

**HERE'S THE**

**LOWDOWN BY**

**HAPPYCODER**

09

|

2024

React Refresher

-- A crash course! --

Jonathan Spurling

PROJECT SPACE TRAINING

[React Refresher: Building a Strong Foundation 4](#_Toc177294261)

[Introduction to React.js 4](#_Toc177294262)

[Example of React’s Usefulness 5](#_Toc177294263)

[Using Vanilla JavaScript: 5](#_Toc177294264)

[Using React.js: 5](#_Toc177294265)

[Key Concepts in React.js 6](#_Toc177294266)

[Getting Started with React 7](#_Toc177294267)

[Setting Up Your React Project 9](#_Toc177294268)

[Our Starting Project 11](#_Toc177294269)

[Guide to Setting Up a React Project with Vite 11](#_Toc177294270)

[Guide to Understanding React JSX Files 12](#_Toc177294271)

[Building Dynamic and Reusable React Components 14](#_Toc177294272)

[Writing Your First Custom React Component 14](#_Toc177294273)

[Core React Principles 17](#_Toc177294274)

[Adding Dynamic Values to React Components 17](#_Toc177294275)

[Reusing Components 18](#_Toc177294276)

[JSX Rules and Restrictions 18](#_Toc177294277)

[Reusing Components with Props in React 21](#_Toc177294278)

[Adding Styling to Components with CSS Modules 23](#_Toc177294279)

[Practice Lesson: Building a Posts List Component 28](#_Toc177294280)

[Summary of the Assignment (Problem): 28](#_Toc177294281)

[Solution to the Problem: Building the Posts List Component 28](#_Toc177294282)

[Introducing State in React 33](#_Toc177294283)

[Adding the Form for a New Post 33](#_Toc177294284)

[Displaying the Form in the App 34](#_Toc177294285)

[Preparing for State Changes 35](#_Toc177294286)

[Understanding State and Event Handling in React 35](#_Toc177294287)

[Setting Up Event Listeners 35](#_Toc177294288)

[Event Listener in React 35](#_Toc177294289)

[Why Event Handling Matters for State 37](#_Toc177294290)

[Understanding React State and Event Handling 37](#_Toc177294291)

[Setting Up Event Listeners in React 37](#_Toc177294292)

[Introducing React’s useState Hook 37](#_Toc177294293)

[How useState Works: 38](#_Toc177294294)

[Using State to Handle User Input 38](#_Toc177294295)

[React Re-Rendering Explained 39](#_Toc177294296)

[Array Destructuring with useState 40](#_Toc177294297)

[Overview of React State and Lifting State Up 40](#_Toc177294298)

[What Happens When You Use useState in React 40](#_Toc177294299)

[The Challenge: Where Should the State Go? 41](#_Toc177294300)

[What is Lifting State Up? 41](#_Toc177294301)

[Steps for Lifting State Up 41](#_Toc177294302)

[Multiple State Variables 42](#_Toc177294303)

[Real-Time UI Updates 43](#_Toc177294304)

[Creating a Modal Component in React 45](#_Toc177294305)

[State and Conditional Content 48](#_Toc177294306)

[Track Modal Visibility with State 48](#_Toc177294307)

[Set Up a Function to Hide the Modal 48](#_Toc177294308)

[Pass the Handler to the Modal 49](#_Toc177294309)

[Conditionally Rendering the Modal 49](#_Toc177294310)

[Adding a Shared Header 51](#_Toc177294311)

[Adding Cancel and Submit Buttons to Form 54](#_Toc177294312)

[Handling Form Submission 59](#_Toc177294313)

[Setting Up Dynamic Posts 63](#_Toc177294314)

[Rendering Posts Dynamically 66](#_Toc177294315)

[Persisting Data in a React App with a Backend 68](#_Toc177294316)

[Building a Full-Stack Post Application with React and Node.js 71](#_Toc177294317)

[Fetching Posts on Initial Page Load in React 76](#_Toc177294318)

[Basic useEffect Structure 77](#_Toc177294319)

[Introduction to Routing in React 82](#_Toc177294320)

[What is Routing? 82](#_Toc177294321)

[React Router 82](#_Toc177294322)

[Setting Up React Router 83](#_Toc177294323)

[Introduction to Routing 83](#_Toc177294324)

[Setting Up React Router 83](#_Toc177294325)

[Introducing Layout Routes 85](#_Toc177294326)

[Steps to Create a Layout Route 85](#_Toc177294327)

[Refactoring the Posts and NewPost Components 87](#_Toc177294328)

[Linking the "New Post" Button to a Modal 92](#_Toc177294329)

[Enhancing Data Fetching with loader()’s 97](#_Toc177294330)

[Submitting Data with React Router 101](#_Toc177294331)

[Using React Router's Form Component 101](#_Toc177294332)

[Handling Data Submission with the Action Function 103](#_Toc177294333)

[Dynamic Routes 105](#_Toc177294334)

[Set up a Dynamic Route 105](#_Toc177294335)

[Fetch Post Data with a Loader 107](#_Toc177294336)

[Display Posts with Links to the Details 107](#_Toc177294337)

[Guide Summary 108](#_Toc177294338)

# React Refresher: Building a Strong Foundation

**Subtitle:** A solid introduction to React, essential for your Next.js journey. While this will provide a strong base, consider deeper dives into React for comprehensive understanding.

**React** is a popular JavaScript library used for building user interfaces. It's known for its component-based architecture, which makes it easier to create reusable and maintainable code.

**Key concepts to review in this refresher include:**

* **JSX:** A syntax extension for JavaScript that allows you to write HTML-like code within your JavaScript.
* **Components:** Building blocks of React applications, encapsulating UI elements and their logic.
* **State and Props:** Mechanisms for managing data and passing it between components.
* **Hooks:** Functions that let you use state and other React features without writing class components.

By understanding these fundamentals, you'll be well-equipped to dive into Next.js and build dynamic web applications. However, to become a React expert, consider exploring more advanced topics such as:

* **Redux:** A state management library for complex React applications.
* **Context API:** A built-in mechanism for sharing data between components.
* **Testing with Jest and React Testing Library:** Ensuring your React components work as expected.
* **Performance optimization techniques:** Improving the speed and responsiveness of your React apps.

## Introduction to React.js

**1. What is React.js?**

React.js is described on its official website as a **JavaScript library for building user interfaces**.

But what does that mean?

* **JavaScript Library**: React allows us to write JavaScript code while enhancing it with tools that help in building **user interfaces** (UI).
* **Highly Interactive UIs**: React is especially helpful when building websites or applications with interactive elements that respond to user inputs or actions.

If you’re building a static or simple webpage with minimal interactivity, you may not need React. However, for more **dynamic and interactive sites** (like the demo project), React vastly simplifies the code needed to create such features.

## Example of React’s Usefulness

**1. Interactive User Interface Example:**

Consider a simple user interface (UI) with a **Contact button**. When clicked, a **modal overlay** appears, allowing the user to enter an address and submit it.

* The website must handle **button clicks**, **modal visibility**, and **user input**.
* While this is a simple example, it illustrates how React helps manage **user interaction** and **reactive behavior** (i.e., the website responding to user actions).

**2. Comparing Code: Vanilla JavaScript vs React.js**

### Using Vanilla JavaScript:

Writing this interaction in plain JavaScript could involve a fair amount of code. Even for a basic example like this, the JavaScript could get quite **verbose** as you add more features and user interactions.

* You would have to handle:
  + Adding event listeners for button clicks.
  + Manually updating the DOM to show/hide the modal.
  + Managing the data input from the form.
  + Handling errors and edge cases.

### Using React.js:

In contrast, building the same user interface with React results in cleaner, more readable, and **manageable** code.

* With React, you can see clear **instructions** such as setShowModal, which shows the modal when certain conditions are met.
* **HTML elements** are blended directly into the JavaScript code, allowing for seamless integration between UI structure and logic.

Even without understanding React fully yet, it's clear that the **React version** of the code is **declarative**, meaning it focuses on **what** should be displayed and React handles the **how** under the hood.

### Key Concepts in React.js

**1. Declarative vs. Imperative Code:**

React allows developers to write **declarative code**.

* **Declarative Code**: You describe the **desired outcome** (e.g., display a modal when the user clicks a button), and React handles the detailed steps to make it happen.
* **Imperative Code**: In contrast, with vanilla JavaScript, you'd write **step-by-step instructions** (e.g., add an event listener, create new HTML elements, manipulate the DOM).

By writing declarative code, you:

* Avoid manual DOM manipulation.
* Write cleaner, more maintainable code.
* Let React handle the specifics of how the UI updates in response to user actions.

**2. JSX: Blending HTML and JavaScript:**

React supports a special syntax called **JSX**, which allows developers to write HTML-like code directly in JavaScript files. This feature simplifies UI development by embedding HTML within JavaScript and makes it easier to manage complex interfaces.

* In the React example, JSX is used to mix **HTML structure** and **JavaScript logic** (e.g., setShowModal) into a single, cohesive code block.

**3. React’s State Management:**

Another powerful feature of React is its ability to manage **state**.

* State refers to variables that determine how a component behaves or displays.
* React allows you to define states (e.g., whether a modal is open or closed) and react to changes in those states, ensuring your UI always reflects the current status of the app.

**Summary**

* **React.js** is a JavaScript library used to create **interactive user interfaces** by simplifying the code required for complex interactions.
* It allows developers to write **declarative code**, reducing the need for step-by-step instructions.
* **JSX** in React enables blending HTML and JavaScript into clean, readable code.
* React handles **state management** to efficiently render UI elements based on user actions.

By using React, you can write more manageable, scalable, and readable code, making it easier to build **powerful and interactive applications**.

## Getting Started with React

**1. Creating a React Project**

When building a web app using **Vanilla JavaScript**, setting up a project is relatively straightforward:

* Create a new folder.
* Add **HTML**, **CSS**, and **JavaScript** files.

However, with **React**, the process is a bit more involved. This is because React requires certain **behind-the-scenes processes** and **tooling** to handle code transformations, development previews, and automatic updates.

**2. Why React Needs Special Setup**

In React, we often **blend HTML and JavaScript** within the same file. This is done for convenience, as it allows developers to write UI elements and logic in a single, cohesive way.

However, **this type of code is not valid JavaScript** in its raw form, so a browser wouldn't be able to understand or execute it. React's syntax must first be **transformed** (or compiled) into **valid JavaScript** that the browser can run. This is where the **React project setup** comes in handy, as it:

* Compiles this mixed code into standard JavaScript.
* Gives you **live previews** of your app while developing.
* **Automatically updates** the preview when you make code changes.

**3. Developer Experience with React**

The React setup isn't just about code compilation. It also provides a **smooth developer experience**:

* **Live Preview**: You can view your app in the browser as you work on it, and the app **automatically reloads** when you make changes to the code.
* **Code Transformation**: React automatically transforms the code you write into **browser-compatible** JavaScript.
* **Deployment-ready**: Once your project is complete, the setup ensures that the final code can be deployed on hosting providers and viewed correctly by users.

**4. Setting Up Your React Project**

To get started with React, you'll typically use tools like **Create React App (CRA)**, which simplifies the process of setting up a React project. CRA handles all the necessary behind-the-scenes tasks:

* **Sets up the environment** for building, previewing, and deploying React applications.
* **Installs essential dependencies** such as **Webpack** (for bundling files) and **Babel** (for compiling modern JavaScript and JSX).
* Provides a **development server** with live reloading, so any changes you make in your code are instantly reflected in your app preview.

**5. Next Steps**

With the project setup complete, you'll have everything needed to:

* **Write React code**, blending HTML and JavaScript.
* Preview your app while you work.
* Ensure the code is compiled into **browser-compatible** JavaScript.

Now you're ready to dive deeper into writing React code and building dynamic, interactive applications.

## Setting Up Your React Project

To get started with React, you'll need to set up a project that supports all the necessary tools and features, such as **auto-reloading**, support for **HTML/JavaScript blending**, and **code transformation**. A popular tool for this is **Create React App (CRA)**.

**1. Tools for React Setup**

Two commonly used tools to set up a React project are:

* **Create React App (CRA)**: This tool automates the setup of a React project with all necessary configurations for auto-reloading, compiling HTML and JavaScript together, and more.
* **Vite**: A modern alternative to CRA that offers faster project setup and better performance, especially for larger projects.

Both tools come with the necessary **development environment** built-in to make coding easier, including features for live reloading and code transformation.

**2. Prerequisite: Node.js**

To use either CRA or Vite, you need to have **Node.js** installed on your system. Node.js is required for:

* Running the project creation tools (CRA or Vite).
* Managing dependencies and running your project.

While you don't need to write or understand Node.js code, the tools rely on Node.js in the background.

You can download the latest version of Node.js from the official website. Choose either the **latest version** or the **LTS (Long-Term Support)** version.

**3. Creating a New React Project**

Once Node.js is installed, you can create a new React project using either CRA or Vite.

* **For Create React App**:
  + Open your terminal and run:

npx create-react-app your-project-name

* **For Vite:**
  + Open your terminal and run:

npm create vite@latest

* + You’ll then be asked to name the project, select a template (choose **React**), and specify **JavaScript** or **TypeScript**.

**4. Setting Up the Project**

Once the project is created, follow these steps:

* **Navigate to your project folder**:

cd your-project-name

* **Install dependencies** (only necessary for Vite):

npm install

* **Start the development server**:
  + For **CRA**:

npm start

* + For **Vite**:

npm run dev

Once your server is running, a local development URL will be displayed in the terminal, allowing you to preview your project in a browser.

**5. Editing the Project**

You can now open your project in any code editor, like **Visual Studio Code**. From there, you can start building your React app by editing the files and structure as needed.

## Our Starting Project

Starting files can be found @ [Academind - GitHub Repository](https://github.com/academind/react-complete-guide-code/blob/zz-reactjs-summary-updated/extra-files/starting-project.zip). Click “View Raw” link to download locally. Once downloaded you must run:

npm install

Then:

npm run dev

To start the development server.

### Guide to Setting Up a React Project with Vite

**1. Starting the Project**

* The project begins by using a setup tool called **Vite**, which provides a basic structure to get started quickly with React.
* Vite cleans up unnecessary files, leaving only the essentials for development, like JavaScript and CSS files.

**2. Project Structure**

* The core of the project is found in the **source folder (src)**, where the main code lives.
* The project contains **JSX files**, which are JavaScript files that can also include HTML-like code. These files are what you will work with in React.
* Example: Instead of using separate HTML and JavaScript files, JSX allows you to write both together in a cleaner, more intuitive way.

**3. JSX Syntax and Transformation**

* **JSX** (JavaScript XML) is a syntax that allows you to write HTML within JavaScript files.
* Browsers don’t understand JSX directly, so **Vite** transforms it behind the scenes into standard JavaScript that browsers can execute.
* Example: You might write a <div>Hello World</div> inside a JavaScript file, and Vite will convert it into valid JavaScript code that renders the HTML.

**4. Working with CSS**

* The project also supports **CSS imports** directly inside JavaScript files.
* Normally, importing CSS into JavaScript wouldn't work in a browser, but the Vite setup handles this, ensuring the CSS is included in the final output.
* Example: If you have a style.css file, you can import it into your JSX file, and it will automatically be applied to the rendered elements.

**5. How the Browser Handles the Code**

* When you run the project, tools like Vite take care of transforming and bundling your code so the browser can understand it.
* The final output that the browser receives is optimized **JavaScript** and **CSS** code, without any JSX syntax or raw CSS imports.
* Example: If you inspect the website using browser developer tools, you'll see that the JSX code you wrote has been replaced by standard JavaScript, and the CSS is applied through scripts injected into the HTML.

**6. Key Concepts to Explore Next**

* Now that the basic project setup is complete, the next step is to dive deeper into the **JSX files** to understand how React components work.
* You'll start by exploring how different pieces of code come together to render simple content like "Hello World" on the screen.

## Guide to Understanding React JSX Files

**1. Main Entry File: main.jsx**

* **Purpose:** This is the main file where everything starts when the website loads. It is the first code executed, which sets up and renders the React app in the browser.
* **What Happens:**
  + Imports necessary features from the **React** and **React DOM** libraries.
  + Uses **ReactDOM.createRoot()** to target an element with the ID root in the index.html file. This is where the React app will be rendered.
  + Calls the **render()** method, passing JSX code (React’s special syntax) to be displayed inside this root element.
  + Example: If your index.html has a <div id="root"></div>, React uses main.jsx to fill this element with content.

**2. Dependencies with package.json**

* **Purpose:** The package.json file manages the project’s dependencies (third-party libraries). In this case, it includes **React** and **React DOM**.
* **What Happens:** When you install React, both libraries are installed:
  + **React** handles component logic.
  + **React DOM** allows React components to interact with the HTML DOM.
* Example: You don't need to manually download files; just list them in package.json and let Node.js handle installation.

**3. Rendering React Components**

* **ReactDOM.createRoot()** and **render()**: These functions start the process of rendering React components inside the browser.
  + **createRoot()** selects the element with the ID root.
  + **render()** places the React content inside that element.
* Example: If you pass JSX code like <App /> to render(), React will render this App component inside the <div id="root"> element.

**4. Strict Mode**

* **Purpose:** React's **StrictMode** is used to help developers by enabling additional checks and warnings for potential issues.
* **What Happens:** It doesn’t affect the functionality of your app but provides alerts for deprecated code or suboptimal practices.
* Example: It helps catch errors during development that might cause problems in future versions of React.

**5. React Components: App.jsx**

* **What is a Component?**
  + Components in React are functions that return JSX code, which gets converted to HTML by the browser.
  + **App.jsx** is an example of a component. It returns a simple **H1 element** with the text "Hello World."
* Example: A React component like App is used like an HTML element: <App />. This is how JSX makes it easy to embed reusable components in your code.

**6. Component Structure**

* **How Components Work:**
  + A component like App is a building block of a React application.
  + It returns JSX code and React takes care of rendering it on the page.
* Example: When the App component returns <h1>Hello World</h1>, React renders that HTML on the screen.

**7. Building Complex User Interfaces with Components**

* **Purpose:** React is all about breaking down user interfaces into smaller, reusable **components**.
* **What Happens:** You can build complex websites by creating components for things like a header, sidebar, or footer, and combining them to make the entire app.
* Example: If you wanted to add a navigation bar and a footer, you could create NavBar.jsx and Footer.jsx components and include them in your main app file.

**8. Next Steps: Creating Custom Components**

* **Purpose:** Components allow you to create a modular structure for your web application.
* **What's Next:** You’ll learn how to create more custom components beyond just the simple App component, to further develop your app's structure and functionality.

By understanding how these key elements like main.jsx, JSX, and components work together, you can start building and organizing React applications effectively.

# Building Dynamic and Reusable React Components

## Writing Your First Custom React Component

**1. Introduction to Components**

* **What Are Components?** In React, components are simply functions that return JSX code. You’ll be writing many components as you build up your React applications.
* **Demo Application:** We’ll start by building a Twitter-like app where users can post messages. As we build, we’ll explore core React features.

**2. Identifying Components in Your Application**

* **Break Down UI:** A React app is made up of building blocks. For example, in this Twitter-like app:
  + Each **post** (tweet) can be its own component.
  + The **list of posts** and **buttons** can also be components.
* **First Custom Component:** We'll start by creating a component for a single post.

**3. Setting Up Your Component**

* **Folder Structure:** In your project’s src folder, create a new folder called components. This folder will store all your component files.
  + **Why?** It’s a good practice to group related files together, but it’s not mandatory.
* **Create Component File:** Inside the components folder, create a file called post.jsx.
  + **Naming Convention:** Name the file based on what the component represents. Since we’re building a post, the file is named post.jsx.

**4. Creating a React Component**

* **Function Structure:** React components are just JavaScript functions. Create a function called Post:

function Post() {

return (

<div>

<p>Author: John Doe</p>

<p>React.js is awesome!</p>

</div>

);

}

* + **Capitalization Rule:** React components must start with an uppercase letter (like Post).
* **Exporting the Component:** Use the **default export** syntax to make this component usable in other files:

export default Post;

* + **Why Export?** This allows you to import and use this component in other parts of your application.

**5. Running Your Development Server**

* **Keep the Server Running:** To see your changes in real-time, ensure your development server is running:
  + If using **Vite**, run: npm run dev.  
     **Preview in Browser:** Vite will serve your website at http://localhost:5173 (or a similar port).  
    **6. Using Your Component in App.jsx**
* **Import the Component:** To use your newly created Post component, you need to import it into the App.jsx file:

import Post from './components/Post';

* **Render the Component:** Replace the existing content (e.g., “Hello World”) in App.jsx with the Post component:

function App() {

return <Post />;

}

**7. How React Renders Components**

* **JSX Syntax:** React treats your custom components like HTML elements. You use them in JSX just like <div> or <p>.
* **Behind the Scenes:** React automatically renders your component’s JSX when you include it in other components like App.

**8. Component Naming Rules**

* **Uppercase vs Lowercase:** Components you create must start with an uppercase letter when you use them in JSX. React uses this to differentiate between:
  + **Built-in HTML elements** (e.g., <div>, <p>) which start with lowercase.
  + **Custom React components** (e.g., <Post />) which start with uppercase.
* **Why This Matters:** React treats lowercase elements as HTML and uppercase as custom components. **9. Check Your Work**
* After saving everything, the browser will now display the content from your Post component:

First Post

This would be the first post! :-)

* **Next Steps:** We can now build more posts by reusing this Post component and expand it with more functionality.

This is your first step in creating custom components in React. By structuring your app using components, you can build complex and maintainable user interfaces.

# Core React Principles

## Adding Dynamic Values to React Components

Now that we’ve built our first custom component in React, let's enhance it by adding some dynamic behavior. Instead of displaying static text, we’ll make the content change every time the component is rendered.

We’ll start by modifying the Post.jsx file. Here's how to add some randomness to the displayed name:

1. **Declare an Array of Names**  
   Begin by creating an array of names, Jonathan and Zayden, which will be randomly selected each time the component is rendered:

import Post from './components/Post';

1. **Pick a Random Name**  
   Inside the component function, we'll use JavaScript’s built-in Math.random() method to choose one of the names. If the random number is greater than 0.5, we'll choose the first name; otherwise, we'll choose the second:

const chosenName = Math.random() > 0.5 ? names[0] : names[1];

1. **Render the Name Dynamically**  
   To display this name dynamically in the JSX, replace the static name in the paragraph element. In JSX, you can use curly braces {} to insert any JavaScript expression:

<p>{chosenName}</p>

Now, every time the page reloads, the displayed name alternates between Jonathan and Zayden because the component gets re-rendered. This is key in React—dynamic content based on logic or data makes your components flexible and reusable.

You can replace the logic for random values with other data sources later, like calculations or fetching from an API, making React’s ability to handle dynamic data incredibly powerful.

### Reusing Components

In React, components can be reused multiple times across your application. A good example is the **Post** component. Instead of having just one post, we may want to display multiple posts at once. This is a common scenario, especially in a real-world application where users might create several posts that need to be displayed.

Here's how you can reuse components:

1. Open your **App.jsx** file and wrap the post components inside a <main> tag.
2. Simply include the **Post** component multiple times inside the <main> tag.

function App() {

return (

<main>

<Post />

<Post />

<Post />

</main>

);

}

Now, when you load the page, you’ll see multiple instances of the post component. And since React renders each component separately, the random name feature we added earlier will work independently for each post. This is one of the powerful aspects of React — components can be reused, and each instance is handled independently by React.

### JSX Rules and Restrictions

When working with JSX in React, there are some important rules to keep in mind:

1. **Single Root Element**: In any component, your JSX must have one root element. If you try to return multiple sibling elements without wrapping them in a parent element, React will throw an error.

For example, this will cause an error:

function App() {

return (

<Post />

<Post />

);

}

To fix this, wrap the components inside a parent element like <main> or an empty tag:

function App() {

return (

<main>

<Post />

<Post />

</main>

);

}

Alternatively, you can use React fragments (<> </>) if you don’t want to add an extra HTML element:

function App() {

return (

<>

<Post />

<Post />

</>

);

}

1. **Self-Closing Tags**: In JSX, if a component or element doesn’t have any content between the opening and closing tags, you must either use a self-closing tag or explicitly include both opening and closing tags.

For example, this works:

<Post />

Or:

<Post></Post

However, this will throw an error:

<Post>

React requires you to always close your elements. This applies to both custom components (like **Post**) and built-in elements (like <img />, <input />, etc.).

**Conclusion**

By reusing components and adhering to JSX rules, you can efficiently build dynamic and scalable UIs in React. React’s flexibility with components allows for seamless repetition of elements like posts while keeping each instance independent. Furthermore, understanding JSX limitations and syntax will ensure that your components function smoothly without unnecessary errors.

### Reusing Components with Props in React

#### Why Make Components More Dynamic?

In React, reusing components multiple times in your app is a common practice. However, using the same static content in each instance makes the component less useful. Imagine if you could dynamically change the content based on different data sources, like from a database or user input.

React gives us this capability with **Props**. By using Props, you can pass different data to the same component, allowing it to be more flexible and adaptable to your app’s needs.

#### Getting Rid of Static Data

Previously, our **Post** component had random or hardcoded content. This limits the reusability of the component. To make it more dynamic, we will remove this randomness and use Props to pass values such as the author’s name and the post’s body text directly to the component.

// App.jsx

function App() {

return (

<main>

<Post author="Shakespear" body="In faith I love thee with thy eyes" />

<Post author="Charles Dickens" body="It was the best of times, it was the worse of times" />

</main>

);

}

In this example, we're using the same **Post** component twice but with different Props: one has the author "Shakespear" and the other "Charles Dickens," with unique content in the body prop. These Props will allow the **Post** component to display different information.

#### Understanding and Accessing Props

React automatically passes Props as a single object to your component function. We can access these Props inside the **Post** component and use them to render the desired content dynamically.

Here’s how we can modify the **Post** component to accept and use Props:

// Post.jsx

function Post(props) {

return (

<article>

<h2>{props.author}</h2>

<p>{props.body}</p>

</article>

);

}

In this example, we use props.author and props.body to access the values passed from the **App** component. The **Post** component is now able to render unique content based on these Props.

#### How Props Work

Props are a powerful feature that allow you to customize the behavior and content of your components. By passing different values as Props, you can reuse the same component in different places with different configurations.

React passes the props object to the component automatically. This object contains all the attributes (props) you’ve set on the component in JSX. You can then access the values in the component by referencing props.<attribute>.

For example:

<Post author="Shakespear" body="In faith I love thee with thy eyes" />

Here, the props object will have two keys, author and body, with values "Shakespear" and "In faith I love thee with thy eyes".

**Final Implementation**

Let’s bring it all together. Below is the full code for both **App.jsx** and **Post.jsx**.

**App.jsx:**

function App() {

return (

<main>

<Post author="Shakespear" body="In faith I love thee with thy eyes" />

<Post author="Charles Dickens" body="It was the best of times, it was the worse of times" />

</main>

);

}

**Post.jsx:**

function Post(props) {

return (

<article>

<h2>{props.author}</h2>

<p>{props.body}</p>

</article>

);

}

Now, when you run the app, you’ll see two distinct posts with unique authors and content, all thanks to the Props feature.

## Adding Styling to Components with CSS Modules

Now that we have our reusable component, it’s time to add some **styling** to make it look better and clearly distinguish between different instances of the same component. Without styling, it can be difficult to tell where one component ends, and another begins.

In React, there are several ways to style components:

1. **Inline Styles**: Using the style attribute directly in your JSX.
2. **Global CSS**: Using a single CSS file where classes are defined globally.
3. **CSS Modules**: A more scalable and isolated approach where styles are scoped to individual components.

Here, we'll explore **CSS Modules** as a way to prevent global CSS clashes and ensure that styles remain scoped to each component.

#### Avoiding Inline Styles

While inline styles allow you to add CSS directly within your JSX using JavaScript objects, they are not generally considered best practice because they can clutter your components and don’t offer the flexibility of CSS files.

function Post(props) {

return (

<article style={{ backgroundColor: 'lightblue', padding: '10px' }}>

<h2>{props.author}</h2>

<p>{props.body}</p>

</article>

);

}

Example of inline styles (we won’t use this approach):

Instead, we will focus on using CSS Modules for better structure and reusability.

#### Creating a CSS Module File

To scope the styles specifically to the **Post** component and avoid class name conflicts in larger applications, we will use a CSS module.

First, create a CSS file called Post.module.css:

/\* Post.module.css \*/

.post {

background-color: lightblue;

border: 1px solid #ccc;

padding: 15px;

margin-bottom: 10px;

border-radius: 5px;

}

/\* Post.module.css \*/

.author {

font-weight: bold;

color: #333;

}

.text {

font-size: 14px;

color: #555;

}

**3. Importing the CSS Module**

To use these styles, we need to import the CSS module in the **Post** component.

import classes from './Post.module.css';

function Post(props) {

return (

<article className={classes.post}>

<h2 className={classes.author}>{props.author}</h2>

<p className={classes.text}>{props.body}</p>

</article>

);

}

// Post.jsx

* **import classes from './Post.module.css';**: This imports all the class names from the Post.module.css file into a classes object.
* **className={classes.post}**: The post class from the CSS module is applied to the <article> element.

This ensures that the styles from Post.module.css are applied uniquely to this component without clashing with other styles in your project.

**4. How CSS Modules Work**

When you use **CSS Modules**, each class name in your CSS file is automatically transformed into a unique identifier behind the scenes. This ensures that there are no conflicts with class names used elsewhere in your project, even if other components use the same class names.

For example:

* In the CSS file, we define .post and .author.
* Behind the scenes, these get transformed into unique class names like Post\_post\_\_xyz and Post\_author\_\_abc, ensuring no overlap with other components using the same class names.

**5. Global CSS vs. CSS Modules**

Global CSS files apply styles across your entire project and can lead to unwanted style overrides. This is especially problematic in large applications with many components.

In contrast, **CSS Modules** scope styles to individual components. This helps maintain separation of concerns and reduces the risk of global CSS class name conflicts.

**6. Final Code Implementation**

// App.jsx

function App() {

return (

<main>

<Post author="William Shakespear" body="In faith I love thee with thy eyes." />

<Post author="Charles Dickens" body="It was the best of times, it was the worst of times." />

</main>

);

}

// Post.jsx

import classes from './Post.module.css';

function Post(props) {   
 return (   
 <article className={classes.post}>   
 <h2 className={classes.author}>{props.author}</h2> <p className={classes.text}>{props.body}</p>  
 </article> );

}

// Post.module.css:

.post {

background-color: lightblue;

border: 1px solid #ccc;

padding: 15px;

margin-bottom: 10px;

border-radius: 5px;

}

.author {

font-weight: bold;

color: #333;

}

.text {

font-size: 14px;

color: #555;

}

**7. Inspecting the Output**

If you inspect the generated HTML in the browser’s developer tools, you will notice that the class names applied to the elements are not the original post, author, and text class names but rather unique, transformed names. This transformation prevents class name clashes and helps ensure your styles remain isolated to the **Post** component.

**Key Takeaways:**

* **CSS Modules** are a powerful way to scope styles to individual components, ensuring they don’t interfere with global styles or other component styles.
* You can still use global CSS files, but CSS Modules offer better scalability, especially in larger projects.
* By using the import syntax, you can bring in styles from CSS Modules and apply them dynamically using the className attribute.

This approach provides a clean, maintainable, and scalable method for adding styles to your React components.

## Practice Lesson: Building a Posts List Component

### Summary of the Assignment (Problem):

In this exercise, you will create a new **Posts List** component in React that renders multiple **Post** components inside an unordered list. Additionally, you'll refactor the existing Post component to use an <li> (list item) tag instead of a <div>. Once the **Posts List** component is complete, you’ll use it in the App component to display a list of posts.

**Tasks**:

1. **Create a new Posts List component**:
   * This component will render an unordered list (<ul>) containing two **Post** components as list items (<li>).
   * Refactor the existing Post component to use <li> instead of <div>.
   * Move the post rendering logic from App.jsx to the new **Posts List** component.
2. **Update the App.jsx**:
   * Replace the current individual Post components in App.jsx with the newly created **Posts List** component.
3. **Optional**: Style the **Posts List** component by creating a **CSS module** for it (PostsList.module.css) and applying some basic styles.

### Solution to the Problem: Building the Posts List Component

**File: PostList.jsx**

The PostList.jsx component renders the list of posts (<ul>), with each individual post inside an li:

import Post from './Post';

import classes from '../components/PostList.module.css'; // Import the CSS file as a module.

function PostList() {

return (

<ul className={classes.posts}>

<Post author="William Shakespeare" body="In faith I love thee with thy eyes" />

<Post author="Charles Dickens" body="It was the best of times, it was the worst of times" />

</ul>

);

}

export default PostList;

In **Post.jsx**, we now move the <li> to be directly inside the PostList.jsx so that the Post component returns only the contents of a list item (<li>). Also, I fixed how the props are used:

import classes from './Post.module.css'; // Import the CSS file as a module.

function Post(props) {

return (

<div className={classes.post}>

<p className={classes.author}>{props.author}</p>

<p className={classes.text}>{props.body}</p>

</div>

);

}

export default Post;

**PostList.module.css**

This CSS file contains the grid layout for the posts list:

.posts {

list-style: none;

max-width: 50rem;

margin: 1rem auto;

padding: 1rem 0;

display: grid;

gap: 1rem;

grid-template-columns: repeat(3, 30%);

justify-content: center;

}

**Post.module.css**

This CSS file handles the styling for each individual post, including animations:

.post {

margin: 1rem 0;

padding: 1rem;

background-color: #ffffff;

border-radius: 8px;

box-shadow: 0 1px 4px rgba(0, 0, 0, 0.2);

animation: animate-in 1s ease-out forwards;

}

.author {

font-size: 0.8rem;

font-weight: bold;

color: #0b6bbf;

margin: 0;

text-transform: uppercase;

}

.text {

white-space: pre-wrap;

font-size: 1.25rem;

margin: 0.25rem 0 0 0;

color: #000;

font-style: italic;

}

@keyframes animate-in {

from {

opacity: 0;

transform: translateY(1rem);

}

to {

opacity: 1;

transform: translateY(0);

}

}

**App.jsx**

The App component simply imports and renders the PostList component:

import React from 'react';

import PostList from './components/PostList';

function App() {

return (

<main>

<PostList />

</main>

);

}

export default App;

**Key Improvements:**

**Semantic HTML:** The <li> element is now correctly placed inside PostList.jsx to ensure that list items are rendered properly within the <ul>.

**Props Usage:** The props in Post.jsx are accessed and rendered appropriately without unused code.

**CSS Styling:** The PostList.module.css applies grid styling to the list, ensuring that posts are rendered in columns. Each individual post is styled using Post.module.css.

With this setup, your component system is more modular and correctly implements the rendering of posts within an unordered list.

## Introducing State in React

As we dive deeper into React, it's time to explore one of its most powerful and essential concepts: **state**. After getting familiar with components and props, state management becomes the next crucial step. But what exactly is state, and how does it work?

Imagine you're building a component that allows users to add new posts. This component, instead of just rendering a static list of posts, should dynamically respond to user input. For example, as you type a new post or author name into a form, that information should appear in the list of posts immediately. This is where state comes into play.

### Adding the Form for a New Post

To set the stage, we'll create a new component to display the form. This form will allow users to input the post content and author name. The component renders basic HTML elements like a textarea for the post body and an input field for the author’s name. Here's how we set that up:

* The form is built using a new NewPost component, and to integrate it properly, you'll import the corresponding NewPost.jsx and NewPost.module.css files.

import classes from './NewPost.module.css';

function NewPost() {

return (

<form className={classes.form}>

<p>

<label htmlFor="body">Text</label>

<textarea id="body" required rows={3} />

</p>

<p>

<label htmlFor="name">Your name</label>

<input type="text" id="name" required />

</p>

</form>

);

}

export default NewPost;

This form component, just like any other component, renders simple HTML markup but using JSX syntax. You might notice the htmlFor attribute on labels and className for styling. These are slight deviations in JSX from standard HTML attributes to avoid clashes with JavaScript keywords.

### Displaying the Form in the App

Next, the form should be rendered alongside the existing list of posts. However, since React requires each component to have only one root element, you’ll need to use a fragment (<>...</>) to wrap the form and the post list.

This approach allows the NewPost form to coexist with the PostList component in the App component, without violating React's rules about multiple sibling root elements.

Here’s how you would add the NewPost component above the post list:

import React from 'react';

import PostList from './components/PostList';

import NewPost from './components/NewPost'; // Import the form component

function App() {

return (

<>

<NewPost /> {/\* Add the form above the list \*/}

<PostList />

</>

);

}

export default App;

### Preparing for State Changes

Now that the form is in place, the next step is to allow the user’s input to affect the displayed posts. Currently, typing into the form doesn’t do anything, but the ultimate goal is for the text entered in the form to dynamically update the posts. This is where React’s **state** feature shines.

State will enable real-time interaction. As you type into the form, the state will capture the input and update the posts below.

## Understanding State and Event Handling in React

State is a key concept in React, especially when you're building dynamic websites. Imagine a website where things change after the page has loaded, like updating a text field as users input data. This change is what we call a change in the "state" of the application.

For example, if your website displays a form where users can input a post's body text and name, the website has different states:

* One state is the initial form with empty fields.
* Another state is when the user enters text, and the form shows these inputs.

React helps you manage these different states and ensures that the user interface (UI) automatically updates when the state changes.

### Setting Up Event Listeners

To achieve this dynamic behavior, we need to listen for events like keystrokes or changes in the input fields. In traditional JavaScript, you would use querySelector to find the element and then use addEventListener to monitor changes. For example, in vanilla JavaScript, you might write:

document.querySelector('textarea').addEventListener('input', function() {

// Handle input change

});

In React, you use a more declarative approach. Instead of selecting elements and manually attaching listeners, you add an **event prop** directly to the JSX element you want to listen to.

### Event Listener in React

In React, we add an event listener to the textarea by using the onChange prop. This onChange is camel-cased (starting with lowercase followed by uppercase for subsequent words) and follows React's convention for event props.

Here’s how to set it up:

1. **Create the Function**: You create a function to handle the change event. This function will be executed whenever the input in the textarea changes.
2. **Attach the Event Listener**: You pass this function to the onChange prop of the textarea.

function NewPost() {

function changeBodyHandler(event) {

console.log(event.target.value); // Logs the current input value

}

return (

<form>

<textarea id="body" onChange={changeBodyHandler} />

</form>

);

}

In this case, the onChange prop listens to the textarea, and when the user types something, it triggers the changeBodyHandler function. Inside this function, we have access to the event object, which provides information about the event (such as which element triggered it).

**Event Object**

Whenever the event listener is triggered, an event object is automatically passed into the handler function. This object contains useful information about the event, such as which element triggered it (event.target). You can access the value of the input field by referencing event.target.value.

In our example:

function changeBodyHandler(event) {

console.log(event.target.value); // Logs the current text in the textarea

}

Every time a keystroke is made, or text is pasted into the textarea, the current value is logged to the console.

### Why Event Handling Matters for State

While handling events like changes to input fields may seem unrelated to state management, they are deeply connected. Events trigger state updates. For example, when a user types in the textarea, you might want to update the state with the current text, which will in turn update the UI.

This combination of event handling and state updates is what makes React so powerful for building interactive, dynamic applications. As the next step, we’ll explore how to connect these event listeners to state to create a fully dynamic user experience.

In short, events allow us to capture user input, and state helps us update the UI based on that input!

## Understanding React State and Event Handling

When building dynamic websites with React, managing **state** is crucial for reflecting changes in your application’s user interface (UI). In this guide, we’ll explore how state works in React, how to properly set up event listeners, and how the useState hook can be used to create reactive applications that respond to user input.

### Setting Up Event Listeners in React

To capture user input, such as text entered into a textarea, we must set up event listeners. In traditional JavaScript, you might create a variable and update it using an event listener:

let enteredBody = '';

document.querySelector('textarea').addEventListener('input', function(event) {

enteredBody = event.target.value;

});

This approach works outside React, but in React, updating variables like this won’t cause the UI to refresh, which leads us to **React’s state system**.

### Introducing React’s useState Hook

In React, changing the value of a variable directly will not automatically update the UI. The framework needs to know when to re-render components. To handle this, React provides the useState hook, a **state management feature** that allows you to register and update state values. When state changes, React re-renders the component to reflect those changes in the UI.

### How useState Works:

1. **Importing useState**:  
   First, import the useState hook from the React library:

import { useState } from 'react';

1. **Initializing State**:  
   To create a piece of state, use the useState function inside your React component:

const [enteredBody, setEnteredBody] = useState('');

* + **enteredBody:** Holds the current state value (in this case, an empty string).
  + **setEnteredBody:** A function that updates the state value and triggers a re-render of the component.

1. **Why Not Use Regular Variables?**   
   A common misconception is that you can use regular variables to store and update values. However, React only renders the UI once during the initial component render, meaning if you update a regular variable, the UI will not reflect that change.

### Using State to Handle User Input

To build a dynamic user interface where a text input updates the content of a textarea and reflects that change elsewhere on the page (e.g., in a paragraph below the input), follow these steps:

1. **Capture the Input with an Event Listener**: Attach the onChange event listener to the textarea, and use the setEnteredBody function to update the state:

<textarea id="body" onChange={(event) => setEnteredBody(event.target.value)} />

1. **Display the State in the UI**: Now, display the updated state inside a paragraph:

<p>{enteredBody}</p>

Here’s how this looks in a complete React component:

import { useState } from 'react';

function NewPost() {

const [enteredBody, setEnteredBody] = useState(''); // Initialize state

return (

<form>

<textarea

id="body"

onChange={(event) => setEnteredBody(event.target.value)}

/>

<p>{enteredBody}</p> {/\* Display updated state \*/}

</form>

);

}

**How This Works:**

* Each time the user types into the textarea, the onChange event is triggered, updating the enteredBody state using the setEnteredBody function.
* Since React re-renders the component whenever the state changes, the paragraph (<p>) below the textarea updates in real-time as the user types.

### React Re-Rendering Explained

React follows a **declarative rendering approach**. When state changes, React compares the updated component with the previous one and only re-renders the parts of the UI that have changed. This makes it more efficient than manually updating the DOM.

Here’s the process:

* When setEnteredBody is called, React stores the new state value and re-executes the NewPost function to get the updated JSX.
* React compares the old JSX and the new JSX, only updating the DOM where changes are necessary.

### Array Destructuring with useState

useState returns an array with two values:

1. The current state.
2. A function to update that state.

It’s common to use **array destructuring** to assign meaningful names to these values, like so:

const [enteredBody, setEnteredBody] = useState('');

* **enteredBody** holds the value that the user entered into the textarea.
* **setEnteredBody** is used to update the value of enteredBody.

This ensures that React is aware of any changes and re-renders the component when necessary.

**6. Summary: Using State and Event Listeners in React**

* React uses the useState hook to track changes in state and update the UI when the state changes.
* Event listeners (such as onChange) capture user input, and state update functions (like setEnteredBody) allow React to reactively update the UI.
* This system ensures that React components efficiently re-render only when necessary, offering a smoother and more optimized user experience.

By using React’s state system, you can build dynamic, interactive applications that respond to user input and update the UI in real time!

## Overview of React State and Lifting State Up

Let's break down what happens when you manage state in React and how you "lift state up" between components.

### What Happens When You Use useState in React

React's useState is a way to create and manage data (state) inside a component. When you change the state, React re-renders the component, updating the User Interface (UI) as needed.

For example, if you use useState to store the content of a text input, React will update the UI each time the text changes.

**Basic Example of useState:**

const [enteredBody, setEnteredBody] = useState('');

function handleInputChange(event) {

setEnteredBody(event.target.value); // Update state with the input value

}

Here, whenever you type something in the input field, the setEnteredBody function updates the state, which triggers React to re-render the UI.

### The Challenge: Where Should the State Go?

Initially, the state is inside the NewPost component where the user types something. However, the entered text needs to be displayed in the PostsList component, not NewPost. We need to move the state to a place where both components can access it.

If the state is in NewPost, but we need to display it in PostsList, how do we solve this? The solution is **lifting the state up**.

### What is Lifting State Up?

Lifting state up means moving the state to a **common parent component** that both child components (where you want to manipulate and use the state) can access.

In our example:

* NewPost is where the user types the input.
* PostsList is where we want to display the input as a post.

So, we move the state from NewPost into PostsList. This way, PostsList can manage the state, and NewPost can still interact with it by passing down functions (event handlers) as props.

### Steps for Lifting State Up

1. **Move the useState Hook to the Parent (PostsList)**:

const [enteredBody, setEnteredBody] = useState('');

Now, PostsList is responsible for managing the text state.

1. **Pass a Function to Handle Input Changes from PostsList to NewPost**: In PostsList, you create a function to handle the text changes:

function bodyChangeHandler(event) {

setEnteredBody(event.target.value);

}

Then, pass it as a prop to NewPost:

<NewPost onBodyChange={bodyChangeHandler} />

1. **Use the Function in NewPost**: In NewPost, you take the onBodyChange prop and use it in the input's onChange event:

<textarea onChange={props.onBodyChange}></textarea>

Now, whenever the user types in the input field in NewPost, the onBodyChange function is triggered, which updates the enteredBody state in PostsList. This state can then be displayed in PostsList.

### Multiple State Variables

You can handle more than one state in a single component by calling useState multiple times. For instance, if we want to manage both the body of the post and the author’s name, we can do this:

const [enteredBody, setEnteredBody] = useState('');

const [enteredAuthor, setEnteredAuthor] = useState('');

Then, just repeat the same process for the author input field:

1. **Pass down an onAuthorChange function from PostsList**:

function authorChangeHandler(event) {

setEnteredAuthor(event.target.value);

}

<NewPost onAuthorChange={authorChangeHandler} />

**Use it in NewPost**:

<input type="text" onChange={props.onAuthorChange} />

### Real-Time UI Updates

Whenever you type into the input fields in NewPost, the state in PostsList is updated, and React re-renders the PostsList component. This re-render causes the state to be passed down as props to the Post component, where it can be displayed.

**Full Example**

Here’s a simplified version of the entire flow:

// PostsList.js (Parent Component)

import React, { useState } from 'react';

import NewPost from './NewPost';

import Post from './Post';

function PostsList() {

const [enteredBody, setEnteredBody] = useState('');

const [enteredAuthor, setEnteredAuthor] = useState('');

function bodyChangeHandler(event) {

setEnteredBody(event.target.value);

}

function authorChangeHandler(event) {

setEnteredAuthor(event.target.value);

}

return (

<div>

<NewPost onBodyChange={bodyChangeHandler} onAuthorChange={authorChangeHandler} />

<Post body={enteredBody} author={enteredAuthor} />

</div>

);

}

export default PostsList;

// NewPost.js (Child Component where user types the input)

import React from 'react';

function NewPost(props) {

return (

<div>

<textarea onChange={props.onBodyChange}></textarea>

<input type="text" onChange={props.onAuthorChange} />

</div>

);

}

export default NewPost;

// Post.js (Where the entered text and author name are displayed)

import React from 'react';

function Post(props) {

return (

<div>

<h1>{props.author}</h1>

<p>{props.body}</p>

</div>

);

}

export default Post;

**Key Takeaways**

* **useState** is used to store data in React components.
* When state changes, React re-renders the UI where needed.
* **Lifting state up** allows you to manage state in one component and share it with others.
* You can pass event handler functions as props to update state from child components.

This pattern of lifting state up helps in building components that share and manage data effectively.

## Creating a Modal Component in React

**Goal**: Wrap content like forms (e.g., NewPost) inside a Modal component to give it an overlay (modal) look with a backdrop. This guide will help you build a reusable Modal component and explain how React's children prop allows for wrapping components.

**Steps:**

1. **Create a Modal Component**
   * Add a new file called Modal.jsx.
   * Inside Modal.jsx, define a functional component named Modal.
   * Use a fragment (<>...</>) to wrap the content inside the modal.

const Modal = ({ children }) => {

return (

<>

<div className="backdrop"></div>

<dialog className="modal" open>

{children}

</dialog>

</>

);

};

* + Create a div for the backdrop and a dialog element for the modal content.

1. **Create a CSS Module for Styling**
   * Add a Modal.module.css file.
   * Inside, define two CSS classes: one for the backdrop and one for the modal.

.backdrop {

background: rgba(0, 0, 0, 0.5);

position: fixed;

top: 0;

left: 0;

width: 100%;

height: 100%;

z-index: 10;

}

.modal {

position: fixed;

background: white;

border-radius: 8px;

padding: 1rem;

z-index: 20;

top: 50%;

left: 50%;

transform: translate(-50%, -50%);

}

1. **Use the Modal Component**
   * In the component where NewPost is used (e.g., PostsList), import and wrap NewPost inside the Modal.

import Modal from './Modal';

const PostsList = () => {

return (

<Modal>

<NewPost />

</Modal>

);

};

1. **Understanding the children Prop**
   * The children prop is a special React prop that allows you to pass elements between the opening and closing tags of a custom component.
   * In the Modal component, {children} refers to whatever is wrapped inside the Modal (in this case, the NewPost form).
   * By using {children}, React knows where to insert the wrapped content inside the Modal.

const Modal = ({ children }) => {

return (

<>

<div className="backdrop"></div>

<dialog className="modal" open>

{children}

</dialog>

</>

);

};

1. **Final Adjustments**
   * Remove unnecessary styling from the NewPost component (e.g., border-radius, margin, box-shadow) to match the modal look.
   * Add the open prop to the dialog element in the Modal component to make sure it’s visible.

<dialog className="modal" open>

**Summary:**

* You now have a reusable Modal component that can be wrapped around any content, such as a form, to give it an overlay look.
* The key concept here is the children prop, which lets you pass content into the Modal and ensures it is displayed inside the modal layout.

This setup can be reused across different components in your app, making it flexible for things like forms, warnings, or other dialogs.

## State and Conditional Content

To enhance our modal, we’ll add functionality to close it, such as clicking on the backdrop. We'll handle this with state, event listeners, and conditional rendering. Here's how:

### Track Modal Visibility with State

We need to control whether the modal is shown or hidden using a piece of state that manages its visibility. This can be done with the useState hook in React.

In the component where the modal is used (e.g., PostsList), we initialize the modal's visibility with a state variable, modaIsVisible, which starts as true (the modal is visible by default).

const [modalIsVisible, setModalIsVisible] = useState(true);

Now, we want to update this state when the backdrop is clicked. This means we’ll add an event listener to the backdrop to trigger the state change, hiding the modal.

### Set Up a Function to Hide the Modal

We create a function, hideModalHandler, that sets modalIsVisible to false. This function will be triggered when the backdrop is clicked.

const hideModalHandler = () => {

setModalIsVisible(false);

};

### Pass the Handler to the Modal

We pass hideModalHandler as a prop to the Modal component so it knows how to handle clicks on the backdrop.

<Modal onClose={hideModalHandler}>

<NewPost />

</Modal>

In the Modal component, we use the onClose prop to trigger the click event on the backdrop:

const Modal = ({ onClose, children }) => {

return (

<>

<div className="backdrop" onClick={onClose}></div>

<dialog className="modal" open>

{children}

</dialog>

</>

);

};

### Conditionally Rendering the Modal

Now we need to ensure that the modal is only rendered when modalIsVisible is true. Here are three ways to handle this:

#### Option 1: Ternary Operator

We can use a ternary operator to conditionally render the modal:

{modalIsVisible ? <Modal onClose={hideModalHandler}><NewPost /></Modal> : null}

This code checks if modaIIsVisible is true. If so, it renders the modal. If not, it renders null, meaning nothing will be shown.

#### Option 2: Logical AND Operator

Another option is using the logical AND (&&) operator. This works by rendering the modal only if modalIsVisible is true:

{modalIsVisible && <Modal onClose={hideModalHandler}><NewPost /></Modal>}

Here, the modal is rendered only when modalIsVisible is truthy. If it's false, nothing is rendered (since false && anything results in false).

#### Option 3: Storing JSX in a Variable

Alternatively, you can store the JSX for the modal in a variable and conditionally assign it based on the state:

let modalContent = null;

if (modalIsVisible) {

modalContent = <Modal onClose={hideModalHandler}><NewPost /></Modal>;

}

Later in the JSX, you can output modalContent:

{modalContent}

This approach allows for more flexibility, as you can prepare the modal content in a variable and render it elsewhere in your component.

**Recap:**

* **State** tracks the modal’s visibility.
* **Event listeners** handle user interactions (e.g., clicking the backdrop).
* **Conditional rendering** ensures the modal is only displayed when necessary. You can choose from the ternary operator, logical AND, or storing JSX in a variable based on your preferences.

All three methods are valid and will ensure that the modal is properly shown or hidden based on user interaction.

### Adding a Shared Header

At this point, we’ve made great progress with the modal by allowing it to be closed by clicking the backdrop. However, we can’t open it again yet, so let's fix that by adding a button that opens the modal.

#### Adding a Button to Open the Modal

We’ll create a header with a button that allows us to open the modal. This will be handled in a new MainHeader component, which we import into our main application file.

**Step 1: Installing react-icons**

The MainHeader component uses icons from the react-icons library. To use this, we first need to install the library:

1. Stop your development server by pressing Ctrl + C.
2. Run the following command to install react-icons:

npm install react-icons

1. Once installed, restart your server with:

npm run dev

**Step 2: Rendering the Header**

We now want to render this MainHeader above our posts list. Open your App.jsx file and import the MainHeader component:

import MainHeader from './components/MainHeader';

In the JSX return statement of App.jsx, we’ll add the MainHeader above the posts list:

return (

<>

<MainHeader />

<PostsList />

</>

);

This displays the header and the button, but the button doesn’t work yet.

**Lifting State Up to Control the Modal**

The modal’s visibility is controlled by the modalIsVisible state, which is currently inside the PostsList component. To make the button in MainHeader control the modal, we need to lift this state up to the App.jsx component.

**Step 3: Moving Modal State to App Component**

Cut the following lines from PostsList and move them to App.jsx:

const [modalIsVisible, setModalIsVisible] = useState(false);

Also, move the hideModalHandler function:

const hideModalHandler = () => {

setModalIsVisible(false);

};

Now, add a new function called showModalHandler, which will open the modal by setting modalIsVisible to true:

const showModalHandler = () => {

setModalIsVisible(true);

};

**Step 4: Passing State and Functions as Props**

To pass the modal visibility information and control functions to PostsList, we’ll use props. In App.jsx, update the PostsList component to pass down the modal state:

<PostsList isPosting={modalIsVisible} onStopPosting={hideModalHandler} />

In MainHeader, we also need to pass the showModalHandler as a prop:

<MainHeader onCreatePost={showModalHandler} />

**Step 5: Updating PostsList and MainHeader to Use the Props**

Now, in PostsList, we’ll accept the isPosting and onStopPosting props:

const PostsList = ({ isPosting, onStopPosting }) => {

return (

<>

{isPosting && <Modal onClose={onStopPosting}><NewPost /></Modal>}

{/\* other content \*/}

</>

);

};

This ensures that the modal is displayed based on the isPosting prop, and the onStopPosting prop controls closing the modal.

Next, in MainHeader.jsx, use the onCreatePost prop to open the modal when the button is clicked:

const MainHeader = ({ onCreatePost }) => {

return (

<header>

<button onClick={onCreatePost}>Create Post</button>

</header>

);

};

**Final Step: Initial State and Modal Visibility**

We initially set modalIsVisible to false so the modal isn’t displayed when the page loads:

const [modalIsVisible, setModalIsVisible] = useState(false);

Now, the button in MainHeader can open the modal, and clicking the backdrop will close it. With these updates, we've made sure that the modal can be opened and closed smoothly. We’ve also passed state and functions across multiple components to manage this behavior, which is a key aspect of React's component-based structure.

Now, we’re ready to move forward with adding new posts and managing the list of posts in the next steps.

### Adding Cancel and Submit Buttons to Form

In this guide, we will walk through how to build a new post form with a cancel and submit button. The form will collect the body of the post and the author's name. We'll also implement functionality to handle form submissions and modal closures.

**Step 1: Implementing the NewPost Component**

The NewPost component renders a form that collects a post's body and author. It includes a cancel button that closes the modal and a submit button that triggers the form submission.

import classes from './NewPost.module.css';

function NewPost({ onBodyChange, onAuthorChange, onCancel }) {

return (

<form className={classes.form}>

<p>

<label htmlFor="body">Text</label>

<textarea id="body" required rows={3} onChange={onBodyChange} />

</p>

<p>

<label htmlFor="name">Your name</label>

<input type="text" id="name" required onChange={onAuthorChange} />

</p>

<p className={classes.actions}>

<button type="button" onClick={onCancel}>Cancel</button>

<button type="submit">Submit</button>

</p>

</form>

);

}

export default NewPost;

* **onBodyChange and onAuthorChange**: These functions are triggered when the user types into the body and name fields, updating the state in the parent component.
* **Cancel Button**: The cancel button has type="button", which prevents it from submitting the form. Instead, it calls the onCancel function to close the modal.
* **Submit Button**: The submit button defaults to type="submit", so it will submit the form.

**Step 2: Setting Up the PostsList Component**

The PostsList component manages the state for the post body and author, and it renders the form inside a modal when the user is creating a new post.

import { useState } from 'react';

import Post from './Post';

import NewPost from './NewPost';

import Modal from './Modal';

import classes from './PostList.module.css';

function PostsList({ isPosting, onStopPosting }) {

const [enteredBody, setEnteredBody] = useState('');

const [enteredAuthor, setEnteredAuthor] = useState('');

function bodyChangeHandler(event) {

setEnteredBody(event.target.value);

}

function authorChangeHandler(event) {

setEnteredAuthor(event.target.value);

}

return (

<>

{isPosting && (

<Modal onClose={onStopPosting}>

<NewPost

onBodyChange={bodyChangeHandler}

onAuthorChange={authorChangeHandler}

onCancel={onStopPosting}

/>

</Modal>

)}

<ul className={classes.posts}>

<Post author={enteredAuthor} body={enteredBody} />

<Post author="Manuel" body="Check out the full course!" />

</ul>

</>

);

}

export default PostsList;

* **State Management**:
  + enteredBody and enteredAuthor store the values entered by the user in the form.
  + The bodyChangeHandler and authorChangeHandler functions update these state variables when the user types in the corresponding input fields.
* **Conditional Rendering**:
  + If isPosting is true, the modal is shown, and the NewPost component is rendered inside it.
  + The modal has an onClose function that closes it when called (passed down as onStopPosting).

**Step 3: Integrating the MainHeader Component**

The MainHeader component includes a button to trigger the modal for creating a new post.

import { MdPostAdd, MdMessage } from 'react-icons/md';

import classes from './MainHeader.module.css';

function MainHeader({ onCreatePost }) {

return (

<header className={classes.header}>

<h1 className={classes.logo}>

<MdMessage />

React Poster

</h1>

<p>

<button className={classes.button} onClick={onCreatePost}>

<MdPostAdd size={18} />

New Post

</button>

</p>

</header>

);

}

export default MainHeader;

* The onCreatePost function, when triggered, makes the modal for creating a new post visible by updating the isPosting state in the parent component.

**Step 4: Managing State and Modal Visibility in App.jsx**

In the parent component (e.g., App.jsx), we manage the visibility of the modal and handle the creation of new posts.

import { useState } from 'react';

import MainHeader from './components/MainHeader';

import PostsList from './components/PostsList';

function App() {

const [isPosting, setIsPosting] = useState(false);

function showModalHandler() {

setIsPosting(true);

}

function hideModalHandler() {

setIsPosting(false);

}

return (

<>

<MainHeader onCreatePost={showModalHandler} />

<PostsList isPosting={isPosting} onStopPosting={hideModalHandler} />

</>

);

}

export default App;

* **isPosting**: This state controls whether the modal for creating a new post is visible.
* **showModalHandler**: This function sets isPosting to true to show the modal.
* **hideModalHandler**: This function sets isPosting to false, closing the modal.

**Key Takeaways**

* **useState**: Used to manage the body and author fields in the form, as well as modal visibility.
* **Props**: Functions like onBodyChange, onAuthorChange, and onCancel are passed to child components to handle input changes and modal closure.
* **Conditional Rendering**: The modal is only rendered when isPosting is true.

By following these steps, we have implemented a dynamic form that allows users to create posts, cancel their input, and view the posts in real-time. This modular approach with reusable components is a best practice in React.

### Handling Form Submission

In this section, we'll build upon the modal form we created earlier and demonstrate how to handle form submission in React. The goal is to capture the input data, log it, close the modal, and eventually render a list of posts dynamically.

**1. Listening to the Submit Event**

In React, the form submission is handled by adding an onSubmit event handler to the form element. This will trigger when the form is submitted. Here's how we set it up:

<form className={classes.form} onSubmit={submitHandler}>

The submitHandler function will handle the event when the user submits the form. This function is defined in the NewPost component.

**2. Preventing the Default Behavior**

When a form is submitted in a traditional web environment, the browser automatically generates and sends an HTTP request. However, since React is running on the client-side and we aren't sending a request to the server, we want to prevent this default behavior. To do this, we call the preventDefault() method on the event object:

function submitHandler(event) {

event.preventDefault();

}

This stops the page from reloading or sending a request to the server.

**3. Collecting Form Data**

Next, we want to capture the values entered into the form fields. We're using React's useState to manage the input state. When the user types in the form fields, the bodyChangeHandler and authorChangeHandler functions update the state accordingly:

const [enteredBody, setEnteredBody] = useState('');

const [enteredAuthor, setEnteredAuthor] = useState('');

function bodyChangeHandler(event) {

setEnteredBody(event.target.value);

}

function authorChangeHandler(event) {

setEnteredAuthor(event.target.value);

}

In the submitHandler, we can now group these state values into a postData object:

jsx

Copy code

function submitHandler(event) {

event.preventDefault();

const postData = {

body: enteredBody,

author: enteredAuthor

};

console.log(postData);

onCancel();

}

By logging the postData object, we can verify that the form submission works as expected by viewing the data in the console.

**4. Closing the Modal on Submit**

In addition to logging the form data, we also want to close the modal when the form is submitted. To do this, we call the onCancel function that was passed as a prop:

onCancel();

This onCancel prop triggers a function that hides the modal when invoked, ensuring the modal closes after submission.

**5. Outputting the Posts Dynamically**

After handling the form submission, our next step is to render a list of posts dynamically. For now, we're just logging the postData, but eventually, we will add this data to an array of posts and render them dynamically in the PostsList component.

Here's a simplified version of the PostsList component that renders a list of posts using the Post component:

import Post from './Post';

import NewPost from './NewPost';

import Modal from './Modal';

import classes from './PostList.module.css';

function PostsList({ isPosting, onStopPosting }) {

return (

<>

{ isPosting && (

<Modal onClose={onStopPosting}>

<NewPost onCancel={onStopPosting} />

</Modal>

)}

<ul className={classes.posts}>

<Post author="John Doe" body="This is a test post" />

{/\* More posts will be added here \*/}

</ul>

</>

);

}

export default PostsList;

In future steps, we'll replace the hardcoded post data with dynamic data generated from the form submission.

**Complete Code Example**

// NewPost.js

import { useState } from 'react';

import classes from './NewPost.module.css';

function NewPost({ onCancel }) {

const [enteredBody, setEnteredBody] = useState('');

const [enteredAuthor, setEnteredAuthor] = useState('');

function bodyChangeHandler(event) {

setEnteredBody(event.target.value);

}

function authorChangeHandler(event) {

setEnteredAuthor(event.target.value);

}

function submitHandler(event) {

event.preventDefault();

const postData = {

body: enteredBody,

author: enteredAuthor

};

console.log(postData);

onCancel();

}

return (

<form className={classes.form} onSubmit={submitHandler}>

<p>

<label htmlFor="body">Text</label>

<textarea id="body" required rows={3} onChange={bodyChangeHandler} />

</p>

<p>

<label htmlFor="name">Your name</label>

<input type="text" id="name" required onChange={authorChangeHandler} />

</p>

<p className={classes.actions}>

<button type="button" onClick={onCancel}>Cancel</button>

<button>Submit</button>

</p>

</form>

);

}

export default NewPost;

This is how you can handle form submission in React while managing state, preventing default behavior, and closing the modal upon successful submission. In the next section, we will focus on dynamically rendering the submitted posts to the screen.

### Setting Up Dynamic Posts

We’re going to walk through how to add new posts dynamically to a list of posts in a React app. The goal is to gather post data from a form and update a list of posts that will render dynamically on the page.

**1. Setting Up State to Manage the Posts**

We'll use React’s useState hook to manage the list of posts. This allows the posts to be updated and rendered dynamically when a new post is added.

In the **PostsList component**, start by initializing an empty array for the posts:

const [posts, setPosts] = useState([]);

* **posts**: This array will hold all the posts.
* **setPosts**: This function updates the array when a new post is added.

**2. Creating the Function to Add Posts**

To add a new post, we need a function that will handle updating the state. This is done by adding an **addPostHandler** function, which takes the new post data and adds it to the array of existing posts.

Here's how that function looks:

function addPostHandler(postData) {

setPosts((existingPosts) => [postData, ...existingPosts]);

}

* **postData**: This contains the information for the new post.
* **Updating the state**: Instead of directly modifying the posts array, we pass a function to setPosts. This function ensures that the state update is based on the latest snapshot of existingPosts. React batches state updates, so this approach guarantees the correct data is used when adding the new post.

**Why use this approach?** React doesn't update the state immediately. It schedules updates, meaning if you try to modify the state directly, you could end up working with outdated information. By passing a function to setPosts, you ensure you're working with the latest version of the state.

**3. Connecting the Form to the Posts List**

Now, we need to connect the form where the user submits a new post to the PostsList component. The form data will be passed to the addPostHandler function when the form is submitted.

In the **PostsList component**, we pass the addPostHandler function as a prop to the form component:

<NewPost onAddPost={addPostHandler} />

In the **NewPost component**, we handle the form submission like this:

function submitHandler(event) {

event.preventDefault();

const postData = {

body: enteredBody,

author: enteredAuthor

};

onAddPost(postData); // Call the handler passed from `PostsList`

onCancel(); // Close the form after submission

}

* **postData**: This contains the text and author entered in the form.
* **Calling onAddPost**: This triggers the addPostHandler function in PostsList, adding the new post to the list.

**4. Rendering the Posts Dynamically**

Now that we can add posts, we need to render them dynamically. To do this, we loop through the posts array and display each post using the Post component.

Inside the PostsList component:

<ul className={classes.posts}>

{posts.map((post, index) => (

<Post key={index} author={post.author} body={post.body} />

))}

</ul>

* **posts.map()**: This loops through each post in the array and returns a Post component for each one.
* **key={index}**: This ensures each list item has a unique key (important for React rendering performance).
* **author and body**: These are the values we gathered from the form and are passed as props to each Post component.

**Summary**

By using useState to manage the posts, creating an addPostHandler function to update the state, and rendering the posts dynamically, you now have a dynamic list that updates whenever a new post is submitted through the form.

### Rendering Posts Dynamically

In this breakdown, you'll learn how to output posts dynamically in a React app, using the map() function to render each post, and how to handle conditional rendering when no posts are available. Additionally, we'll ensure each post has a unique key, which React requires for efficiently managing lists.

**1. Rendering an Array of JSX Elements**

When you have an array of data (like posts), you want to output them as a list of components. This can be achieved using the map() function, which transforms each item in the array into a JSX element.

In the **PostsList component**, we have a posts state variable that stores an array of post objects. The goal is to loop through this array and render a **Post component** for each post.

<ul className={classes.posts}>

{posts.map((post, index) => (

<Post key={index} author={post.author} body={post.body} />

))}

</ul>

* **posts.map()**: This method transforms each post object in the array into a JSX element (a Post component in this case).
* **JSX inside map()**: Each post object is passed to the Post component, which takes in the author and body as props.
* **key={index}**: React requires a unique key for each item in the list to help it identify which items have changed, added, or removed. In this case, we're using the index from the map() function.

**Why is the key prop important?** React uses keys to optimize rendering performance when updating lists. If a key is missing or not unique, React may incorrectly identify which items have changed, leading to bugs or inefficiencies.

**2. Conditional Rendering for Empty Lists**

When there are no posts, it’s important to display a message informing the user. For this, we’ll use **conditional rendering** to display different content depending on whether there are posts.

In the **PostsList component**, check the length of the posts array to determine whether to display the list or a fallback message:

{posts.length > 0 && (

<ul className={classes.posts}>

{posts.map((post, index) => (

<Post key={index} author={post.author} body={post.body} />

))}

</ul>

)}

* **posts.length > 0**: If the array has at least one post, render the list of posts.

If there are no posts, render a message instead:

{posts.length === 0 && (

<div style={{ textAlign: 'center', color: 'white', fontSize: '1.5rem' }}>

<h2>No posts yet.</h2>

<p>Start adding some!</p>

</div>

)}

* **Fallback message**: This will be displayed when the posts array is empty, encouraging the user to add a post.

**3. Adding Inline Styles**

For the fallback message, we're using **inline styles** to center the text and change its color. Here’s a brief breakdown:

<div style={{ textAlign: 'center', color: 'white', fontSize: '1.5rem' }}>

* **textAlign: 'center'**: Centers the text horizontally.
* **color: 'white'**: Sets the text color to white.
* **fontSize: '1.5rem'**: Adjusts the font size.

Remember to use **camelCase** when defining inline CSS styles in JSX, as JSX is JavaScript. For example, use textAlign instead of text-align.

**4. Final Workflow: Adding and Rendering Posts**

When a new post is submitted via the form in the **NewPost component**, the addPostHandler in **PostsList** updates the posts array. Once the array is updated, React automatically re-renders the list of posts, including the new post. If there are no posts initially, the fallback message is shown until the first post is added.

**Summary**

With map() and conditional rendering, you can dynamically output a list of posts while ensuring each item has a unique key. The fallback message improves user experience by guiding them to take action when the list is empty.

## Persisting Data in a React App with a Backend

**Problem:**

When building a React app (a front-end JavaScript library used to build interactive user interfaces), the data stored (like posts) only exists in memory. If the page is reloaded, the data is lost, as React runs entirely in the browser.

**Solution:**

To prevent data loss and allow persistence across sessions or users, you need to store the data on a backend. The backend is a separate service (like an API) that runs on a server and interacts with a database or files. Your React app communicates with the backend using HTTP requests (e.g., GET, POST).

**Example Backend with Node.js and Express:**

A sample backend is provided using Node.js and Express, where posts are stored in a posts.json file. This backend provides REST API endpoints, allowing the React app to create, fetch, and store posts persistently.

1. **GET /posts**: Fetches all stored posts.
2. **GET /posts/:id**: Fetches a single post by its ID.
3. **POST /posts**: Creates a new post and stores it.

You can run this backend locally with:

* npm install (to install dependencies)
* npm start (to start the backend server)

This allows the React front-end to interact with this API by sending requests.

**Alternative Backends: PHP and Python**

You can also create a similar backend API using PHP or Python, which will provide the same functionality for storing and fetching posts.

**PHP Backend (Using Slim Framework)**

1. **GET /posts**:

PHP

$app->get('/posts', function ($request, $response) {

$data = json\_decode(file\_get\_contents('posts.json'), true);

return $response->withJson($data['posts']);

});

1. **POST /posts**:

PHP

php

Copy code

$app->post('/posts', function ($request, $response) {

$postData = $request->getParsedBody();

$data = json\_decode(file\_get\_contents('posts.json'), true);

$postData['id'] = uniqid();

array\_unshift($data['posts'], $postData);

file\_put\_contents('posts.json', json\_encode($data));

return $response->withJson(['message' => 'Stored new post.', 'post' => $postData], 201);

});

**Python Backend (Using Flask)**

1. **GET /posts**:

python

from flask import Flask, jsonify

import json

app = Flask(\_\_name\_\_)

@app.route('/posts', methods=['GET'])

def get\_posts():

with open('posts.json') as f:

data = json.load(f)

return jsonify(data['posts'])

if \_\_name\_\_ == '\_\_main\_\_':

app.run(port=8080)

1. **POST /posts**:

from flask import Flask, request, jsonify

import json

app = Flask(\_\_name\_\_)

@app.route('/posts', methods=['POST'])

def create\_post():

post\_data = request.json

post\_data['id'] = str(uuid.uuid4())

with open('posts.json', 'r+') as f:

data = json.load(f)

data['posts'].insert(0, post\_data)

f.seek(0)

json.dump(data, f)

return jsonify({'message': 'Stored new post.', 'post': post\_data}), 201

if \_\_name\_\_ == '\_\_main\_\_':

app.run(port=8080)

**Key Takeaways:**

* **Frontend (React)**: Handles UI interactions, displays posts, and submits form data to the backend.
* **Backend**: Can be written in any server-side language like Node.js, PHP, or Python. It stores data (posts) in a file or database and responds to requests from the frontend.
* **API Communication**: The React app makes HTTP requests to the backend (GET to fetch posts, POST to create a new post).

Let me know if you'd like further adjustments to any of the code examples!

### Building a Full-Stack Post Application with React and Node.js

In this section, you'll integrate a React frontend with a Node.js/Express backend to create, store, and retrieve posts. We’ll start by configuring the backend and then connect it with the React frontend to allow users to add posts, which will be persisted on the backend.

**1. Backend Setup: Node.js and Express**

The backend API is responsible for handling HTTP requests for creating and retrieving posts. Here’s a simplified breakdown of the backend code:

* **Express.js** is used to handle requests.
* **Posts are stored in a JSON file** (posts.json) to simulate database storage.
* **CORS headers** are configured to allow communication between the frontend and backend when running on different domains or ports.

Here’s the code for the backend (app.js):

**const express = require('express');**

**const bodyParser = require('body-parser');**

**const { getStoredPosts, storePosts } = require('./data/posts');**

**const app = express();**

**app.use(bodyParser.json());**

**// Set CORS headers**

**app.use((req, res, next) => {**

**res.setHeader('Access-Control-Allow-Origin', '\*');**

**res.setHeader('Access-Control-Allow-Methods', 'GET,POST');**

**res.setHeader('Access-Control-Allow-Headers', 'Content-Type');**

**next();**

**});**

**app.get('/posts', async (req, res) => {**

**const storedPosts = await getStoredPosts();**

**res.json({ posts: storedPosts });**

**});**

**app.post('/posts', async (req, res) => {**

**const existingPosts = await getStoredPosts();**

**const postData = req.body;**

**const newPost = { ...postData, id: Math.random().toString() };**

**const updatedPosts = [newPost, ...existingPosts];**

**await storePosts(updatedPosts);**

**res.status(201).json({ message: 'Stored new post.', post: newPost });**

**});**

**app.listen(8080);**

The data/posts.js file is responsible for reading and writing posts to the posts.json file:

**const fs = require('node:fs/promises');**

**async function getStoredPosts() {**

**const rawFileContent = await fs.readFile('posts.json', { encoding: 'utf-8' });**

**const data = JSON.parse(rawFileContent);**

**return data.posts ?? [];**

**}**

**function storePosts(posts) {**

**return fs.writeFile('posts.json', JSON.stringify({ posts: posts || [] }));**

**}**

**exports.getStoredPosts = getStoredPosts;**

**exports.storePosts = storePosts;**

To run the backend:

* Run npm install to install the dependencies.
* Start the server using npm start.

The backend will now listen on http://localhost:8080 and handle requests for creating and retrieving posts.

**2. Frontend Setup: React Application**

The React frontend connects to this backend to add and fetch posts. Let’s walk through the key parts of the frontend code.

**State Management and Adding Posts**

The PostsList component manages the state for the posts and handles interactions such as adding new posts and displaying them. Here’s the breakdown:

* The addPostHandler function is responsible for sending a **POST request** to the backend whenever a new post is submitted.
* The posts are stored locally in the component state using React’s useState hook.
* When a new post is added, the local state is updated and the post is sent to the backend simultaneously.

Here’s the code for PostsList.js:

**import { useState } from 'react';**

**import Post from './Post';**

**import NewPost from './NewPost';**

**import Modal from './Modal';**

**import classes from './PostList.module.css';**

**function PostsList({ isPosting, onStopPosting }) {**

**const [posts, setPosts] = useState([]);**

**function addPostHandler(postData) {**

**fetch('http://localhost:8080/posts', {**

**method: 'POST',**

**body: JSON.stringify(postData),**

**headers: {**

**'Content-Type': 'application/json',**

**},**

**});**

**// Update the state with the new post**

**setPosts((existingPosts) => [postData, ...existingPosts]);**

**}**

**return (**

**<>**

**{isPosting && (**

**<Modal onClose={onStopPosting}>**

**<NewPost onCancel={onStopPosting} onAddPost={addPostHandler} />**

**</Modal>**

**)}**

**{posts.length > 0 && (**

**<ul className={classes.posts}>**

**{posts.map((post, index) => (**

**<Post key={index} author={post.author} body={post.body} />**

**))}**

**</ul>**

**)}**

**{posts.length === 0 && (**

**<div style={{ textAlign: 'center', color: 'white', fontSize: '1.5rem' }}>**

**<h2>No posts yet.</h2>**

**<p>Start adding some!</p>**

**</div>**

**)}**

**</>**

**);**

**}**

**export default PostsList;**

**Key Points in the Code:**

1. **Fetch API for Sending Data**  
   The fetch function is used to send a **POST request** to the backend at http://localhost:8080/posts. The post data is converted to JSON using JSON.stringify(), and the request includes a Content-Type header to specify that we're sending JSON.

fetch('http://localhost:8080/posts', {

method: 'POST',

body: JSON.stringify(postData),

headers: {

'Content-Type': 'application/json',

},

});

1. **State Update Best Practice**  
   When updating the state based on the previous state, always use the function form of setPosts to ensure you’re working with the most up-to-date state.

setPosts((existingPosts) => [postData, ...existingPosts]);

1. **Rendering Posts**  
   The component conditionally renders posts if they exist, or shows a message when there are no posts.

**How It Works Together**

1. When a user submits a new post, it is sent to the backend, which stores the post in the posts.json file.
2. Simultaneously, the post is added to the local state in React, updating the UI.
3. When the page is refreshed or reopened, the posts will still exist in the backend and can be fetched (explained in the next section).

### Fetching Posts on Initial Page Load in React

When building a React app that interacts with a backend, fetching data from the server is a common requirement. For our post application, we need to fetch posts when the page is first visited or reloaded. Here's how we can do this efficiently while avoiding common pitfalls like infinite loops.

**1. The Backend Setup**

In the previous section, we set up a backend that provides a list of posts via a **GET request** to /posts. The server responds with an array of posts stored in a JSON file. Now, we want to retrieve these posts when the page first loads and display them on the frontend.

**2. The Problem with Fetching Inside the Component**

At first glance, it may seem straightforward to place a fetch request inside the PostsList component. However, doing so directly within the component function causes an issue. React components are re-rendered whenever their state changes. If we fetch posts and update the state, it will cause the component to re-render, which in turn triggers another fetch, leading to an **infinite loop**. To prevent this, we need to avoid placing the fetch directly inside the component function.

**3. Introducing the useEffect Hook**

To handle side effects like fetching data from a backend in React, we use the useEffect hook. Side effects are any operations that don't directly involve rendering UI (e.g., fetching data, subscriptions). The useEffect hook allows us to control when our code runs to prevent infinite loops and unnecessary fetch requests.

### Basic useEffect Structure

The useEffect hook takes two arguments:

* **A function**: This function contains the code to execute (e.g., the fetch request).
* **A dependency array**: This array tells React when to re-run the effect. An empty array ensures that the effect only runs once, when the component is first mounted.

Let’s break down the steps to fetch posts when the page loads using useEffect.

**4. Fetching Posts with useEffect**

Here’s the updated PostsList component that fetches posts when the page is first loaded:

import { useState, useEffect } from 'react';

import Post from './Post';

import NewPost from './NewPost';

import Modal from './Modal';

import classes from './PostList.module.css';

function PostsList({ isPosting, onStopPosting }) {

const [posts, setPosts] = useState([]);

// Fetch posts from the backend when the component is first loaded

useEffect(() => {

async function fetchPosts() {

const response = await fetch('http://localhost:8080/posts');

const responseData = await response.json();

setPosts(responseData.posts); // Update state with fetched posts

}

fetchPosts(); // Call the function when the component first renders

}, []); // Empty array ensures the effect runs only once

function addPostHandler(postData) {

fetch('http://localhost:8080/posts', {

method: 'POST',

body: JSON.stringify(postData),

headers: {

'Content-Type': 'application/json',

},

});

setPosts((existingPosts) => [postData, ...existingPosts]);

}

return (

<>

{isPosting && (

<Modal onClose={onStopPosting}>

<NewPost onCancel={onStopPosting} onAddPost={addPostHandler} />

</Modal>

)}

{posts.length > 0 && (

<ul className={classes.posts}>

{posts.map((post, index) => (

<Post key={index} author={post.author} body={post.body} />

))}

</ul>

)}

{posts.length === 0 && (

<div style={{ textAlign: 'center', color: 'white', fontSize: '1.5rem' }}>

<h2>No posts yet.</h2>

<p>Start adding some!</p>

</div>

)}

</>

);

}

export default PostsList;

**5. How useEffect Works**

* **The function inside useEffect**: In this example, the fetchPosts function is defined inside useEffect. It is responsible for sending the **GET request** to fetch the posts from the backend.
* **Why async/await inside useEffect**: We want to use async/await to handle the asynchronous nature of the fetch request. However, we cannot make the useEffect callback itself async because React expects it to return either nothing or a cleanup function. To work around this, we define an **inner function** (fetchPosts) that is async and call it inside the effect.
* **Dependency array**: The empty array ([]) ensures that the effect only runs **once**, when the component is first mounted. This prevents the infinite loop problem caused by re-fetching data on every re-render.

**6. Explanation of the Code**

* **Fetching posts**: When the page first loads, useEffect calls the fetchPosts function, which sends a **GET request** to the backend at http://localhost:8080/posts. The response is then converted into JSON and the posts are extracted from responseData.posts.
* **Updating state**: Once the posts are fetched, the setPosts function is called to update the component state with the fetched posts. This triggers a re-render, and the posts are displayed.
* **Rendering posts**: If there are posts in the state, the component renders them in a list. If no posts are present, a message prompts the user to start adding posts.

**Summary**

1. **Avoiding Infinite Loops**: Placing the fetch request inside useEffect with an empty dependency array ensures that the data is fetched once when the component is first rendered, avoiding an infinite loop.
2. **Handling Async Requests**: The async/await pattern is used to handle asynchronous requests, but the effect function itself isn’t made async. Instead, an inner function is created and executed.
3. **Rendering Posts**: Posts are fetched from the backend, stored in the component state, and rendered dynamically in the UI.

This approach ensures that the posts are fetched and displayed correctly without causing performance issues or unnecessary re-renders.

Enhancing User Experience with Loading Indicators

When fetching data from a backend, like loading posts for a blog, the process isn't always instant, especially when the backend is slower. Users may experience delays, and without visual feedback, it may seem like the application isn’t working. This is where adding a **loading state** and **error handling** can improve the user experience.

Here’s how you can implement it in React:

**Step 1: Setting Up useState for Loading**

Start by introducing a new state that tracks whether data is being fetched. In the PostList component, use React’s useState hook to manage this state.

const [isFetching, setIsFetching] = useState(false);

Initially, isFetching is false because we aren't loading data yet. When we start fetching the posts, we’ll update this state.

**Step 2: Updating State While Fetching**

Next, inside your useEffect hook where you are fetching the posts, set isFetching to true at the beginning of the fetch request. Once the data is successfully retrieved, set isFetching to false to indicate loading is complete.

useEffect(() => {

const fetchPosts = async () => {

setIsFetching(true); // Start loading

const response = await fetch('/posts');

const data = await response.json();

setPosts(data.posts);

setIsFetching(false); // End loading

};

fetchPosts();

}, []);

**Step 3: Displaying a Loading Indicator**

While the posts are being fetched, we’ll show a loading message. In the JSX, check if isFetching is true, and if so, display a loading spinner or message. Otherwise, display the posts.

return (

<div>

{isFetching ? (

<p>Loading posts...</p>

) : posts.length > 0 ? (

<PostList posts={posts} />

) : (

<p>No posts available</p>

)}

</div>

);

This ensures users see "Loading posts..." while waiting for the backend response. If there are no posts, it shows "No posts available."

**Step 4: Simulating a Slow Backend**

For testing purposes, you can simulate a slow backend by adding a delay in the backend’s response. In your backend code, you could add a simple delay using setTimeout.

setTimeout(() => {

res.json({ posts });

}, 2000); // Simulates a 2-second delay

This delay mimics a real-world scenario where the backend might take time to respond.

**Final Thoughts**

By adding a **loading state** and an optional **error handling** mechanism, you make your application feel more responsive and user-friendly. This approach ensures that users understand the app is working behind the scenes and waiting for data, rather than thinking something is broken or not functioning.

## Introduction to Routing in React

Many React apps, especially those that are more than just simple demos, need routing. Routing allows different paths in the URL to load different components or pages, even though React apps are single-page applications (SPAs).

Currently, our app only has one path, and the URL doesn’t change regardless of what we click. This means we can’t link users directly to specific parts of the app, such as a "New Post" mode. We want to support multiple paths and load different components for different URLs, like /new-post or /posts.

### What is Routing?

Routing allows us to listen to URL changes while the app is running or evaluate the URL when the app is first loaded, and then load different components based on that URL. For example, if the URL is /products, we could load a products page, and if the URL is /new-post, we could load the new post form.

Even though React apps are SPAs and technically have only one HTML page, routing lets us simulate multiple pages by loading different components for different paths.

## React Router

React doesn't come with built-in routing capabilities, but the **React Router** package is the most popular solution for adding routing to React apps. It handles URL changes and enables loading different components for different routes.

While you could write your own solution to handle URLs, React Router offers a powerful and feature-rich way to manage routes. We’ll be using **React Router DOM**, which is designed for browser-based routing.

**Installing React Router DOM**

To get started, you need to install **React Router DOM** in your React app. This package gives us routing capabilities for the frontend.

1. First, quit your development server (if it's running).
2. Run this command in your terminal:

npm install react-router-dom

This will install the React Router package in your app.

### Setting Up React Router

Now that React Router is installed, we can start adding routing to our app. Keep in mind that this routing happens entirely on the client side, so there's no need to involve a backend server.

### Introduction to Routing

Now that we have data fetching and sending set up in our demo website, we can further enhance the user experience by introducing **routing**. Routing allows us to have multiple URLs or "paths" in our React app, loading different components based on the URL. This way, users can directly link to specific pages, such as creating a new post or viewing individual post details.

**Adding Routes to the Demo Website**

In this demo, we want to support three main routes:

1. The **Home Page** – where all posts are listed.
2. The **New Post Page** – for creating new posts, likely displayed as a modal.
3. The **Post Detail Page** – to view details of a specific post when clicked.

### Setting Up React Router

To add routing, we’ll use the **React Router DOM** package. After installing it with npm install react-router-dom, we can configure the routing in our app. Here's the breakdown of what needs to be done:

1. **Import Router Components**: We need to import RouterProvider and createBrowserRouter from react-router-dom to enable routing.
2. **Define Routes**: The routes are configured in an array passed to createBrowserRouter(). Each route is an object with a path (the URL path) and an element (the component to load when that path is accessed).
3. **Render the Router Provider**: Instead of rendering the App component directly, we wrap our app with RouterProvider and pass the router configuration as a prop.

**Code Implementation**

Here's the code to set up basic routing in your app:

import React from 'react'

import ReactDOM from 'react-dom/client'

import { RouterProvider, createBrowserRouter } from 'react-router-dom'

import App from './App'

import './index.css'

import NewPost from './components/NewPost'

const router = createBrowserRouter([

{ path: '/', element: <App /> }, // Home page route

{ path: '/create-post', element: <NewPost /> } // New post route

]);

ReactDOM.createRoot(document.getElementById('root')).render(

<React.StrictMode>

<RouterProvider router={router} /> // Setting up the router provider

</React.StrictMode>

)

**Explanation:**

1. **Home Route ('/')**: This is the default route that loads the App component, where all posts are displayed.
2. **New Post Route ('/create-post')**: This route loads the NewPost component. Currently, it replaces the entire content of the page with the new post form.

**Behavior**

* When you access the home path ('/'), it renders the App component, showing all posts.
* When you access the new post path ('/create-post'), it renders the NewPost component.

This basic setup allows for navigation between multiple pages without reloading the entire app. The next step would be to refine this by keeping the rest of the app (like the post list) visible while displaying the new post form as a modal, using layout routes

### Introducing Layout Routes

In more complex React apps with routing, it's common to have shared layout elements across different pages. For instance, a navigation bar or header that remains consistent throughout the app. To implement this, **layout routes** are used in React Router. Layout routes allow you to define common elements, such as a header, and "nest" other routes within them, ensuring the layout is shared across different pages.

**Key Concepts**

1. **Layout Route**: A route that wraps other routes and defines common layout elements like headers, sidebars, or footers.
2. **Children Routes**: Nested routes that inherit the layout and are rendered inside the layout route's content.
3. **Outlet Component**: A placeholder component provided by React Router where the content of the nested routes is injected.

### Steps to Create a Layout Route

1. **Create the Layout Component**: In this case, we create a RootLayout component that includes the shared MainHeader and an Outlet where the nested route content will appear.
2. **Update the Route Configuration**: Modify the route configuration to include the RootLayout component as a layout route, and nest the other routes inside it using the children key.
3. **Refactor Code**: Ensure the MainHeader is only in the layout route, so you don't end up with duplicate headers in your app.

**Code Explanation**

* **RootLayout.jsx**: This component contains the layout that will be shared across multiple routes (in this case, the MainHeader). The Outlet component serves as a placeholder where the nested route components (like App or NewPost) will be rendered.

import { Outlet } from 'react-router-dom';

import MainHeader from '../components/MainHeader';

function RootLayout() {

return (

<>

<MainHeader />

<Outlet /> {/\* Placeholder for nested routes \*/}

</>

);

}

export default RootLayout;

* **main.jsx**: Here, we configure the router to use RootLayout as the layout route and nest the individual routes ('/' and '/create-post') inside it. This ensures that both routes share the same layout.

import React from 'react'

import ReactDOM from 'react-dom/client'

import { RouterProvider, createBrowserRouter } from 'react-router-dom'

import App from './App'

import './index.css'

import NewPost from './components/NewPost'

import RootLayout from './routes/RootLayout'

const router = createBrowserRouter([

{

path: '/', element: <RootLayout />, // Layout route with MainHeader

children: [

{ path: '/', element: <App /> }, // Home route

{ path: '/create-post', element: <NewPost /> } // New post route

],

},

]);

ReactDOM.createRoot(document.getElementById('root')).render(

<React.StrictMode>

<RouterProvider router={router} />

</React.StrictMode>

);

**Key Changes**

1. **RootLayout**: This is a layout route that wraps the MainHeader and places the route-specific content via the Outlet component.
2. **Nested Routes**: Inside the children array, the home page ('/') and the new post page ('/create-post') are nested under the layout route. This ensures the layout is shared while rendering different components based on the URL.
3. **Refactoring**: The MainHeader is now placed only in RootLayout, so it's not duplicated in the App component, ensuring the header appears only once across the app.

**Behavior**

* When you visit the home path ('/'), the RootLayout component renders the MainHeader, and the App component is loaded inside the Outlet.
* When you navigate to the new post path ('/create-post'), the RootLayout still renders the MainHeader, but the NewPost component is loaded inside the Outlet.

## Refactoring the Posts and NewPost Components

In this refactoring, we are making the following changes:

1. **Organizing Components**: Move route components (App and NewPost) into the routes folder for clarity and maintainability. Non-route components will remain in the components folder.
2. **Renaming App to Posts**: The former App component is now responsible for rendering the list of posts, so we rename it to Posts.
3. **Removing Duplicated Logic**: We refactor both the Posts and NewPost components to clean up unnecessary or duplicated logic, like handling modals in both places.
4. **Using Outlet**: We make Posts a layout route by adding an Outlet to render the NewPost component as a child route. This allows NewPost to be displayed as an overlay above the PostList component.
5. **Implementing the Modal Overlay**: We ensure that the NewPost form is displayed as a modal overlay when navigating to /create-post.

**Updated Code**

**Posts.jsx (previously App.jsx):**

The Posts component now functions as a layout route, displaying the list of posts and allowing NewPost to render as a child route using the Outlet.

import { Outlet } from 'react-router-dom';

import { useState } from 'react';

import PostList from '../components/PostList';

function Posts() {

return (

<>

<Outlet /> {/\* This Outlet will render the NewPost modal as a child route \*/}

<main>

<PostList />

</main>

</>

);

}

export default Posts;

**NewPost.jsx:**

This component remains responsible for rendering the form to add a new post. It now uses the Modal component to wrap the form.

import { useState } from 'react';

import classes from './NewPost.module.css';

import Modal from '../components/Modal';

function NewPost({ onCancel, onAddPost }) {

const [enteredBody, setEnteredBody] = useState('');

const [enteredAuthor, setEnteredAuthor] = useState('');

function bodyChangeHandler(event) {

setEnteredBody(event.target.value);

}

function authorChangeHandler(event) {

setEnteredAuthor(event.target.value);

}

function submitHandler(event) {

event.preventDefault();

const postData = {

body: enteredBody,

author: enteredAuthor

};

onAddPost(postData);

onCancel(); // Close the modal after submission

}

return (

<Modal>

<form className={classes.form} onSubmit={submitHandler}>

<p>

<label htmlFor="body">Text</label>

<textarea id="body" required rows={3} onChange={bodyChangeHandler} />

</p>

<p>

<label htmlFor="name">Your name</label>

<input type="text" id="name" required onChange={authorChangeHandler} />

</p>

<p className={classes.actions}>

<button type="button" onClick={onCancel}>Cancel</button>

<button>Submit</button>

</p>

</form>

</Modal>

);

}

export default NewPost;

**main.jsx:**

We update the route configuration to nest the NewPost component inside the Posts route using the children key. This ensures the modal is rendered as an overlay when visiting /create-post.

import React from 'react';

import ReactDOM from 'react-dom/client';

import { RouterProvider, createBrowserRouter } from 'react-router-dom';

import Posts from './routes/Posts';

import './index.css';

import NewPost from './routes/NewPost';

import RootLayout from './routes/RootLayout';

const router = createBrowserRouter([

{

path: '/',

element: <RootLayout />,

children: [

{

path: '/',

element: <Posts />,

children: [{ path: '/create-post', element: <NewPost /> }],

},

],

},

]);

ReactDOM.createRoot(document.getElementById('root')).render(

<React.StrictMode>

<RouterProvider router={router} />

</React.StrictMode>

);

**Explanation of Changes**

1. **Posts Component as a Layout Route**: The Posts component now functions as a layout route by using Outlet to render the nested NewPost component as a modal overlay. This is the key to ensuring the new post form is shown over the list of posts without replacing it.
2. **Route Configuration**: The NewPost component is nested under the Posts route, ensuring that it appears as a child of the post list and as a modal overlay when navigating to /create-post.
3. **Modal Behavior**: The modal is now fully managed within the NewPost component, making it responsible for rendering the form inside a modal overlay. When submitting the form or clicking "Cancel," the modal closes.

**Next Steps**

* **Handling Close Functionality**: The next part of the refactor would involve wiring up the New Post button and modal close functionality to ensure that the modal can be opened and closed interactively without manually typing the URL.

### Linking the "New Post" Button to a Modal

In this section, we’ll walk through how to create a "New Post" button that opens a modal for adding new content, using Next.js and React Router's navigation features. The goal is to avoid a full page reload, maintaining the SPA behavior.

**1. Replacing the Button with a Link Component**

First, we'll update the button in the MainHeader component that opens the modal. The Link component from react-router-dom will allow us to navigate between different routes without refreshing the page, which is essential for SPAs.

// MainHeader.js

import { Link } from 'react-router-dom';

import { MdPostAdd, MdMessage } from 'react-icons/md';

import classes from './MainHeader.module.css';

function MainHeader() {

return (

<header className={classes.header}>

<h1 className={classes.logo}>

<MdMessage />

React Poster

</h1>

<p>

<Link to="/create-post" className={classes.button}>

<MdPostAdd size={18} />

New Post

</Link>

</p>

</header>

);

}

export default MainHeader;

This replaces the button with a Link component, which ensures that clicking the "New Post" button will navigate to the /create-post route.

**2. Avoid Full Page Reloads**

Using a regular anchor tag (<a>) would cause the page to reload, which defeats the purpose of an SPA. By using the Link component, React Router ensures that only the relevant component (in this case, the modal) is rendered, without reloading the entire app.

To implement this, we'll import the Link component and replace the button element with it, setting the to property to the route where we want the modal to open (i.e., /create-post).

<Link to="/create-post" className={classes.button}>

<MdPostAdd size={18} />

New Post

</Link>

**3. Creating a Modal Component**

The modal component will wrap the content in a dialog tag and display it when the user navigates to /create-post. We also need a backdrop to close the modal when clicking outside of it. Additionally, we’ll use React Router’s useNavigate hook for programmatic navigation when clicking on the backdrop.

// Modal.js

import { useNavigate } from 'react-router-dom';

import classes from './Modal.module.css';

function Modal({ children }) {

const navigate = useNavigate();

function closeHandler() {

navigate('..'); // Go back one route

}

return (

<>

<div className={classes.backdrop} onClick={closeHandler} />

<dialog open className={classes.modal}>

{children}

</dialog>

</>

);

}

export default Modal;

Here, we set up the modal with a backdrop div and a dialog for the content. The useNavigate hook allows the user to programmatically close the modal by navigating back to the previous page.

**4. Handling Form Submission**

In the NewPost component, we add a form where users can submit their post content. When the form is submitted, we handle the data and navigate back to the main page. We use Link again for canceling the action, keeping the navigation within the SPA.

// NewPost.js

import { useState } from 'react';

import { Link } from 'react-router-dom';

import classes from './NewPost.module.css';

import Modal from '../components/Modal';

function NewPost({ onAddPost }) {

const [enteredBody, setEnteredBody] = useState('');

const [enteredAuthor, setEnteredAuthor] = useState('');

function bodyChangeHandler(event) {

setEnteredBody(event.target.value);

}

function authorChangeHandler(event) {

setEnteredAuthor(event.target.value);

}

function submitHandler(event) {

event.preventDefault();

const postData = {

body: enteredBody,

author: enteredAuthor,

};

onAddPost(postData);

}

return (

<Modal>

<form className={classes.form} onSubmit={submitHandler}>

<p>

<label htmlFor="body">Text</label>

<textarea id="body" required rows={3} onChange={bodyChangeHandler} />

</p>

<p>

<label htmlFor="name">Your name</label>

<input type="text" id="name" required onChange={authorChangeHandler} />

</p>

<p className={classes.actions}>

<Link to=".." type="button">Cancel</Link>

<button>Submit</button>

</p>

</form>

</Modal>

);

}

export default NewPost;

Here, we handle form submission inside the modal. On clicking "Cancel," the modal will close, navigating back to the previous page.

**5. Improving the User Experience**

By using React Router’s Link and useNavigate, we prevent unnecessary full-page reloads and ensure a smooth navigation experience in the SPA. The modal stays in sync with the app’s routing system, allowing for easy sharing of URLs and a more fluid user experience.

This approach allows you to maintain the core functionality of the app while optimizing performance and user experience through SPA routing with React and Next.js.

## Enhancing Data Fetching with loader()’s

In this process, we’ll explore how to improve data fetching and routing in a React application using React Router’s loader functionality. The goal is to reduce component complexity by moving data fetching logic into the route definitions and utilizing the React Router’s built-in features for handling data before rendering.

**Step 1: Setting Up the Router with Routes**

We’ll start by defining routes for our application using createBrowserRouter from react-router-dom. These routes will define where each component is rendered in our app.

**In index.js or main.js file**:

* Import createBrowserRouter and RouterProvider from React Router.
* Define the routes and assign a loader to handle data fetching for the Posts route.

import Posts, { loader as postsLoader } from './routes/Posts';

import NewPost from './routes/NewPost';

import RootLayout from './routes/RootLayout';

const router = createBrowserRouter([

{

path: '/',

import { RouterProvider, createBrowserRouter } from 'react-router-dom';

element: <RootLayout />,

children: [

{

path: '/',

element: <Posts />,

loader: postsLoader, // Loader fetches data before rendering the Posts component

children: [{ path: '/create-post', element: <NewPost /> }],

},

],

},

]);

ReactDOM.createRoot(document.getElementById('root')).render(

<React.StrictMode>

<RouterProvider router={router} />

</React.StrictMode>

);

**Step 2: Creating the Loader for Data Fetching**

Instead of fetching data inside the PostsList component, you’ll create a loader function that fetches data when the route is activated. This allows React Router to handle the fetching process before rendering the component.

**In Posts.jsx**:

* Create a loader function that fetches the posts data.
* Use async/await to handle the HTTP request.
* Return the fetched posts data, which will be available to the component.

export async function loader() {

const response = await fetch('http://localhost:8080/posts');

const resData = await response.json();

return resData.posts;

}

In this function:

* The loader runs before the component is rendered.
* It fetches the posts data and returns it.
* The returned data will be accessible through React Router’s data hooks.

**Step 3: Accessing Fetched Data in Components with useLoaderData**

Now that the loader is responsible for fetching data, you can access that data inside your component using the useLoaderData hook. This removes the need for managing local component state for fetched data.

**In PostsList.jsx**:

* Import and use the useLoaderData hook from react-router-dom to access the data returned by the loader.
* Replace the old useEffect and useState code with this new approach.

import { useLoaderData } from 'react-router-dom';

import Post from './Post';

import classes from './PostList.module.css';

function PostsList() {

const posts = useLoaderData(); // Fetches the data returned by the loader

return (

<>

{posts.length > 0 && (

<ul className={classes.posts}>

{posts.map((post, index) => (

<Post key={index} author={post.author} body={post.body} />

))}

</ul>

)}

{posts.length === 0 && (

<div style={{ textAlign: 'center', color: 'white', fontSize: '1.5rem' }}>

<h2>No posts yet.</h2>

<p>Start adding some!</p>

</div>

)}

</>

);

}

export default PostsList;

In this step:

* useLoaderData() provides the data fetched by the loader function, making it available to the component.
* The need for useState and useEffect for data fetching is eliminated.
* The component no longer manages fetching logic or loading states, simplifying its structure.

**Step 4: Refactoring Component Logic**

With the data now coming from the loader, you can remove the old state and side-effect management related to data fetching.

* Remove the useEffect hook and useState calls that were previously used to fetch and store posts.
* Remove any manual loading states such as isFetching, as React Router handles rendering the component only after the data is ready.

// Remove these lines:

const [posts, setPosts] = useState([]);

const [isFetching, setIsFetching] = useState(false);

useEffect(() => {

setIsFetching(true);

fetch('http://localhost:8080/posts')

.then((response) => response.json())

.then((data) => {

setPosts(data.posts);

setIsFetching(false);

});

}, []);

Since React Router now waits for the loader to finish fetching before rendering the component, you no longer need a loading indicator or manual state management for fetched data.

**Step 5: Handling Post Creation (Optional Improvement)**

Although the focus here is on data fetching, it’s worth noting that a similar approach can be used for form submissions and post creation. Using React Router’s action property, you could move the post submission logic outside of the component, following the same pattern as for loaders.

**In the current example**:

* The post creation logic is still inside the addPostHandler function, but this could be moved to a dedicated route action for better separation of concerns.

function addPostHandler(postData) {

fetch('http://localhost:8080/posts', {

method: 'POST',

body: JSON.stringify(postData),

headers: {

'Content-Type': 'application/json',

},

});

// You could optimize state management here when adding posts

}

**Recap of the Process:**

1. **Router Setup**: Define routes using createBrowserRouter and assign loaders to fetch data for specific routes.
2. **Loader Creation**: Build a loader function to handle data fetching before rendering the component.
3. **Accessing Loader Data**: Use useLoaderData to access the fetched data within components.
4. **Simplifying Components**: Refactor the components to remove state management and side-effects related to data fetching.
5. **Optional**: Use a similar approach for data submissions (like post creation) by utilizing React Router’s action property.

This approach streamlines your data fetching logic and improves performance by ensuring that the component only renders when the necessary data is available, while reducing the complexity of the component code itself.

## Submitting Data with React Router

With React Router's powerful features, we can streamline data submission from a form, just as we previously leveraged it for data fetching. Let's focus on using React Router's **Form** component and **action** function to handle submitting new data to the backend, like creating a new post.

In this example, we're focusing on the NewPost component that allows users to submit a new post, consisting of a body and an author, through a modal.

### Using React Router's Form Component

Instead of manually handling form submissions and state management, we can now use React Router's Form component. This will automatically handle the form submission, gather the input data, and trigger the route's associated **action** function without requiring extra code to manage form states.

In the NewPost component, we replace the traditional form with React Router's Form:

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

import classes from './NewPost.module.css';

import Modal from '../components/Modal';

function NewPost() {

return (

<Modal>

<Form method="post" className={classes.form}>

<p>

<label htmlFor="body">Text</label>

<textarea id="body" name="body" required rows={3} />

</p>

<p>

<label htmlFor="name">Your name</label>

<input type="text" id="name" name="author" required />

</p>

<p className={classes.actions}>

<Link to=".." type="button">Cancel</Link>

<button>Submit</button>

</p>

</Form>

</Modal>

);

}

export default NewPost;

Here, we use the Form component from React Router, set the method to post, and add name attributes to the inputs, which will be used to extract the form data in the action function.

### Handling Data Submission with the Action Function

The action function is a feature in React Router that handles form submissions linked to a specific route. In this case, when the form is submitted, the action for the create-post route will be executed. We extract the form data from the request, send it to the backend, and then redirect the user back to the main page.

export async function action({ request }) {

const formData = await request.formData();

const postData = Object.fromEntries(formData); // { body: '...', author: '...' }

// Send a POST request to the backend API

await fetch('http://localhost:8080/posts', {

method: 'POST',

body: JSON.stringify(postData),

headers: {

'Content-Type': 'application/json',

},

});

// Redirect to the home page after the post is created

return redirect('/');

}

* request.formData() retrieves the form data submitted by the user.
* Object.fromEntries(formData) converts the form data into a key-value object, with the form input names as keys.
* We send the postData to the backend using fetch() and then use the redirect function to navigate the user back to the main page (/ route).

**3. Updating the Routes Configuration**

We need to ensure that the NewPost component and the action function are connected to the correct route in our router configuration. Here's how to set up the route for creating a post:

import React from 'react';

import ReactDOM from 'react-dom/client';

import { RouterProvider, createBrowserRouter } from 'react-router-dom';

import Posts, { loader as postsLoader } from './routes/Posts';

import './index.css';

import NewPost, { action as newPostAction } from './routes/NewPost';

import RootLayout from './routes/RootLayout';

const router = createBrowserRouter([

{

path: '/',

element: <RootLayout />,

children: [

{

path: '/',

element: <Posts />,

loader: postsLoader,

children: [

{

path: '/create-post',

element: <NewPost />,

action: newPostAction, // Connect the action to the route

},

],

},

],

},

]);

ReactDOM.createRoot(document.getElementById('root')).render(

<React.StrictMode>

<RouterProvider router={router} />

</React.StrictMode>

);

In this setup:

* We link the /create-post route to the NewPost component.
* The action function (newPostAction) is executed when the form is submitted, allowing us to send the form data to the backend.

**4. Summary**

* **Form Submission**: React Router's Form component simplifies form submission by automatically gathering input data and preventing the default form submission behavior.
* **Action Function**: The **action** associated with the route is triggered on form submission, sending the data to the backend and handling any post-submission logic (e.g., redirection).
* **Redirect**: After successfully submitting the post, we redirect the user back to the main page, keeping the application flow smooth.

This approach makes form handling more efficient and reduces the amount of code needed to manage form state and navigation manually.

## Dynamic Routes

To implement a feature where clicking on a post opens the details in a modal overlay, similar to the "New Post" overlay, we need to follow these steps:

### Set up a Dynamic Route

The first thing is to add a route that uses the post ID to load the details. This route will use dynamic routing, allowing you to load any post based on its unique ID.

{

path: '/:id',

element: <PostDetails />,

loader: postDetailsLoader

}

Here, :id is a dynamic parameter that represents the unique ID of each post. When this route is activated, React Router will fetch the post's details and render them in the PostDetails component.

**Create the PostDetails Component**

This component will handle fetching and displaying the post's details in the modal overlay.

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

import Modal from '../components/Modal';

import classes from './PostDetails.module.css';

function PostDetails() {

const post = useLoaderData();

if (!post) {

return (

<Modal>

<main className={classes.details}>

<h1>Could not find post</h1>

<p>Unfortunately, the requested post could not be found.</p>

<p>

<Link to=".." className={classes.btn}>

Okay

</Link>

</p>

</main>

</Modal>

);

}

return (

<Modal>

<main className={classes.details}>

<p className={classes.author}>{post.author}</p>

<p className={classes.text}>{post.body}</p>

</main>

</Modal>

);

}

export default PostDetails;

In this component:

* useLoaderData fetches the post data for the given ID.
* A Modal is used to wrap the post details, creating an overlay effect.
* If the post is not found, an error message is displayed inside the modal.

### Fetch Post Data with a Loader

A loader is responsible for fetching the post data based on the dynamic id parameter. Here’s how to set up the loader function:

export async function loader({ params }) {

const response = await fetch('http://localhost:8080/posts/' + params.id);

const resData = await response.json();

return resData.post;

}

This function sends a request to the backend using the id parameter and returns the post data.

### Display Posts with Links to the Details

In the Post component, wrap each post with a link that points to its dynamic route.

import { Link } from 'react-router-dom';

import classes from './Post.module.css';

function Post({ id, author, body }) {

return (

<li className={classes.post}>

<Link to={id}>

<p className={classes.author}>{author}</p>

<p className={classes.text}>{body}</p>

</Link>

</li>

);

}

export default Post;

Here, the Link component creates a clickable element that navigates to the post's details when clicked.

**Adjust the Styles**

To maintain the look of the overlay and clickable post links, you can style them as needed in the respective CSS files. For instance, you can remove the text decoration for the Link component and adjust the modal's styles to suit your design.

With these steps, clicking on a post will open its details in a modal overlay, similar to how new posts are created. This dynamic routing and loader approach ensures that each post loads its data efficiently, enhancing the user experience.

## Guide Summary

This React crash course focuses on building a demo application where users can create, view, and load post details using React and React Router. Throughout the course, the core concepts of React are covered, including:

* **Components & Props**: Understanding how to break down the UI into reusable components and pass data between them using props.
* **State Management**: Using useState to manage and update the application's state dynamically.
* **Side Effects with useEffect**: Handling data fetching, event listeners, and other side effects that occur in React components.
* **Routing**: Implementing React Router for navigating between pages while maintaining a single-page application (SPA) experience. This includes loading data for specific routes dynamically and submitting data through forms.
* **Simplified Component Code**: By using React Router for data loading and submission, the course demonstrates how the need for manual state management is drastically reduced, leading to cleaner and more maintainable component code.

The course provides a solid foundation for working with React, teaching the essentials to build multi-page applications using routing. It encourages further learning, whether through official React documentation, more in-depth guides, or exploring frameworks built on React, like Next.js or Remix, to develop full-stack applications.

By the end of the course, learners should feel confident in their React skills and be ready to dive deeper into React or start working with more advanced frameworks.