Introuction to React JS

By

Narasimha Rao T

Microsoft.Net FSD Trainer

Professional Development Trainer

tnrao.trainer@gmail.com

Index

Introduction to Client-Side Technologies

- **Definition**: Client-side technologies are tools and frameworks that run on the user's browser rather than the server.
- **Purpose**:
  - Improve **user experience** with interactive UI.
  - Reduce **server load** by handling rendering and logic on the client side.
  - Enable **faster updates** and dynamic web applications.

Significance of Client-Side Technologies in Modern Web Development

- Create **rich, dynamic, and responsive** applications.
- Enhance **performance** by minimizing server round trips.
- Support **Single Page Applications (SPAs)**.
- Allow **component reusability** for faster development.
- Enable cross-platform compatibility with mobile and desktop apps.

Examples of Client-Side Technologies

- **HTML, CSS, JavaScript** (core building blocks).
- **Frameworks & Libraries**:
  - Angular
  - React
  - Vue.js
  - Bootstrap
  - jQuery

What is React?

- **React**: A JavaScript library for building **user interfaces**.
- Developed and maintained by **Facebook (Meta)**.
- Focuses on **component-based architecture** and efficient **DOM updates** with the **Virtual DOM**.

History of React

- **2011**: Created by Jordan Walke (Facebook engineer).
- **2013**: Open-sourced to the public.
- **2015 onwards**: Adoption grew due to its simplicity and efficiency.
- Now one of the most widely used **frontend libraries**.

React Features

- **Component-based**: Build UI with reusable components.
- **Virtual DOM**: Efficient rendering by updating only changed elements.
- **JSX (JavaScript XML)**: Write HTML-like syntax inside JavaScript.
- **One-way Data Binding**: Ensures data flow is predictable.
- **Fast Rendering**: Optimized UI updates.
- **Strong Community**: Large ecosystem of tools and libraries.

React Component-Based Architecture

- **Component** = Independent, reusable piece of UI.
- Types:
  - **Functional Components** (preferred).
  - **Class Components**.
- Components can be **nested**, **reused**, and **managed** easily.

Set Up First React Project

- Install **Node.js** (includes npm).
- Use **create-react-app** or alternatives like **Vite**.
- Start development server and modify **App.js**.

Folder Structure Overview

```
my-app/
│
├── node_modules/   → Dependencies
├── public/         → Static files (index.html, favicon)
├── src/            → React code
│   ├── App.jsx      → Main component
│   ├── main.js      → Entry point (index.js in CRA)
│   └── App.css     → Styling
├── package.json    → Project metadata & dependencies
```

Create a Simple “Hello World” Component

```
// HelloWorld.js
import React from "react";

function HelloWorld() {
  return <h1>Hello, World!</h1>;
}

export default HelloWorld;
```

```
// In App.js
import HelloWorld from "./HelloWorld";

function App() {
  return (
    <div>
      <HelloWorld />
    </div>
  );
}

export default App;
```

What is JSX?

- **JSX** = JavaScript XML.
- Syntax extension that lets us write **HTML-like code** inside JavaScript.
- Example:

```
const element = <h1>Hello JSX!</h1>;
```

JSX Syntax and Rules

- **Must return a single root element**.
- Use **className** instead of `class`.
- JSX expressions enclosed in `{}`.
- Elements must be **properly closed**.
- Example:

```
function Greeting() {
  const name = "John";
  return <h2>Hello, {name}</h2>;
}
```

Understanding Virtual DOM in React JS

Component Naming and File Conventions

- **Naming**:
  - Use PascalCase for component names (e.g., `MyComponent`).
  - Avoid reserved JavaScript words or generic terms like `Component`.
  - Names should reflect the component's purpose (e.g., `UserProfile`, `Navbar`).
- **File Conventions**:
  - Store each component in its own file, typically in a `components/` folder.
  - File names match the component name (e.g., `MyComponent.jsx`).
  - Use `.jsx` or `.js` extension for React components.
  - Group related components in subfolders (e.g., `components/Navbar/Navbar.jsx`).

- Example structure:

  ```
  src/
    components/
      Welcome.jsx
      Navbar/
        Navbar.jsx
        NavbarItem.jsx
  ```

Props in React

What are Props in React?

Props (short for properties) are read-only inputs passed to components to customize their behavior or rendering. They allow components to be reusable and dynamic.

- **Characteristics**:
    - Immutable within the receiving component.
    - Passed as attributes in JSX.
    - Can include data, functions, or other components.

Passing Props to Components

- Props are passed as attributes in JSX and accessed as an object in the component.
- Example:

```
function Greeting({ name }) {
  return <h1>Hello, {name}!</h1>;
}
function App() {
  return <Greeting name="Alice" />;
}
```

- Passing Multiple Props:

```
function UserCard({ name, age, email }) {
  return (
    <div>
      <p>Name: {name}</p>
      <p>Age: {age}</p>
      <p>Email: {email}</p>
    </div>
  );
}
function App() {
  return <UserCard name="Bob" age={25} email="bob@example.com" />;
}
```

Passing Event Handlers as Props

Event handlers can be passed as props to child components for handling events in a parent component.

- **Example**:

```
function Button({ onClick, label }) {
  return <button onClick={onClick}>{label}</button>;
}
function App() {
  const handleClick = () => {
    console.log('Button clicked from parent!');
  };
  return <Button onClick={handleClick} label="Click Me" />;
}
```

Q & A

Narasimha Rao T
tnrao.trainer@gmail.com