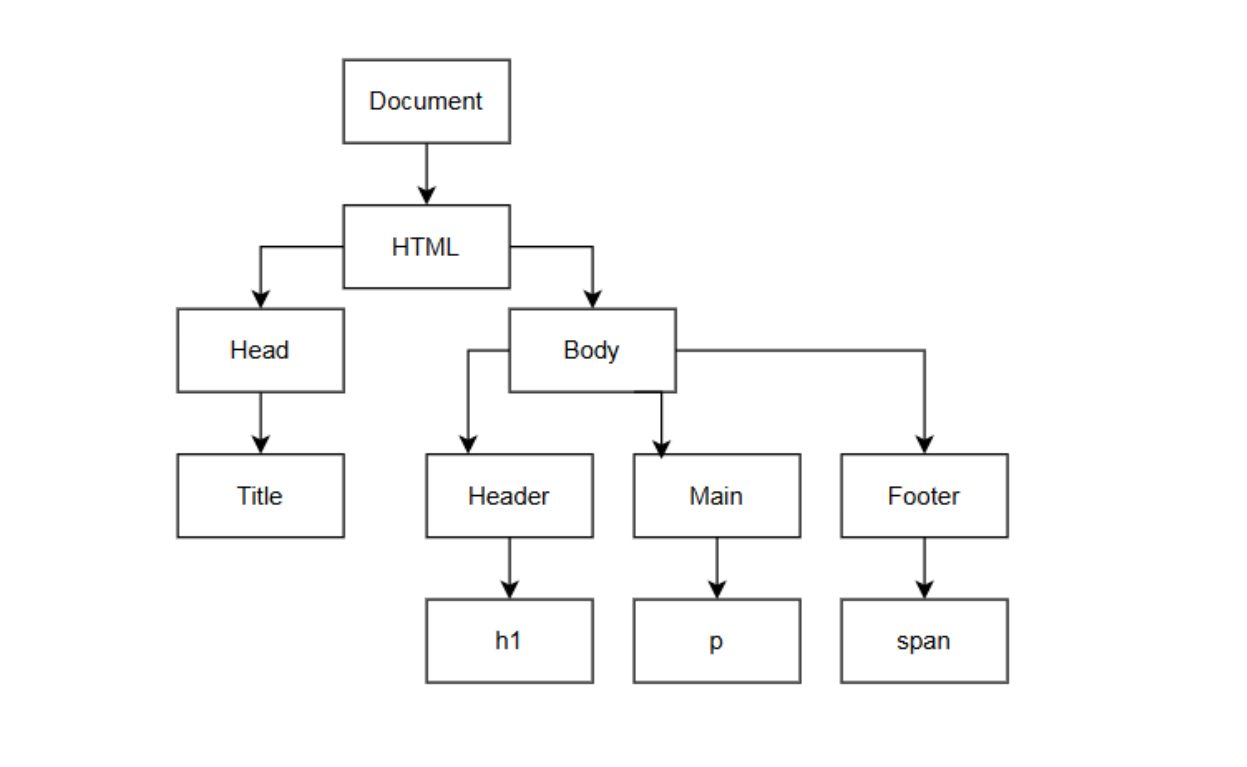
DOM VS. VIRTUAL DOM

What is DOM?

The Document Object Model (DOM) is a programming interface for web documents. It represents the structure of an HTML or XML document as a tree of objects, where each element (such as `<div>`, `<p>`, `<h1>`, etc.) is a node in the tree.

```html
<!DOCTYPE html>
<html>
<head>
    <title>My Page</title>
</head>
<body>
    <header>
        <h1>Welcome</h1>
    </header>
    <main>
        <p>Hello World!</p>
    </main>
    <footer>
        <span>Copyright 2025</span>
    </footer>
</body>
</html>
```

The DOM Tree

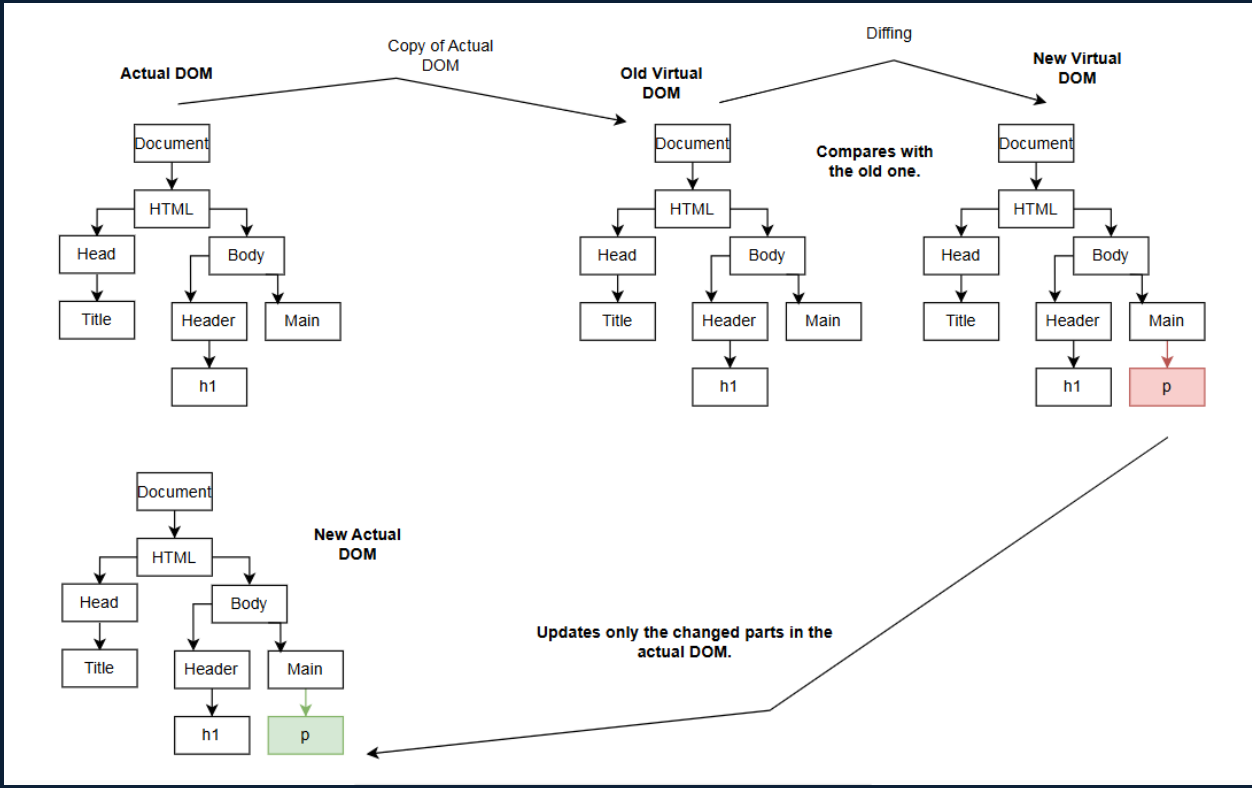


- The document is the root of everything.
- The HTML element contains two main branches: head and body.
- Each element contains its children (like title inside head).
- Text content appears as leaf nodes at the end of branches.

- The browser uses the DOM to render a web page and update it dynamically.
- Each HTML element becomes a DOM node.
- JavaScript can manipulate the DOM using methods like document.getElementById() or document.querySelector().

What is Virtual DOM?

The Virtual DOM (VDOM) is a lightweight copy of the actual DOM, used to improve performance in JavaScript frameworks like React.



- React creates a Virtual DOM tree based on the initial state.
- When data changes, React creates a new Virtual DOM.
- React compares the new Virtual DOM with the previous one.
- React updates only the changed elements in the actual DOM.

DOM vs. Virtual DOM

Feature	DOM	Virtual DOM
Performance	Slower when frequently updated	Faster due to efficient diffing
Updates	Updates the whole tree	Updates only changed elements
Re-rendering	Directly modifies the DOM	Creates a copy and updates the DOM efficiently

⚛ SETTING UP REACT WITH VITE

1. Installing Node.js & npm

Make sure you have Node.js and npm installed. Check versions using Terminal :

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

If not installed, download from Node.js Official Site

2. Create a Project

Open the terminal and do the following.

```
npm create vite@latest my-react-app --template react
```

```
Select a framework:
○ Vanilla
○ Vue
● React
○ Preact
○ Lit
○ Svelte
○ Solid
○ Qwik
○ Angular
○ Others
```

```
Select a variant:
○ TypeScript
○ TypeScript + SWC
● JavaScript
○ JavaScript + SWC
○ React Router v7 ↗
```

3. Navigate to the Project Directory

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

4. Install Dependencies

```
npm install
```

5. Start the Development Server

```
npm run dev
```

FOLDER STRUCTURE

```
my-react-app/
├── node_modules/          # Contains all installed npm packages and dependencies.
├── public/                # Stores static assets like images and the main HTML file.
├── src/                   # Main directory for your React application code.
|   ├── assets/            # Holds static files like images, fonts, or styles.
|   ├── App.css            # CSS styles for the main App component.
|   ├── App.jsx            # The main React component where your UI is defined.
|   ├── index.css          # Global CSS styles for the entire application.
|   └── main.jsx           # Entry point for the React app, renders the App component.
├── .gitignore             # Specifies files and folders to ignore in Git.
├── eslint.config.js       # Configuration file for ESLint (code linting tool).
├── index.html             # The main HTML file that serves as the entry point.
├── package-lock.json      # Lock file for dependency versions.
├── package.json           # Contains project metadata, scripts, and dependencies.
├── README.md              # Documentation file for your project.
└── vite.config.js         # Configuration file for Vite (build tool).
```

node_modules: This directory contains all the npm packages and dependencies required for your project.

public: This folder typically contains static assets like images, fonts, and the index.html file. These assets are served directly to the client.
src: This is the main directory where your React application code resides.

assets: This subfolder is used for storing static assets like images, stylesheets, or fonts that are used within your React components.

App.css: This file contains the CSS styles for your main App component.

App.jsx: This is the main React component file where your application's UI is defined.

index.css: This file contains global CSS styles that apply to your entire application.

main.jsx: This is the entry point for your React application. It renders the App component into the DOM.

.gitignore: This file specifies which files and directories should be ignored by Git, such as node_modules and build artifacts.

eslint.config.js: This configuration file is used for setting up ESLint, a tool for identifying and fixing problems in your JavaScript code.

index.html: The main HTML file that serves as the entry point for your application. It includes a root div where your React app is mounted.

package-lock.json: This file is automatically generated by npm, it locks the versions of your project's dependencies.

package.json: This file contains metadata about your project, including dependencies, scripts, and other configurations.

README.md: This markdown file typically contains documentation about your project, such as how to set it up and run it.

vite.config.js: This is the configuration file for Vite, where you can customize the build and development server settings.

⚛ REACT BASICS

React Components

In React, components are the building blocks of a user interface. They are reusable, self-contained pieces of code that represent a part of the UI. Components can be thought of as custom HTML elements that encapsulate logic, structure, and styling.

A component is a JavaScript function or class that returns a piece of UI (usually written in JSX).

❖ How to Create a Component File

When working with React and Vite, creating and organizing components is straightforward. Here's a step-by-step guide to creating a component file, including the file extension, directory structure, and best practices.

1. File Extension

In a Vite + React.js project, React components typically use the .jsx file extension

2. Steps to Create a Component File

    1. Create the components Directory
    - If it doesn’t already exist, create a components directory inside the src folder.
    2. Create a Component File
    - Inside the components directory, create a new file with a .jsx extension. For example, let's create a Greeting.jsx file.

3. Write the Component Code

```jsx
// src/components/Greeting.jsx
function Greeting() {
  return <h1>Hello, World!</h1>;
}

export default Greeting;
```

4. Use the Component in Another File

- Now, you can import and use the Greeting component in another file, such as App.jsx.

```jsx
// src/App.jsx
import Greeting from './components/Greeting';

function App() {
  return (
    <div>

      <Greeting />

    </div>
  );
}

export default App;
```

5. Run the Application

```
npm run dev
```

- Visit http://localhost:5173 (or the URL provided by Vite) to see the output.

❖ Types of Components

1. Functional Components

- A functional component is a JavaScript function that returns JSX (JavaScript XML) to describe the UI.
- It is simpler and more concise compared to class components.

Example:

```
function Greeting() {
    return (
        <>
         <h1>Hello, World!</h1>;
        </>
    )
}
export default Greeting;
```

2. Class Components (Older Style)

- A class component is a JavaScript class that extends React.Component and must include a render() method that returns JSX.
- Class components have been the traditional way to create components in React, especially before the introduction of hooks.

Example:

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

class ClassC extends Component {
  render() {
    return (
      <>
        <h1>This Is The Class Component</h1>
      </>
    );
  }
}

export default ClassC;
```

JSX

- JavaScript XML (JSX) is a syntax extension that allows you to write HTML-like code inside JavaScript.
- JSX makes React code easier to read and write.

React Fragments (`<React.Fragment>` or `<>...</>`)

React Fragments are a way to group multiple elements without adding an extra DOM node. This is useful when you need to return multiple elements from a component without wrapping them inside a `<div>` or another container.

In React, when returning multiple elements from a component, you must wrap them inside a single parent element because JSX requires a single root node. This can be done using a `<div>`, `<React.Fragment>`, or the shorthand `<>...</>`.

Using a `<div>` creates an extra element in the DOM, which can sometimes affect styling and layout, especially when dealing with CSS frameworks like Flexbox or Grid. On the other hand, React Fragments (`<React.Fragment>` or `<>...</>`) allow you to group elements without adding unnecessary DOM nodes, keeping your markup clean and efficient. If you try to return multiple sibling elements without wrapping them, React will throw a syntax error, as JSX does not allow multiple root elements.

Example 01: Without a Wrapper (Causes Error)

If we try to return multiple elements without a wrapper, React will throw an error.

```
function App() {
    return (

        <h1>Hello</h1>
        <p>World</p>

    );
  }
export default App;
```

Example 02: Using a `<div>` (Works, but Adds Extra DOM Node)

Wrapping elements inside a `<div>` fixes the issue, but adds an extra `<div>` to the DOM.

```
function App() {
    return (
      <div>
        <h1>Hello</h1>
        <p>World</p>
      </div>
    );
  }
export default App;
```

Example 03: Using <React.Fragment> (No Extra DOM Nodes)

Using <>...</> (Shorter Fragment Syntax)

A React Fragment groups elements without adding an extra wrapper.

<React.Fragment>

```
import React from "react";
function App() {
    return (
      <React.Fragment>
        <h1>Hello</h1>
        <p>World</p>
      </React.Fragment>
    );
  }
export default App;
```

<>

```
function App() {
    return (
      <>
        <h1>Hello</h1>
        <p>World</p>
      </>
    );
  }
export default App;
```

Props (Properties)

In React, props (short for "properties") are used to pass data from a parent component to a child component. Props allow components to be reusable and dynamic by enabling the parent to send different values each time the component is used. They are read-only, meaning a child component cannot modify the props it receives. Instead, changes must happen in the parent component, which then passes updated values as props. Props can be any data type, including strings, numbers, arrays, objects, or even functions. To use props, a component receives them as an argument and accesses them using props.propName.

Additionally, props.children lets you pass content between a component's opening and closing tags, useful for layouts like modals or cards.

Example 01:

- Props are passed to a component as attributes in JSX and accessed inside the component using props.

Parent Component

```
import Greeting from './Components/Greeting';

function App() {
    return (
     <>
       <Greeting name="John" age={25}  />
     </>
    )
}
export default App;
```

- Here, pass the name and age props to the Greeting component.
- The Greeting component receives name and age as props and displays them

Child Component

```
function Greeting(props) {
  return (
    <>
      <h1>Hello, {props.name}!</h1>
      <p>Age: {props.age}</p>
    </>
  );
}

export default Greeting;
```

Or (Destructuring Props for Cleaner Code).

- Instead of props.name and props.age, we can use destructuring for cleaner code.

```
function Greeting({name, age}) {
    return (
      <>
        <h1>Hello, {name}!</h1>
        <p>Age: {age}</p>
      </>
    );
  }

  export default Greeting;
```

Output:

Example 02: props.children

- props.children is a special prop that allows you to pass content between the opening and closing tags of a component. It is commonly used for creating reusable wrapper components like modals, cards, or layout containers.
- You can pass multiple elements, or text as children.

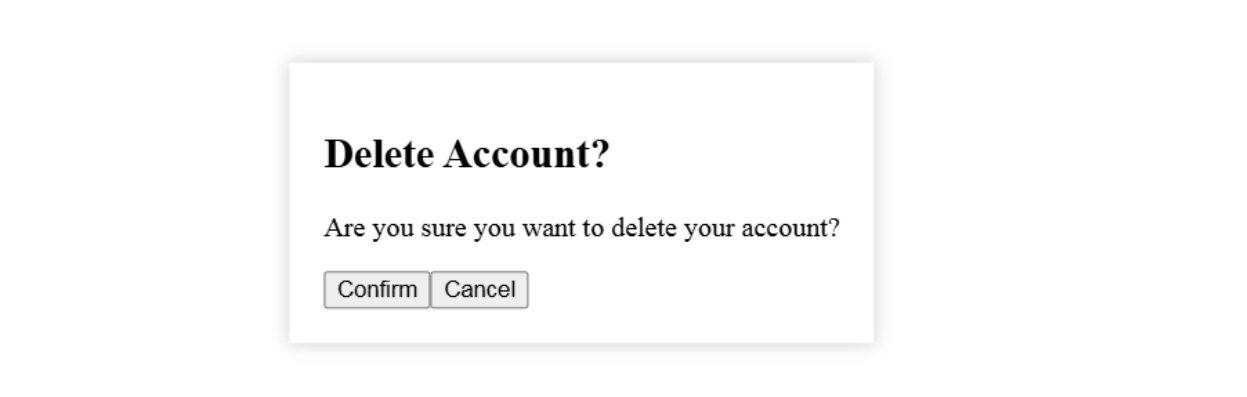
Parent Component

```
import Modal from "./components/Modal";

function App() {
  return (
    <Modal>
      <h2>Delete Account?</h2>
      <p>Are you sure you want to delete your account?</p>
      <button>Confirm</button>
      <button>Cancel</button>
    </Modal>
  );
}

export default App;
```

Child Component

```
function Modal(props) {
    return (
      <div style={{
        position: "fixed",
        top: "50%",
        left: "50%",
        transform: "translate(-50%, -50%)",
        background: "white",
        padding: "20px",
        boxShadow: "0 0 10px rgba(0,0,0,0.2)",
        zIndex: 1000,
      }}>
        {props.children}
      </div>
    );
  }

  export default Modal;
```

Output:



Event Handling

In React, event handling is similar to regular JavaScript but follows JSX syntax, where events like onClick, onChange, and onSubmit are written in camelCase instead of lowercase (onclick → onClick).

Event handlers are written inside curly braces {}, unlike regular HTML where they are written as strings. For example, in React, you write onClick={showAlert} instead of onclick="showAlert()".

Example:

```
function App() {

    function showAlert() {
        alert("Button Clicked");
    }

  return (
    <>
       <button onClick={showAlert}>Click</button>
    </>
  )
}

export default App;
```

React Hooks

React Hooks are special functions that allow developers to use state and lifecycle features inside functional components.Introduced in React 16.8, hooks make React code more concise, readable, and reusable.

Key hooks like useState allow for direct state management within functional components, while useEffect handles side effects such as data fetching or DOM manipulation. Hooks like useContext simplify the process of accessing context values, and useRef provides a way to interact with DOM elements or persist values across renders without triggering re-renders.

We must import them from the 'react' package.

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

Example: useState Hook

```
import { useState } from 'react';
function App() {
    const [number, setNumber] = useState(0);

    function increment() {
        setNumber(number + 1);
    }

  return (
    <>
      <button onClick={increment}>+</button>
      <p>{number}</p>
    </>
  )
}

export default App;
```

- In this example, number holds the state value, and setNumber updates it when the button is clicked.

useState is allows functional components to manage state. It returns an array with two values: the current state and a function to update the state. The initial state is passed as an argument to useState. When the state is updated using the provided function, React re-renders the component with the new state value.

Conditional Rendering

Conditional rendering in React allows developers to dynamically display or hide components and elements based on specific conditions, enabling more interactive and responsive user interfaces.

Instead of rendering the same elements every time, React can render specific components, text, or elements based on state, props, or user interactions. There are several ways to implement conditional rendering in React.

❖ **If-else Statement**
Best for complex conditions inside functions.

Message.jsx (Component)

```
function Message({ isLoggedIn }) {
  if (isLoggedIn) {
    return <h1>Welcome back!</h1>;
  } else {
    return <h1>Please log in.</h1>;
  }
}
export default Message;
```

Usage: App.jsx

```jsx
import Message from "./components/Message";

function App() {
  return (
    <div>
      <Message isLoggedIn={true} />
    </div>
  );
}

export default App;
```

If isLoggedIn is true, it displays: Welcome back!
If isLoggedIn is false, it displays: Please log in.

Output

❖ Ternary Operator (condition ? trueCase : falseCase)
It's often used when the logic is simple and there are only two options to render.

Message.jsx (Component)

```jsx
function Message({ isLoggedIn }) {
  return <h1>{isLoggedIn ? "Welcome back!" : "Please log in."}</h1>;
}

export default Message;
```

Usage: App.jsx

```jsx
import Message from "./components/Message";

function App() {
  return (
    <div>
      <Message isLoggedIn={false} />
    </div>
  );
}

export default App;
```

If isLoggedIn is true, it displays: Welcome back!
If isLoggedIn is false, it displays: Please log in.

Output

❖ **Logical && (AND) Operator (Short-circuit Evaluation)**
This can be useful when you only want to render something when a condition is true.

Notification.jsx (Component)

```jsx
function Notification({ hasNewMessages}) {
  return (
    <div>
      <h1>Dashboard</h1>
      {hasNewMessages && <p>You have new messages!</p>}
    </div>
  );
};

export default Notification;
```

Usage: App.jsx

```jsx
import Notification from "./components/Notification";

function App() {
  return (
    <div>
      <Notification hasNewMessages={true} />
    </div>
  );
}

export default App;
```

If hasNewMessages is true it displays: You have new notifications!

If hasNewMessages is false: Nothing is rendered.

Output

❖ **switch Statement (Multiple Cases)**

Best for multiple case handling

StatusMessage.jsx (Component)

```jsx
function StatusMessage({ status }) {
  switch (status) {
    case "success":
      return <h1>Operation successful!</h1>;
    case "error":
      return <h1>Something went wrong!</h1>;
    case "loading":
      return <h1>Loading...</h1>;
    default:
      return <h1>Unknown status</h1>;
  }
};
export default StatusMessage;
```

Usage: App.jsx

```jsx
import StatusMessage from "./components/StatusMessage";

function App() {
  return (
    <div>
      <StatusMessage status="success" />
    </div>
  );
}
export default App;
```

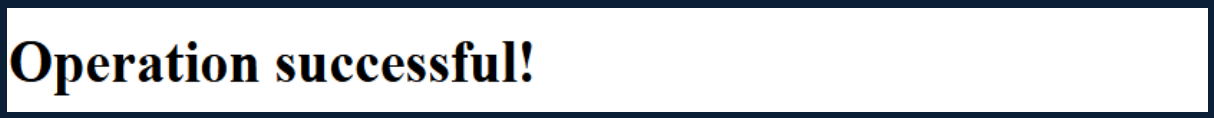
If status is "success", it displays:Operation successful!

If status is "error", it displays:Something went wrong!

If status is "loading", it displays:Loading...

If status is any other value, it displays:Unknown status.

Output



Styling React Components

React offers multiple ways to style components.

❖ **Inline Styles**

Inline styles in React are written as JavaScript objects. There are several ways to apply them efficiently

- ➢ **Directly Inside style Attribute**
  - The style attribute takes an object with CSS properties written in camelCase.

```
function App() {
  return <h1 style={{ color: "blue", fontSize: "24px" }}>Hello,
React!</h1>;
}

export default App;
```

- **Using a JavaScript Object**
  - Best for reusable styles inside components.
  - Define the styles in an object and use it in the style attribute.

```
function App() {
  const headingStyle = {
    color: "blue",
    fontSize: "24px",
    backgroundColor: "lightgray",

    paragraph: {
      fontSize: "20px",
      color: "#901000",
    },
  };
  return (
    <>
      <h1 style={headingStyle}>Hello, React!</h1>
      <p style={headingStyle.paragraph}>This is a simple React
application.</p>
    </>
  );
}
export default App;
```

❖ CSS Stylesheets (.css files)

The most common way of styling React apps.

Styles are written in external .css files and imported into your components.

style.css

```css
.container {
    color:aqua;
    background-color: black;
    padding: 10px;
}
```

App.jsx

```jsx
import "./style.css";

function App() {

  return (
    <div>
      <p className="container"> Hello World </p>
    </div>
  );
}

export default App;
```

For large scale and fast development in React, using CSS frameworks like Tailwind CSS, Bootstrap can significantly speed up the styling process.

How to Use Images in React JS

React allows you to include images in several ways, depending on where the image is stored and how you want to optimize it.

❖ **Using Images from the public Folder**

- Place images inside the public/ folder and reference them with a relative URL.
- The path starts with / (root of public folder).

- Best for large assets like logos or background images.

Example:

```
function App() {

  return (
    <div>
      <img src="/my-image.jpg" alt="My Image" />
    </div>
  );
}

export default App;
```

❖ **Using Local Images (from the src folder)**

- Store images in the src folder and import them directly.

Example:

```
import logo from './assets/images/logo.png'; // Path
relative to the component

function App() {
  return (
    <div>
      <img src={logo} alt="Company Logo"/>
    </div>
  );
}

export default App;
```

- The word logo is the variable name you assign to the image file you imported. It is an arbitrary identifier that you choose yourself when importing the file. You can use any variable name when importing an image in React.

❖ **Using External Image URLs**

- Directly use an external URL in the src attribute.

Example:

```
function App() {
  return (
    <div>
      <img src="https://example.com/my-image.jpg"
alt="External Image" />
    </div>
  );
}

export default App;
```

React Router

React Router enables client-side navigation in single-page applications (SPAs) by dynamically rendering components based on the URL.

➢ **First, install the package**

```
npm install react-router-dom
```

➢ **Basic Routing**

Wrap your app with `<BrowserRouter>` and define routes using `<Routes>` and `<Route>`.

- `<BrowserRouter>` → Manages URL changes.
- `<Routes>` → Container for all `<Route>` definitions.
- `<Route>` → Maps a path to a React component (element).

Example:

```
import Navbar from "./Components/Navbar";
import { BrowserRouter, Routes, Route } from "react-
router-dom";
import About from "./Components/About";
import Home from "./Components/Home";
import Contact from "./Components/Contact";

function App() {
  return (
    <>
      <BrowserRouter>
        <Navbar />
        <Routes>
          <Route path="/" element={<Home />} />
          <Route path="/about" element={<About />} />
          <Route path="/contact" element={<Contact />}
/>
        </Routes>
      </BrowserRouter>
    </>
  );
}
```

- **Navigating Between Pages**

Use the `<Link>` instead of `<a>` for navigation without reloading

You need to import `<Link>` from react-router-dom

Example:

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

function Navbar() {
  return (
    <nav>
      <Link to="/">Home</Link>

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

      <Link to="/contact">Contact</Link>
    </nav>
  );
}

export default Navbar;
```

➢ **Nested Routes**

Nested routing means having routes inside other routes, allowing hierarchical navigation structures. It helps organize complex applications by grouping related components together.

In React Router, nested routes let child components be rendered inside a parent component while maintaining separate URLs.

To properly render these nested components, React Router provides the `<Outlet />` component. `<Outlet />` acts as a placeholder inside the parent component where the child route's content will be dynamically displayed.

Example:

```
<>
  <BrowserRouter>
   <Navbar />
   <Routes>
      <Route path="/" element={<Home />} />
      <Route path="/about" element={<About />} >
        <Route path="dashboard" element={<Dashnoard />} />
      </Route>
      <Route path="/contact" element={<Contact />} />

   </Routes>
  </BrowserRouter>
</>
```

```
<Route path="/about" element={<About />} >
  <Route path="dashboard" element={<Dashnoard />} />
</Route>
```

Render nested routes with <Outlet /> in the parent

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

function About() {
  return (
    <div>
      <h1>About Page</h1>

      <Link to="dashboard">Dashboard Page</Link>
      <Outlet />
    </div>
  )
}

export default About
```

Resources

- https://react.dev/learn
- https://legacy.reactjs.org/
- https://www.w3schools.com/react/default.asp
- https://www.geeksforgeeks.org/react/
- https://v3.vitejs.dev/guide/
- https://reactrouter.com/home
- https://www.npmjs.com/package/react-router-dom

Thanks for reading!

I hope this guide helped you understand React with Vite.

@thusitha-sampath-2000ts