1. React Installation and Setup

What

In this section, we'll set up a basic React application using a tool called Vite. Vite is a modern, fast way to create React apps with a simpler project structure and faster development times. Setting up React correctly is the first step to building any React app, as it creates the environment we need to write and run React code.

Why

React doesn't run on its own in the browser; it needs a build environment to transform modern JavaScript and JSX (JavaScript XML) into code that browsers can understand. Setting up the project also gives us a ready-made folder structure and a development server that makes it easy to see changes as we code.

How

Let's set up a React project step-by-step. We'll start by installing Node.js and npm (if you don't have them already), then use Vite to create our React project, and finally explore the key files in the project setup.

Step 1: Install Node.js and npm

Node.js is a JavaScript runtime, and npm (Node Package Manager) helps you install packages like React. To check if you have Node.js and npm installed:

1. Open your terminal (Command Prompt on Windows, Terminal on macOS/Linux).

2. Run the following commands to check if Node and npm are installed:

```
node -v
```

```
npm -v
```

3. If you see version numbers, you're good to go! If not, [download Node.js](https://nodejs.org/) and install it.

Step 2: Set Up a React App Using Vite

1. Navigate to Your Project Folder: In your terminal, navigate to the folder where you want to create the project.

In Terminal-

```
cd path/to/your/folder
```

2. Create the Project: Now, run this command to set up a new React app using Vite.

In Terminal-

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

- `my-react-app` is the name of your project folder (you can change it to anything you like).

- `--template react` tells Vite to set up a React project template.

You'll see a prompt asking for a package name. You can press `Enter` to accept the default.

3. Navigate to Your Project Folder: Move into the project directory.

In Terminal-

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

4. Install Dependencies: Install the necessary packages by running:

In Terminal-

```
npm install
```

5. Start the Development Server: Once the setup is complete, start your development server.

In Terminal-

```
npm run dev
```

6. Open the App in Your Browser: You should see a message in the terminal with a local URL (usually `http://localhost:5173`). Open this URL in your browser to see your new React app running!

Step 3: Understand the Project Structure

Let's look at the most important files in your new project:

- index.html: This file is in the `public` folder and is the single HTML file that React uses. There's only one HTML file because React is a single-page application (SPA) framework.

- src/main.jsx: This is the entry point for your React app. Here, React's core library and the `App` component are connected to the `index.html` file by telling React where to render the app.

```
import React from 'react';
import ReactDOM from 'react-dom/client';
import App from './App';

ReactDOM.createRoot(document.getElementById('root')).render(
  <React.StrictMode>
    <App />
  </React.StrictMode>
);
```

- `ReactDOM.createRoot` tells React where to render the app—in this case, it's the `root` element in `index.html`.

- src/App.jsx: This is the main React component of the app. Here's what a simple `App` component might look like:

```
function App() {
  return (
    <div>
      <h1>Hello, React!</h1>
      <p>This is my first React app.</p>
    </div>
  );
}
export default App;
```

- `App` is a functional component, a key concept in React. We'll be building many of these to create different parts of our app.

- package.json: This file manages the project's dependencies (like React) and scripts (like `npm run dev`).

Conclusion

After following these steps, you have a working React environment ready for development! This setup gives you everything you need to start creating components, managing state, and adding interactivity in your React app. In the next section, we'll dive deeper into creating and styling components.

2. Creating Components and Styling

What

In React, a component is like a building block. Each component represents a part of the UI, like a button, a header, or a section of content. React apps are often built using many small, reusable components.

Why

Components help you split your UI into independent, manageable pieces. By reusing components, you keep your code organized and make it easy to maintain. Styling these components is equally important, as it makes your app visually appealing and enhances the user experience.

How

We'll create and style a simple React component. Along the way, we'll explore different styling methods: inline styles, CSS files, and CSS modules.

Step 1: Create a Simple Component

1.1. Create a New Component File: Inside the `src` folder, create a new file called `Greeting.jsx`. This file will hold a reusable component that displays a simple greeting message.

1.2. Define the Component: In `Greeting.jsx`, write the following code:

```
import React from 'react';
function Greeting() {
  return (
    <div>
      <h2>Hello, World!</h2>
      <p>Welcome to your first custom React component.</p>
    </div>
  );
}
export default Greeting;
```

Here, `Greeting` is a functional component that returns a small piece of JSX, which will be rendered on the screen.

Components in React must start with a capital letter, so `Greeting` is correct, while `greeting` would not work.

1.3. Use the Component: Open `App.jsx` (the main component) and import `Greeting` to use it:

```
import React from 'react';
import Greeting from './Greeting';
function App() {
 return (
  <div>
   <h1>Main App</h1>
   <Greeting />
  </div>
 );
}
export default App;
```

Here, we use `<Greeting />` to include our custom `Greeting` component inside `App`.

1.4. Run the App: If your development server is running, you should see the `Greeting` component in the browser.

Step 2: Styling the Component

React allows for multiple ways to style components. We'll go over a few of the most common methods.

Method 1: Inline Styles

You can apply styles directly within your JSX using a `style` attribute. Here's how you might add inline styles to the `Greeting` component.

1.1 Modify `Greeting.jsx` to use inline styles:

```
function Greeting() {
  return (
    <div style={{ textAlign: 'center', padding: '20px' }}>
      <h2 style={{ color: 'blue', fontSize: '24px' }}>Hello, World!</h2>
      <p style={{ color: 'gray' }}>Welcome to your first custom React component.</p>
    </div>
  );
}
```

Inline styles in React are written as JavaScript objects, with style properties written in camelCase (e.g., `fontSize` instead of `font-size`).

Each style value is a string (e.g., `'center'`, `'24px'`).

Method 2: External CSS File

Using an external CSS file keeps styling separate from the component code, which is more readable for larger projects.

2.1. Create a CSS File: Create a file called `Greeting.css` in the same folder as `Greeting.jsx`.

2.2. Add Some Styles: In `Greeting.css`, add the following styles:

```
.greeting-container {
  text-align: center;
  padding: 20px;
}

.greeting-title {
  color: blue;
  font-size: 24px;
}

.greeting-text {
  color: gray;
}
```

2.3. Import the CSS File in `Greeting.jsx` and apply the classes:

```
import React from 'react';
import './Greeting.css';
function Greeting() {
  return (
    <div className="greeting-container">
      <h2 className="greeting-title">Hello, World!</h2>
      <p className="greeting-text">Welcome to your first custom React component.</p>
    </div>
  );
}
export default Greeting;
```

Now, the component uses classes from `Greeting.css`. CSS class names are applied with `className` instead of `class` in JSX.

3: CSS Modules

CSS Modules are scoped locally by default, so they prevent styles from accidentally affecting other components. This is useful in larger projects.

1. Rename `Greeting.css` to `Greeting.module.css`.

2. Update the Stylesheet: The styles in `Greeting.module.css` will remain the same.

3. Import the Module in `Greeting.jsx`:

```
import React from 'react';
import styles from './Greeting.module.css';

function Greeting() {
 return (
  <div className={styles.greetingContainer}>
   <h2 className={styles.greetingTitle}>Hello, World!</h2>
   <p className={styles.greetingText}>Welcome to your first custom React component.</p>
  </div>
 );
}
export default Greeting;
```

Here, `styles` is an object containing all the CSS classes, scoped locally. For example, `styles.greetingContainer` maps to the `.greeting-container` class in CSS.

Summary

Now, you have a fully functional `Greeting` component, styled in different ways:

- Inline styling for quick, one-off styles.

- External CSS for clean, centralized styles.

- CSS Modules for scoped, component-specific styling.

This gives you a flexible foundation for styling your components in ways that best suit your project needs.

Let’s move to the next concept where we’ll look into passing data to components using props!

3. Props and Prop Validation

What

Props (short for "properties") are a way to pass data from one component to another in React. They are a core part of how components communicate, allowing us to make components dynamic and reusable. **Prop validation** helps ensure that props passed to a component have the expected type and structure, preventing bugs and making our code more predictable.

Why

Props make components reusable by allowing them to behave differently depending on the data they receive. For instance, we could use the same button component multiple times but give it different text labels and actions using props. Prop validation adds an extra layer of error-checking, helping us catch potential issues if a prop is missing or has the wrong type.

How

Let’s dive into using props step-by-step by creating a **Profile Card** component that receives data as props, then validate those props to ensure the component works correctly.

Step 1: Create a Component with Props

1.1 Create a New Component File: Inside the `src` folder, create a file called `ProfileCard.jsx`.

1.2. Define the Component and Add Props: In `ProfileCard.jsx`, let's define a component that takes three props: `name`, `age`, and `bio`.

```
import React from 'react';

function ProfileCard({ name, age, bio }) {
    return (
      <div style={{ border: '1px solid #ddd', padding: '20px', borderRadius: '8px', width: '250px' }}>
        <h2>{name}</h2>
        <p>Age: {age}</p>
        <p>{bio}</p>
      </div>
    );
  }
export default ProfileCard;
```

Here, `ProfileCard` is a functional component that takes `name`, `age`, and `bio` as props.

We use `{name}`, `{age}`, and `{bio}` within JSX to display the values of these props.

1.3. Pass Props from the Parent Component: Now, let's use `ProfileCard` inside `App.jsx` and pass different data through props.

```
import React from 'react';
import ProfileCard from './ProfileCard';
function App() {
  return (
    <div>
      <h1>User Profiles</h1>
      <ProfileCard name="John Doe" age={30} bio="Loves hiking and outdoor adventures." />
      <ProfileCard name="Jane Smith" age={25} bio="Avid reader and aspiring author." />
    </div>
  );
}
export default App;
```

Here, we pass `name`, `age`, and `bio` as props to each `ProfileCard` instance.

This makes `ProfileCard` reusable for different user profiles by changing the prop values.

Step 2: Validate Props Using PropTypes

PropTypes is a tool that lets you define the expected data types for each prop, which helps prevent errors. If the wrong data type is passed, you'll get a warning in the console (in development mode).

2.1. Install PropTypes: First, we need to install the `prop-types` library to use PropTypes.

```
npm install prop-types
```

2.2. Add Prop Validation in `ProfileCard.jsx`: Import `PropTypes` and define the expected types for each prop.

```
import React from 'react';
import PropTypes from 'prop-types';
function ProfileCard({ name, age, bio }) {
    return (
      <div style={{ border: '1px solid #ddd', padding: '20px', borderRadius: '8px', width: '250px' }}>
        <h2>{name}</h2>
        <p>Age: {age}</p>
        <p>{bio}</p>
      </div>
    );
  }
```

Prop validation

```
ProfileCard.propTypes = {
  name: PropTypes.string.isRequired,
  age: PropTypes.number.isRequired,
  bio: PropTypes.string,
};
export default ProfileCard;
```

- Here, we specify that:
  - `name` should be a string and is required.
  - `age` should be a number and is required.
  - `bio` should be a string but is optional.

3. Test Prop Validation: Now, try passing a wrong prop type (e.g., a string instead of a number for `age`) to see PropTypes in action:

```
<ProfileCard name="Sam" age="twenty-five" bio="Enjoys coding and coffee." />
```

This will produce a console warning that the `age` prop has an invalid type (`string` instead of `number`).

Step 3: Set Default Props (Optional)

We can also set default values for props in case they aren't provided. This way, our component has a fallback value.

1. Define Default Props in `ProfileCard.jsx`:

```
ProfileCard.defaultProps = {
  bio: 'No bio available.',
};
```

Now, if `bio` is not passed, the component will display "No bio available."

Summary

Using props and PropTypes is essential for creating flexible, reusable components in React:

- Props allow us to make components dynamic by passing in data.
- PropTypes add a layer of error-checking that helps us catch issues early.
- Default Props provide fallback values, making components more resilient.

Props and Prop Validation are foundational in React. Next, we'll look into handling user interactions through Event Handling.