ReactJS

Introduction to React & JSX

What is React?

React is a JavaScript library used to build user interfaces (UIs), particularly for single-page applications (SPAs). It helps developers create large web applications that can update and render efficiently in response to data changes.

Explanation:

- **Declarative:** You describe what the UI should look like, and React efficiently updates it to match your description. You focus on the *what*, not the *how*.
- **Component-Based:** UIs are broken into small, independent, and reusable pieces called components. This promotes modularity and easier management of complex UIs.
- **Virtual DOM:** React creates a lightweight, in-memory copy of the real browser DOM. When data changes, React first updates this Virtual DOM, then efficiently calculates the minimal changes needed to update the *real* DOM, leading to better performance.
- **Unidirectional Data Flow:** Data typically flows one-way, from parent components to child components, making data management predictable and debugging easier.

Why React?

- **Reusability:** Write components once and reuse them across different parts of your application, or even in different projects.
- **Performance:** React's Virtual DOM and efficient reconciliation algorithm ensure fast and smooth UI updates.
- **Ecosystem:** React boasts a massive and active community, a rich set of libraries, tools (like React Developer Tools), and extensive learning resources, which also translates to high job demand.
- **Simplified Development:** Its declarative nature and component-based approach simplify the development of complex UIs.

Setting up a React Project (Brief Overview)

Before we dive into code, it's good to know how React projects are typically started. We'll be using tools like **Create React App** or **Vite** to quickly set up our development environment. These tools handle all the necessary configurations and compilations (like transforming JSX into browser-understandable JavaScript) for us, allowing us to focus purely on React code.

What is JSX?

JSX (JavaScript XML) is a syntax extension for JavaScript. It allows you to write HTML-like syntax directly within your JavaScript code. While it looks like HTML, it's not valid JavaScript on its own; it gets compiled into regular JavaScript (specifically, `React.createElement()` calls) by a build tool like Babel before the browser executes it.

Example: `const element = <h1>Hello, React!</h1>;`

Explanation: JSX makes it much easier to visualize UI structures within JavaScript code, making your components more readable and intuitive. Every JSX expression must follow these rules:

1. **Return a Single Parent Element:** All JSX elements returned from a component must be wrapped within a single parent tag.
    - **Common Solution: React Fragments (`<>...</>`)** This is the preferred way to group multiple elements without adding an unnecessary extra `div` to your actual DOM.

```
// GOOD: Using a React Fragment (empty tags) - no extra div in the DOM
function Welcome() {
  return (
    <>
      <h1>Welcome to React</h1>
      <p>This is a JSX-based functional component</p>
    </>
  );
}

// Also GOOD: Using a div if you need a container anyway
// function Welcome() {
//   return (
//     <div>
//       <h1>Welcome to React</h1>
//       <p>This is a JSX-based functional component</p>
//     </div>
//   );
// }
```

2. **Use `camelCase` for Attributes:** HTML attributes like `onclick` become `onClick`, and `class` becomes `className` in JSX.
3. **Self-Close Tags:** Tags that don't have children (like `<img>`, `<input>`, `<br>`) must be self-closed with a `/` (e.g., `<img />`, `<input />`).

JSX Hands-on Demo

Functional Component Version:

```
function Welcome() {
  return (
    <> {/* Using a React Fragment here */}
      <h1>Welcome to React</h1>
      <p>This is a JSX-based functional component</p>
    </>
  );
}
```

Class Component Version:

```
import React, { Component } from 'react'; // Import Component from React

class Welcome extends Component {
  render() {
    return (
      <div>
        <h1>Welcome (Class Component)</h1>
        <p>Rendered using React.Component</p>
      </div>
    );
  }
}
```

Components & Props

What is a Component?

A component is an independent, reusable building block in React that encapsulates its own logic and returns a piece of UI (JSX). They are the core of React's modular architecture.

- **Functional Components:** These are simply JavaScript functions that accept `props` as an argument and return JSX. They are the preferred way to write new components in modern React due to hooks and their simplicity.
- **Class Components:** These are ES6 classes that extend `React.Component`. They have a `render()` method that returns JSX and can manage their own internal state and lifecycle methods. You'll encounter these in older codebases.

Explanation: Components allow for modular and maintainable development. You can break down complex UIs into smaller, manageable units (e.g., a header, a footer, a sidebar, a form, a button), each handling its own piece of logic and rendering. This makes your application easier to understand, test, and debug.

Functional Component Example:

```
function Greeting() {
  return <h2>Hello from Functional Component</h2>;
}
```

Class Component Example:

```
import React from 'react'; // Ensure React is imported

class Greeting extends React.Component {
  render() {
    return <h2>Hello from Class Component</h2>;
  }
}
```

What are Props?

Props (short for **properties**) are a mechanism for passing data from a parent component to a child component. Think of them as arguments you pass to a function.

- They are **read-only**: A child component should **never modify its received props**. This maintains the unidirectional data flow and predictability of your application. If a child needs to affect the parent, it does so through callbacks (which we'll see later).
- They help make components reusable with different data, as demonstrated in the `UserCard` example below.

Props Example:

```
function Welcome(props) { // props is an object containing all passed
attributes
  return <h2>Hello, {props.name}</h2>;
}

// Usage in a parent component:
function App() {
  return <Welcome name="Farhan" />; // 'name' becomes a property on the props
object
}
```

Props + Card Component Demo:

```
function UserCard({ name, role }) { // Using object destructuring for cleaner
access to props
  return (
    <div className="card">
      <h3>{name}</h3>
      <p>Role: {role}</p>
    </div>
  );
}

function App() {
  return (
    <div>
      <UserCard name="Ali" role="Developer" />
      <UserCard name="Sara" role="Designer" />
      <UserCard name="John Doe" role="Project Manager" />
    </div>
  );
}
```

This shows the power of reusability, where you pass different data into the same `UserCard` component to render distinct UIs. Notice how we used **object destructuring (`{ name, role }`)** in the `UserCard` function signature. This is a very common and cleaner way to extract specific props from the `props` object, avoiding repetitive `props.name` syntax.

Routing with React Router

Why Routing?

React applications are typically Single-Page Applications (SPAs). This means there's generally only one HTML file (`index.html`) loaded. However, users expect to navigate between different "pages" or views within the application (e.g., a "Home" page, an "About" page, a "Contact" page). React Router helps simulate this multi-page navigation experience within a SPA by changing the URL and rendering different components based on the path.

Install React Router:

First, you need to add React Router to your project: `npm install react-router-dom`

Basic Routing Structure:

React Router provides components to define your routes and link to them.

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

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

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

```
function App() {
  return (
    <BrowserRouter> {/* Wrapper for all routing components */}
      <Routes> {/* Defines a collection of routes */}
        <Route path="/" element={<Home />} /> {/* Renders Home component when
path is '/' */}
        <Route path="/about" element={<About />} /> {/* Renders About
component when path is '/about' */}
      </Routes>
    </BrowserRouter>
  );
}
```

- `BrowserRouter`: The recommended router for web browsers. It uses the HTML5 history API to keep your UI in sync with the URL.
- `Routes`: A wrapper for individual `Route` components. It looks through its children `Route`s and renders the first one that matches the current URL.
- `Route`: Defines a mapping between a URL path and a component to render.
    - `path`: The URL path to match.
    - `element`: The React component to render when the path matches.

Navigation using `<Link>` & `<NavLink>`:

To navigate between pages without full page reloads, we use specific components provided by React Router.

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

function Navbar() {
  return (
    <nav>
      {/* <Link>: Basic link, similar to <a>, but prevents full page reload
*/}
      <Link to="/">Home</Link> |

      {/* <NavLink>: A special <Link> that can add styling to active links
*/}
      <NavLink
        to="/about"
        className={({ isActive }) => isActive ? "active-link" : ""}
      >
        About
      </NavLink>
      {/* You would typically define 'active-link' in your CSS to highlight
the active page */}
```

```
    </nav>
  );
}
```

- `<Link>`: This component is similar to a standard HTML `<a>` tag, but it prevents the default browser refresh when clicked, allowing React Router to handle the navigation client-side.
- `<NavLink>`: A special version of `<Link>` that adds styling attributes (like an `active-link` class) to the rendered element when it matches the current URL. This is very useful for highlighting the current page in a navigation bar.

Dynamic Routing with `useParams` (Brief Mention):

For routes that depend on dynamic data, like viewing a specific user profile (e.g., `/users/123`), React Router provides the `useParams` hook. You define a dynamic segment in your `Route` path using a colon (e.g., `path="/users/:userId"`), and then use `useParams` in the component rendered by that route to extract the value (e.g., `const { userId } = useParams();`). We'll explore this more in a later session if time permits.

Routing Demo: Multi-Page Site

```
function App() {
  return (
    <BrowserRouter>
      <Navbar /> {/* Our navigation bar */}
      <Routes>
        <Route path="/" element={<Home />} />
        <Route path="/about" element={<About />} />
        {/* You can add more routes here, e.g., <Route path="/contact"
element={<Contact />} /> */}
      </Routes>
    </BrowserRouter>
  );
}
```

State with `useState` & Event Handling

What is State?

State is data that is managed *within* a component and can change over time. Unlike props, which are passed from the outside and are read-only, state is internal to the component and can be updated by the component itself.

- When a component's state changes, React automatically re-renders that component (and its children) to reflect the new data, keeping your UI in sync.
- For functional components, the `useState` Hook is the primary way to declare and update state.

`useState` Hook Example:

```
import { useState } from 'react'; // Always import useState

function Counter() {
  // useState returns an array: [current state value, function to update
state]
  const [count, setCount] = useState(0); // Initialize count with 0

  return (
    <div>
      <h2>Count: {count}</h2>
      {/* Event handler calls setCount to update the state */}
      <button onClick={() => setCount(count + 1)}>Increment</button>
    </div>
  );
}
```

Explanation: Immutability of State A crucial concept when working with state, especially arrays and objects, is **immutability**. When you update state, you should **never directly modify (mutate) the existing state object or array**. Instead, always create a **new** object or array with the desired changes and then pass that new one to the state update function (`setCount`, `setItems`, etc.).

Why? React relies on changes in object/array references to detect that state has been updated and a re-render is needed. If you mutate the original object/array, its reference remains the same, and React might not detect the change, leading to your UI not updating.

```
// BAD Example (Mutating state directly - UI might not update reliably)
// function ShoppingList() {
//   const [items, setItems] = useState(['Milk', 'Bread']);
//   const addItem = () => {
//     items.push('Eggs'); // DIRECT MUTATION!
//     setItems(items); // React might not see a change in 'items' reference
//   };
//   return (/* ... */);
// }

// GOOD Example (Creating a new array for state update)
function ShoppingList() {
  const [items, setItems] = useState(['Milk', 'Bread']);
  const addItem = () => {
    // Creates a NEW array using spread syntax (...items) and adds 'Eggs'
    setItems([...items, 'Eggs']);
  };
  return (
    <div>
      <h3>Shopping List</h3>
      <ul>
        {items.map((item, index) => <li key={index}>{item}</li>)}
      </ul>
      <button onClick={addItem}>Add Eggs</button>
    </div>
  );
}
```

Asynchronous Nature of State Updates: State updates (like `setCount(count + 1)`) can sometimes be asynchronous and batched by React for performance reasons. This means that if you `console.log(count)` immediately after `setCount(count + 1)`, you might still see the *old* value of `count`. React guarantees that the UI will eventually reflect the latest state.

Toggle Demo (Show/Hide Element with Conditional Rendering):

This demo combines `useState` with **conditional rendering** (displaying elements based on a condition).

```
function Toggle() {
  const [show, setShow] = useState(true); // State to control visibility

  return (
    <div>
      <button onClick={() => setShow(!show)}> {/* Toggles the 'show' state
*/}
        {show ? "Hide" : "Show"} Text {/* Button text changes based on state
*/}
      </button>
      {show && <p>This is visible text.</p>} {/* Renders <p> only if 'show'
is true */}
    </div>
  );
}
```

Event Handling in Functional Component:

React events are named using camelCase (`onClick`, `onChange`). You pass a function as the event handler.

```
function Clicker() {
  function handleClick() { // Define the event handler function
    alert("You clicked me!");
  }

  return <button onClick={handleClick}>Click</button>; // Pass the function
reference
}
```

Event Handling in Class Component:

In class components, you often define event handlers as methods of the class. You need to be mindful of `this` context (though with arrow functions, it's less of an issue).

```
import React from 'react';

class Clicker extends React.Component {
  handleClick() { // Method for event handling
    alert("Class component button clicked!");
  }

  render() {
    // When passing event handlers, often bind `this` if not using arrow
functions
    // For simplicity, using a direct method call here assumes context
binding
    // or class field syntax in a real-world scenario.
    return <button onClick={() => this.handleClick()}>Click Me
(Class)</button>;
    // Or, more commonly with class properties:
    // handleClick = () => { alert("..."); }
    // <button onClick={this.handleClick}>Click Me (Class)</button>
  }
}
```

Preventing Default Behavior with Events (`e.preventDefault()`): Many browser events have default behaviors (e.g., submitting a form reloads the page, clicking a link navigates away). In React, you can prevent these default behaviors using `event.preventDefault()`.

```
function MyForm() {
  const handleSubmit = (e) => {
    e.preventDefault(); // Prevents the default browser form submission (page
reload)
    alert("Form submitted!");
    // You would typically process form data here
  };

  return (
    <form onSubmit={handleSubmit}>
      <input type="text" placeholder="Enter something" />
      <button type="submit">Submit Form</button>
    </form>
  );
}
```

Summary Checklist (Day-1)

- What is React and why it's powerful (Declarative, Component-Based, Virtual DOM).
- Basic project setup awareness (Create React App/Vite).
- JSX syntax, rules, and using React Fragments (`<>...</>`).
- Understanding of Functional vs. Class components.
- Props for data communication (read-only, destructuring).
- React Router for navigation (`BrowserRouter`, `Routes`, `Route`, `Link`, `NavLink`, brief `useParams`).
- `useState` Hook for managing component state.
- Crucial concept of **immutability** when updating state.
- Basic Event Handling in both functional and class components.
- `event.preventDefault()` for controlling browser defaults.
- Real-world demos: Card display, toggler, counter, navbar router, basic form submit.