React JS

NOVEMBER 4

FSD -Central Training Team

A. What is React JS

React.js is a popular open-source JavaScript library for building user interfaces, primarily for single-page applications (SPA). It allows developers to build web applications that can update and render efficiently in response to data changes. React is maintained by Facebook and a community of developers.

Benefits of Using React.js

- **Declarative UI**: React allows you to build user interfaces by describing what the UI should look like based on the application's state.
- **Reusable Components**: Components are independent, reusable pieces of code that make development easier and more efficient.
- **Virtual DOM**: React's virtual DOM ensures that changes are efficiently applied to the real DOM, improving performance.
- **Rich Ecosystem**: React has a large ecosystem of libraries and tools to extend its capabilities, from routing (React Router) to state management (Redux, MobX).
- **Community and Support**: React is maintained by Facebook and has a large, active developer community.

B. Rendering Approach

Traditional Website:

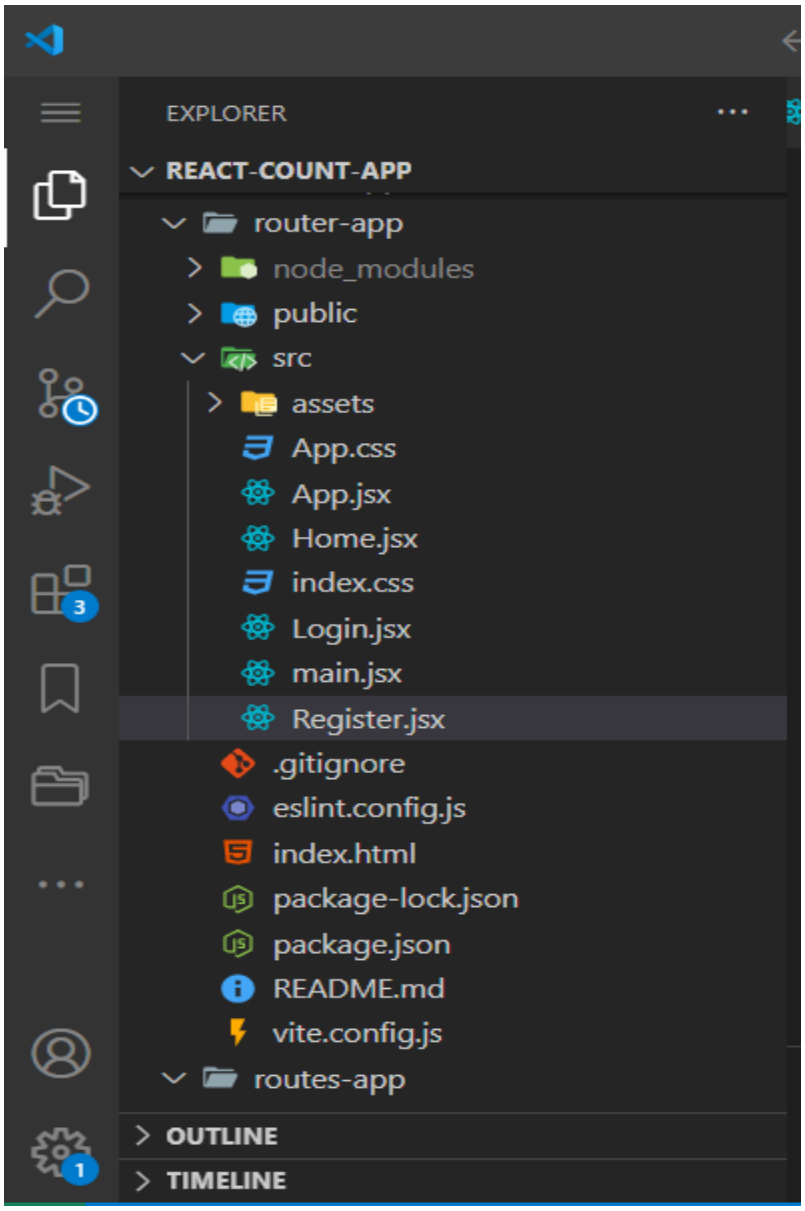
- Server-Side Rendering (SSR): Each time a user navigates to a new page or performs an action, the browser sends a request to the server.
- Full-Page Reloads: The server responds by rendering a full HTML page and sending it back, which the browser displays. This process can be slow as each request triggers a complete page reload.
- HTML-Driven UI: Pages are static or dynamically generated on the server and sent to the client.

React.js (SPA):

- Client-Side Rendering (CSR): The initial load fetches a minimal HTML page and a JavaScript bundle. After this, React takes over rendering on the client side.
- Single-Page Application (SPA): React manipulates the DOM dynamically, updating only the parts of the page that need changing, without reloading the entire page. This creates a smoother, faster user experience.
- Component-Based UI: The UI is broken down into reusable components, allowing for efficient updates and maintenance.

C. Basic Folder Structure

When you create a new React project with Vite, the default structure looks something like this:



Key Folders and Files

node_modules

This folder contains all the dependencies and packages installed via npm. It's managed automatically by npm, so you generally don't need to interact with it directly.

public

The public folder holds static assets that are accessible directly in the application, such as images or icons. Vite will serve files from this folder as-is, meaning they won't be processed or bundled.

src

The src folder is where the main code for your application lives. Here's a breakdown of the core files and subfolders:

assets: This folder is typically used for storing images, fonts, and other static assets specific to your application. Vite uses a more efficient way to handle assets, allowing you to import them directly in your JavaScript/JSX files.

components: This folder holds reusable React components that you'll use throughout your application. For example, you might have a Header.jsx, Footer.jsx, or Button.jsx component here.

App.jsx: This is the main component that serves as the root of your application. You can think of it as the container for your entire app, where you structure routes or render top-level components.

App.css: A CSS file used for styling the App.jsx component or for global styles. You can organize additional CSS files here or in individual component folders as needed.

index.css: This is the global CSS file for base styles or reset CSS.

main.jsx: This is the entry point of the application. It's responsible for rendering the App component into the DOM using ReactDOM.createRoot(). Here, you'll also set up the context providers or routing libraries if needed.

.gitignore

This file specifies which files and folders should be ignored by Git. Common entries include node_modules, environment variable files, and build outputs.

index.html

Vite provides a direct entry point for the application using this index.html file. Unlike typical React setups (like create-react-app), Vite allows for a more flexible index.html, which is used to inject the root of the app directly.

package.json

This file contains metadata about your project, including its name, version, dependencies, and scripts. Important scripts include:

dev: Starts the development server.
build: Creates an optimized production build.
preview: Previews the production build.

package-lock.json

The package-lock.json file is automatically generated when you run npm install in a Node.js project, including projects created with React and Vite.

The package-lock.json file helps ensure consistent dependency management across different environments and among team members. Its main roles are to:

- Lock Dependency Versions: It locks the exact versions of each package and its dependencies. This ensures that every time you or someone else installs the project, the exact same dependency versions are used, avoiding unexpected issues caused by version updates.

- Improve Installation Speed: package-lock.json allows npm to skip version resolution during installation, making it faster since npm knows exactly which versions to install

NOTE: `package.json` specifies the dependencies a project requires, `package-lock.json` provides an exact dependency tree.

vite.config.js

This configuration file is where you can customize Vite's behavior. For example, you might specify a different public base path, define environment variables, or configure plugins (like React or TypeScript support).

README.md

The README provides an overview of your project, instructions for setup, usage, and any other relevant details for future developers or users.

Environment Variables: Vite uses .env files for environment variables, such as .env or .env.production.

Routing and State Management: If your app grows, you might add folders for features like routing (e.g., react-router-dom) or state management (e.g., Redux, Context API).

D. Key Concepts of React.js

1. **Component-Based Architecture**
    - **Components**: React applications are made up of components, which are small, reusable pieces of code that represent parts of the user interface (UI). Each component can have its own state and logic.
    - **Functional Components**: These are simple JavaScript functions that accept props (inputs) and return JSX (UI elements).

Example of a simple **functional component**:

```
import React from "react";
const Greeting = ({ name }) => {
  return <h1>Hello, {name}!</h1>;
};
export default Greeting;
```

2. **JSX (JavaScript XML)**
    - **JSX** is a syntax extension that allows writing HTML-like code inside JavaScript. It makes it easier to write UI logic alongside JavaScript logic.
    - JSX is not a necessity but is commonly used in React applications to improve readability.

Example of JSX:

```
const element = <h1>Hello, world!</h1>;
```

3. **Virtual DOM**
    - React uses a **virtual DOM**, which is an in-memory representation of the real DOM. When the state of a component changes, React updates the virtual DOM first, then efficiently updates only the parts of the actual DOM that have changed.
    - This improves performance by minimizing expensive DOM manipulations.
4. **One-Way Data Binding**

- React enforces a unidirectional data flow. The data flows from parent components to child components through **props** (properties). This makes tracking data changes and debugging easier.

Example of passing props:

```
const ParentComponent = () => {
  const name = "Alice";
  return <ChildComponent name={name} />;
};

const ChildComponent = ({ name }) => {
  return <p>{name}</p>;
};
```

5. **State Management**
   - React components can have **state**, which is an object that holds data that may change over time. The state is managed using the useState hook in functional components or the setState method in class components.

Example using the useState hook:

```
import React, { useState } from "react";
const Counter = () => {
  const [count, setCount] = useState(0);

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

export default Counter;
```

6. **React Hooks**
    - **Hooks** are special functions that allow functional components to use React features such as state and lifecycle methods.
        - useState: Manage local component state.
        - useEffect: Manage side effects like data fetching, subscriptions, etc.
        - useContext: Access data from React Context API (global state management).
        - useReducer: Manage more complex state logic, similar to Redux.

7. **Props (Properties)**
    - **Props** are the way to pass data from parent components to child components. Props are read-only and cannot be modified by the child component. In React, props (short for "properties") are used to pass data from parent components to child components. There are several different ways to pass and use props in React, ranging from basic prop passing to more advanced techniques involving destructuring, default props, and using functions as props. Below are various methods to work with props in React:

Basic Prop Passing

This is the simplest way to pass props. In the parent component, you pass the props as attributes to the child component.

Example:

```
// Parent Component
const Parent = () => {
  return <Child name="John" age={30} />;
};

// Child Component
const Child = (props) => {
  return (
    <div>
      <h1>Name: {props.name}</h1>
      <p>Age: {props.age}</p>
```

```
    </div>
  );
};
```

Destructuring Props

Instead of accessing props with props.name, you can destructure the props for cleaner code.

Example:

```
// Parent Component
const Parent = () => {
  return <Child name="John" age={30} />;
};

// Child Component with Destructuring
const Child = ({ name, age }) => {
  return (
    <div>
      <h1>Name: {name}</h1>
      <p>Age: {age}</p>
    </div>
  );
};
```

Passing Objects as Props

You can pass an entire object as a prop, which is helpful when passing multiple related values.

Example:

```
// Parent Component
const Parent = () => {
  const user = { name: "John", age: 30 };
  return <Child user={user} />;
};

// Child Component
```

```
const Child = (props) => {
  return (
    <div>
      <h1>Name: {props.user.name}</h1>
      <p>Age: {props.user.age}</p>
    </div>
  );
};
```

You can also destructure the object inside the child component:

```
const Child = ({ user: { name, age } }) => {
  return (
    <div>
      <h1>Name: {name}</h1>
      <p>Age: {age}</p>
    </div>
  );
};
```

4. Default Props

React allows you to define default props in case the parent component does not pass certain props.

Example:

```
// Child Component
const Child = ({ name, age }) => {
  return (
    <div>
      <h1>Name: {name}</h1>
      <p>Age: {age}</p>
    </div>
  );
};

// Default Props
Child.defaultProps = {
```

```
  name: "Default Name",
  age: 18,
};

// Parent Component
const Parent = () => {
  return <Child />;
};
```

If the Parent does not pass any props, the Child component will use the default values.

5. Passing Functions as Props (Callback Props)

You can pass a function from the parent component to the child component, which the child can call.

Example:

```
// Parent Component
const Parent = () => {
  const handleClick = () => {
    alert("Button clicked!");
  };

  return <Child onButtonClick={handleClick} />;
};

// Child Component
const Child = ({ onButtonClick }) => {
  return <button onClick={onButtonClick}>Click Me</button>;
};
```

In this example, clicking the button in the Child component will trigger the handleClick function in the Parent.

Props with JSX Spread Operator

The spread operator (...) can be used to pass multiple props at once, especially when passing an object or many props.

Example:

```
const Parent = () => {
```

```
  const user = { name: "John", age: 30, occupation: "Developer" };
  return <Child {...user} />;
};

// Child Component
const Child = ({ name, age, occupation }) => {
  return (
    <div>
      <h1>Name: {name}</h1>
      <p>Age: {age}</p>
      <p>Occupation: {occupation}</p>
    </div>
  );
};
```

In this case, all properties of the user object are passed to the Child component as individual props.

Conditional Rendering with Props

Props can be used to conditionally render elements in a child component based on the data passed from the parent.

Example:

```
const Child = ({ isLoggedIn }) => {
  return (
    <div>
      {isLoggedIn ? <p>Welcome back!</p> : <p>Please log in.</p>}
    </div>
  );
};

// Parent Component
const Parent = () => {
  return <Child isLoggedIn={true} />;
};
```

In this example, the child component conditionally renders based on the isLoggedIn prop.

Passing Children as Props

In React, you can pass JSX elements or components as props.children. This is called **children props**.

Example:

```
// Parent Component
const Parent = () => {
  return (
    <Child>
      <p>This is passed as children props</p>
    </Child>
  );
};

// Child Component
const Child = (props) => {
  return <div>{props.children}</div>;
};
```

In this example, the <p> element is passed to the Child component as children and rendered inside it.

Summary of Different Ways to Use Props in React:

1. **Basic Prop Passing**: Pass props as attributes.
2. **Destructuring Props**: Cleaner way to access props.
3. **Passing Objects**: Pass an object as a single prop.
4. **Default Props**: Set default values for props.
5. **Functions as Props**: Pass callback functions as props.
6. **Spread Operator**: Pass multiple props with the spread operator.
7. **Conditional Rendering**: Conditionally render based on props.
8. **Children Props**: Pass JSX elements or components as children.

Example of passing props:

```
const App = () => {
  return <Welcome name="John" />;
};

const Welcome = (props) => {
  return <h1>Hello, {props.name}</h1>;
};
```

8. **Lifecycle Methods**
    - React components go through a series of lifecycle stages such as mounting, updating, and unmounting. These can be controlled with lifecycle methods (for class components) or useEffect (for functional components).
    - In modern React with functional components, useEffect replaces most lifecycle methods such as componentDidMount, componentDidUpdate, and componentWillUnmount.
9. **Routing with React Router**
    - React Router is a library that helps manage routes and navigation in a React app. It allows you to build single-page applications (SPAs) with different views or pages based on the URL path.

Example with React Router:

```
import { BrowserRouter as Router, Route, Routes } from "react-router-dom";
import Home from "./Home";
import About from "./About";

const App = () => {
  return (
    <Router>
      <Routes>
        <Route path="/" element={<Home />} />
        <Route path="/about" element={<About />} />
      </Routes>
    </Router>
  );
};

export default App;
```

State Management with Context and External Libraries

- For global state management, React provides the **Context API**. Additionally, developers often use external state management libraries like **Redux** or **MobX** for complex state management.

Example using Context:

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

const ThemeContext = createContext();

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

  return (
```

```
    <ThemeContext.Provider value={{ theme, setTheme }}>
      <Header />
    </ThemeContext.Provider>
  );
};

const Header = () => {
  const { theme, setTheme } = useContext(ThemeContext);

  return (
    <div>
      <h1>Current Theme: {theme}</h1>
      <button onClick={() => setTheme(theme === "light" ? "dark" : "light")}>
        Toggle Theme
      </button>
    </div>
  );
};

export default App;
```