Week 3 - Advanced React Concepts and Component Lifecycle

Amr A Khllaf

July 12, 2025

Contents

1 NavLink & Differences Between Variable & State and Hooks In Top Level

1.1 NavLink

NavLink is a special version of the React Router's Link component that allows you to apply styles to the active link. It is used to create navigation links that can be styled based on whether they are active or not.

1.1.1 Example

```
import { NavLink } from "react-router-dom";
function Navigation() {
  return (
    <nav>
      <NavLink to="/" exact activeClassName="active">
        Home
      </NavLink>
      <NavLink to="/about" activeClassName="active">
        About
      </NavLink>
      <NavLink to="/contact" activeClassName="active">
        Contact
      </NavLink>
    </nav>
  );
}
```

- To style the active link, you can use the `activeClassName` prop to specify a CSS class that will be applied when the link is active.

- Example on using `NavLink` with a custom style:

```
.active {
  font-weight: bold;
  color: blue;
}

/* - To Make the active link more visually distinct, you can also use
↪  inline styles or CSS modules. */

/* -  Can You make a css Style to put a line under the active link but
↪  with modern style */

.active {
  font-weight: bold;
  color: blue;
  border-bottom: 2px solid blue;
}
```

1.2 What is the React Fiber Architecture?

- **React Fiber:** is the reconciliation algorithm used by React to manage the rendering of components. It allows React to break down the rendering work into smaller units, enabling more efficient updates and better performance, especially for complex applications.
- Fiber introduces a new reconciliation algorithm that allows React to pause and resume work, making it more efficient in handling updates and rendering.
- It also allows for better handling of asynchronous rendering, which is crucial for modern web applications that require smooth user experiences.
- Example of React Fiber in action:

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

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

// What is the Difference between React Fiber and Reconciliation
↪  Algorithm?
```

```
// React Fiber is a complete rewrite of the reconciliation algorithm in
↪  React. While the original reconciliation algorithm was based on a
↪  simple tree diffing approach, Fiber introduces a more sophisticated
↪  and incremental rendering strategy. This allows React to pause and
↪  resume work on components, making it more efficient in handling
↪  complex updates and improving the overall user experience.
```

1.3 Differences Between Variable and State

- **Variable:** A variable is a simple container for storing data. It can be changed at any time, and its value is not preserved across re-renders in React. Variables do not trigger re-renders when their values change.
- **State:** State is a special type of variable in React that is managed by the component. It is used to store data that affects the rendering of the component. When the state changes, React re-renders the component to reflect the new state. State is preserved across re-renders, making it suitable for managing dynamic data in your application.
- **Example of Variable vs. State:**

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

function Counter() {
  const [count, setCount] = useState(0);
  let variableCount = 0;

  return (
    <div>
      <p>State Count: {count}</p>
      <p>Variable Count: {variableCount}</p>
      <button onClick={() => setCount(count + 1)}>Increment State</button>
      <button onClick={() => (variableCount += 1)}>Increment
       ↪  Variable</button>
    </div>
  );
}
export default Counter;

// In this example, clicking the "Increment State" button updates the
↪  state and re-renders the component, while clicking the "Increment
↪  Variable" button does not trigger a re-render, and the variable count
↪  will not be displayed correctly.
```

- **Best Practice for Using State Hook:**
  Always use the `useState` hook (and other React hooks) directly inside a function component or a custom hook (**Top Level of the Function Component**). Do **not** call hooks at the top level of your file, outside of any component, or inside regular functions defined within your component. Hooks rely on React's component lifecycle, and using them incorrectly will cause errors or unpredictable behavior.
- Example of using hooks correctly:

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

```jsx
function ExampleComponent() {
  const [count, setCount] = useState(0);

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

// Example of using hooks incorrectly (this will cause an error):
import React, { useState } from "react";
function ExampleComponent() {
  const [count, setCount] = useState(0);

  function increment() {
    // This is a regular function, not a hook
    // Incorrect usage of hooks inside a regular function
    setCount(count + 1);
  }

  // Incorrect usage of hooks outside of the component
  useState(0); // This will cause an error

  return (
    <div>
      <p>Count: {count}</p>
      <button onClick={increment}>Increment</button>
    </div>
  );
}
```

---

2 Intro To Component Life-cycle and Mounting Phase

2.1 Component Lifecycle

- The component lifecycle in React refers to the series of methods that are invoked at different stages of a component's existence, from its creation to its removal from the DOM. Understanding the lifecycle is crucial for managing side effects, optimizing performance, and ensuring that your components behave as expected.
- The lifecycle methods can be categorized into three main phases:
  - 1. **Mounting Phase**
  - 2. **Updating Phase**
  - 3. **Unmounting Phase**
- Each phase has specific methods that are called at various points, allowing you to perform actions like fetching data, updating the UI, or cleaning up resources.

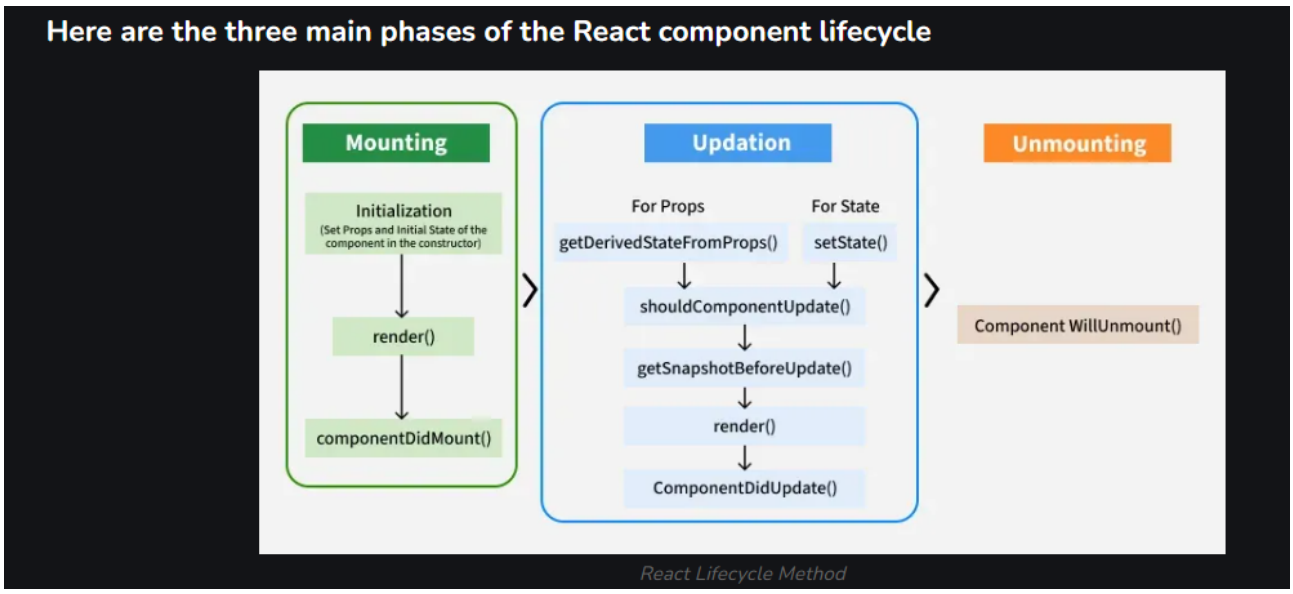

Figure 1: Component Lifecycle Diagram

2.2 Mounting Phase

- The mounting phase is the first phase in the component lifecycle.
- It occurs when a component is being created and inserted into the DOM (**When You Render the Component**). During this phase, the following lifecycle methods are called:
  - `constructor`: This is the first method called when a component is created. It is used to initialize state and bind event handlers.
    - **This is in Class components, not in Function components.**
  - `getDerivedStateFromProps`: This method is called right before rendering and allows the component to update its state based on changes in props.
  - `render`: This method is required and **returns the JSX** that represents the component's UI.
  - `componentDidMount`: This method is **called after the component is mounted in the DOM**. It is a good place to perform side effects, such as fetching data or subscribing to events.

2.2.1 What is the ComponentDidMount Method?

- `componentDidMount` is a lifecycle method in React that is called once, immediately after a component is mounted (inserted into the DOM). It is commonly used for:
  - **Fetching data from an API (Calling API).**
  - Setting up subscriptions or event listeners.
  - Initializing third-party libraries that require access to the DOM.
- It is important to note that `componentDidMount` is only called once during the lifecycle of a component, making it suitable for one-time setup tasks.

2.2.2 Updating Phase

- The updating phase occurs **when a component's state or props change**, causing it to re-render. During this phase, the following lifecycle methods are called:

- `getDerivedStateFromProps`: Called right before rendering, this static method allows the component to update its state in response to changes in props.
- `shouldComponentUpdate`: Invoked before rendering when new props or state are being received. Use this method to optimize performance by preventing unnecessary re-renders.
- `render`: Required method that returns the JSX representing the component's UI.
- `getSnapshotBeforeUpdate`: Called right before the most recently rendered output is committed to the DOM. It enables your component to capture current values (like scroll position) before they are potentially changed.
- `componentDidUpdate`: Called immediately after the component updates in the DOM. This is a good place to perform side effects **based on the `updated state` or `props`.**

2.2.3 Unmounting Phase

- The unmounting phase occurs **when a component is being removed from the DOM** or **when its parent component is unmounted** or **Leaving The Component**. During this phase, the following lifecycle method is called:
  - `componentWillUnmount`: This method is called immediately before a component is unmounted and destroyed. **It is used for cleanup tasks, such as removing event listeners or canceling network requests.**

---

3 UseEffect To Handle The Component Life-Cycle Methods & ComponentDidMount Handling

3.1 What is the UseEffect Hook?

- The `useEffect` hook is a built-in React hook that allows you to perform side effects in function components. It is used to handle tasks like data fetching, subscriptions, and manual DOM manipulations.
- The `useEffect` hook replaces the need for lifecycle methods like `componentDidMount`, `componentDidUpdate`, and `componentWillUnmount` in class components. It provides a more declarative way to manage side effects in function components.
- The `useEffect` hook takes two arguments:
  - **A function that contains the side effect logic.**
  - **(`React.DependencyList`) An optional dependency array that specifies when the effect should run.**
- If the dependency array is empty, the effect runs only once after the initial render (similar to `componentDidMount`) ==> **This is the Default Behavior**.

3.2 What is the Usage of ComponentDidMount in Function Components?

- In function components, you can use the `useEffect` hook to replicate the behavior of `componentDidMount`. By providing an empty dependency array, the effect will run only

once after the initial render, allowing you to perform one-time setup tasks like data fetching or subscriptions.

```jsx
import React, { useState } from "react";
import Navbar from "../Navbar/Navbar";

export default function Home() {
  const userName = "Khllaf";

  // render ==> componentDidMount
  // usage of the componentDidMount? ==> is the best place to Call APIs
  // Mounting Phase ==> ComponentDidMount
  // const [x, setX] = useState(20);
  // useEffect Hook

  useEffect(() => {
    console.log("Hello From componentDidMount");

    setInterval(() => {
      console.log("Hello From setInterval");
    }, 500);
  }, []);

  return (
    <>
      {console.log("Hello From Render")}
      <Navbar />
      <div className="icons"></div>
      <h2>Hello ya {userName}</h2>
      <p className="bg-red-500">
        Lorem ipsum dolor sit amet consectetur adipisicing elit. Ullam quia
↪   quae
        quaerat repellat officiis perspiciatis!
      </p>
    </>
  );
}

// The First Print Statement : Hello From Render
// The Second Print Statement : Hello From componentDidMount

// This mean the Code after `return` will be executed after the component
↪   is mounted
// and the code before `return` will be executed when the component is
↪   rendered.

// The useEffect Hook is used to handle side effects in function
↪   components, such as data fetching or subscriptions. It allows you to
↪   perform actions after the component has rendered, similar to
↪   `componentDidMount` in class components.
```

- There is a problem with the above code, which is that the `setInterval` will keep running even after the component is unmounted, leading to memory leaks. To prevent this, you can return a cleanup function from the `useEffect` hook.
- You Should solve this problem by returning a cleanup function from the `useEffect` hook. This cleanup function will be called when the component is unmounted, allowing you to clear the interval and prevent memory leaks.

---

4 ComponentWillUnmount & Exercise on Cleaning and Remove Listeners

4.1 What is the ComponentWillUnmount Method?

- `componentWillUnmount` is a lifecycle method in React that is called immediately before a component is unmounted and destroyed. It is used for cleanup tasks, such as removing event listeners, canceling network requests, or clearing timers.
- In function components, you can achieve the same functionality using the `useEffect` hook by returning a cleanup function. This cleanup function will be executed when the component is unmounted, allowing you to perform necessary cleanup tasks.

```
import React, { useEffect } from "react";

export default function Home() {
  useEffect(() => {
    // ComponentDidMount Function Scope
    console.log("Hello From componentDidMount");
    const intervalId = setInterval(() => {
      console.log("Hello From setInterval");
    }, 500);

    // ComponentWillUnmount Function Scope
    // Cleanup function
    return () => {
      clearInterval(intervalId);
      console.log("Cleanup function called");
    };
  }, []);
  // [] is the dependency (List) array, meaning this effect runs only once
   ↪  after the initial render.

  return <div>Hello World</div>;
}

// In this example, the `useEffect` hook is used to set up an interval
 ↪  that logs a message every 500 milliseconds. The cleanup function
 ↪  returned from the `useEffect` hook clears the interval when the
 ↪  component is unmounted, preventing memory leaks and ensuring that the
 ↪  interval does not continue running after the component is removed from
 ↪  the DOM.
```

4.2 Exercise on Cleaning and Removing Listeners

- Create a simple React component that sets up an event listener for the `resize` event on the window object. The component should log the current window size whenever the window is resized.

```
import React, { useEffect } from "react";

export default function WindowSizeLogger() {
  useEffect(() => {
    const logWindowSize = () => {
      console.log(`Window size: ${window.innerWidth} x
      ↪  ${window.innerHeight}`);
    };

    window.addEventListener("resize", logWindowSize);

    // Cleanup function
    return () => {
      window.removeEventListener("resize", logWindowSize);
      console.log("Cleanup function called");
    };
  }, []);

  return <div>Resize the window to see the effect.</div>;
}
```

- What if the Clicked function Take a Parameter and i want to remove the listener when the component is unmounted?

```
import React, { useEffect } from "react";

export default function ClickLogger() {
  useEffect(() => {
    // Define the handler with a fixed name parameter
    const handleClick = () => {
      const name = "Mahmoud";
      console.log(`Button clicked by: ${name}`);
    };

    // Add the event listener
    window.addEventListener("click", handleClick);

    // Cleanup function to remove the listener
    return () => {
      window.removeEventListener("click", handleClick);
      console.log("Cleanup function called");
    };
  }, []);

  return <div>Click anywhere on the window to see the effect.</div>;
}
```

4.3 Comparison between ComponentDidMount and ComponentWillUnmount

- `componentDidMount` is called once after the component is mounted, making it suitable for one-time setup tasks like data fetching or subscriptions.
- `componentWillUnmount` is called immediately before the component is unmounted, allowing you to perform cleanup tasks like removing event listeners or canceling network requests.
- The dependency array (`[]`) in `useEffect` determines when the effect runs and when cleanup occurs:
  - If the dependency array is **empty** (`[]`), the effect runs **only once** after the initial render (mounting phase), just like `componentDidMount`. The cleanup function (if provided) runs when the component is unmounted, just like `componentWillUnmount`.
  - If you provide dependencies (e.g., `[count]`), the effect runs **after every render** where any dependency value has changed (updating phase), similar to `componentDidUpdate`. The cleanup function runs before the effect re-runs and when the component unmounts.
  - If you omit the dependency array entirely, the effect runs **after every render**, both on mount and update, and the cleanup runs before each new effect and on unmount.
- By controlling the dependency array, you can make `useEffect` behave like `componentDidMount`, `componentDidUpdate`, and `componentWillUnmount`.

---

5 ComponentDidUpdate Handling in UseEffect Hook and How To Separate The didMount and didUpdate?

5.1 What is the ComponentDidUpdate Method?

- `componentDidUpdate` is a lifecycle method in React that is called immediately after a component updates in the DOM. It is used to perform side effects **based on changes in state or props.**
- In function components, you can achieve the same functionality using the `useEffect` hook. By providing a dependency array, you can control when the effect runs and when it behaves like `componentDidUpdate`.

5.2 How to Separate the `componentDidMount` and `componentDidUpdate`?

- To separate the behavior of `componentDidMount` and `componentDidUpdate`, you can use the `useEffect` hook with different dependency arrays:
  - **For `componentDidMount`:** Use an empty dependency array (`[]`) to run the effect only once after the initial render.
  - **For `componentDidUpdate`:** Use a dependency array with specific state or props that you want to monitor for changes.

```
useEffect(() => {
  // Code to run on component update
}, [someState, someProp]);
```

5.3 Example of Using useEffect to Handle componentDidMount and componentDidUpdate

```
import React from "react";
import { useState } from "react";
export default function Gallery() {
  const [counter, setCounter] = useState(100);
  const [userName, setUserName] = useState("");
  function incrementCounter() {
    setCounter(counter + 1);
  }

  function showUserName() {
    setUserName("Khllaf");
    console.log(`User Name: ${userName}`);
  }

  useEffect(() => {
    console.log("Counter Updated");
  }, [counter, userName]);

  console.log("Increment Counter", counter);
  return (
    <>
      <div className="test3 padding">
        <h2>Gallery Component</h2>
        <h2>Counter: {counter}</h2>
        <h2>User Name: {userName}</h2>
        <button onClick={incrementCounter} className="test padding">
          Click
        </button>
        <button onClick={showUserName} className="test padding margin">
          Show User Name
        </button>
      </div>
    </>
  );
}

// In the above code, the `useEffect` hook acts as both
↪  `componentDidMount` and `componentDidUpdate`. It runs after the
↪  initial render (mounting) and after every update when either `counter`
↪  or `userName` changes, allowing you to handle side effects for both
↪  lifecycle phases in function components,
```

- It Prefer to use the `useEffect` hook to handle componentDidMount and componentWillUnmount in an empty dependency array and another `useEffect` hook to handle componentDidUpdate with specific dependencies. This way, you can clearly separate the logic for each lifecycle method and ensure that your component behaves as expected during updates and unmounting.

Example of Using `useEffect` to Handle `componentDidMount` and `componentDidUpdate`

- It prefer to use componentDidUpdate for every state change.
- If you want a `useEffect` to run only on updates (not on the initial mount), you can use a state variable to track whether the component has mounted. For example:

```
import React, { useEffect, useRef, useState } from "react";

function ExampleComponent() {
  const [count, setCount] = useState(0);
  const hasMounted = useRef(false); // Create a ref to track if the
  ↪  component has mounted (Flag Concept)

  useEffect(() => {
    if (hasMounted.current) {
      // This code runs only on updates, not on initial mount
      console.log("Component updated!");
    } else {
      hasMounted.current = true;
    }
  }, [count]);

  return (
    <button onClick={() => setCount(count + 1)}>Increment
    ↪  ({count})</button>
  );
}
```

- Here, the `hasMounted` ref ensures the effect only runs after the initial mount, so side effects inside the effect are triggered only on updates.
- hasMounted.current : tracks whether the component has mounted or not.
- Explain what is current in the hasMounted.current :
  - `current` is a property of the `useRef` object that holds the current value of the ref. It allows you to store a mutable value that persists across renders without causing re-renders when updated. In this case, it tracks whether the component has mounted or not, allowing you to conditionally run side effects based on the component's lifecycle.
- We Can It by Using the `useState` hook to track whether the component has mounted or not. This way, you can conditionally run side effects based on the component's lifecycle.

```
import React, { useState, useEffect } from "react";
function ExampleComponent() {
  const [count, setCount] = useState(0);
  const [hasMounted, setHasMounted] = useState(false); // Track if the
  ↪  component has mounted (Flag Concept)

  useEffect(() => {
    if (hasMounted) {
      // This code runs only on updates, not on initial mount
      console.log("Component updated!");
    } else {
```

```
      setHasMounted(true); // Set to true after the first render
    }
  }, [count, hasMounted]);

  return (
    <button onClick={() => setCount(count + 1)}>Increment
    ↪  ({count})</button>
  );
}
```

5.3.1 useRef Explanation

- `useRef` is a React hook that creates a mutable object with a `current` property. It is commonly used to access DOM elements or store values that persist across renders without causing re-renders when updated.
- The `current` property of the object returned by `useRef` can be used to store any value, including DOM references, state values, or flags. It does not trigger a re-render when its value changes, making it suitable for storing values that need to persist across renders without affecting the component's rendering behavior.
- `useRef` is often used to:
  - Access and manipulate DOM elements directly.
  - Store values that need to persist across renders without causing re-renders.
  - Create mutable references that can be updated without triggering a re-render.
- Example of using `useRef` to access a DOM element:

```
import React, { useRef } from "react";

function TextInputWithFocusButton() {
  const inputRef = useRef(null);

  const focusInput = () => {
    inputRef.current.focus();
  };

  return (
    <>
      <input ref={inputRef} type="text" />
      <button onClick={focusInput}>Focus Input</button>
    </>
  );
}

export default TextInputWithFocusButton;

// Explanation:
// - `useRef` is used to create a reference (`inputRef`) that can be
↪  attached to a DOM element (the input field).
// - The `ref` attribute is set to `inputRef`, allowing you to access the
↪  input element directly.
```

```
// - The `focusInput` function uses `inputRef.current` to access the input
↪  element and call its `focus` method.
// - When the button is clicked, the input field receives focus.
// - This demonstrates how `useRef` can be used to interact with the DOM
↪  without causing re-renders.
```

---

6 Important Information

- **Hooks**: Hooks are functions that let you use state and other React features in function components. They allow you to manage component state, side effects, and context without using class components.
- **useState**: A hook that lets you add state to function components. It returns an array with the current state value and a function to update it.
- **useEffect**: A hook that lets you perform side effects in function components. It can be used to handle component lifecycle methods like `componentDidMount`, `componentDidUpdate`, and `componentWillUnmount`.
- **useRef**: A hook that creates a mutable object with a `current` property. It is commonly used to access DOM elements or store values that persist across renders without causing re-renders.
- **Dependency Array**: An array passed as the second argument to `useEffect` that determines when the effect should run. If empty, the effect runs only once after the initial render. If it contains dependencies, the effect runs whenever those dependencies change.

6.1 What is the difference if I write a dependency array in useEffect or not?

- If you provide a dependency array to `useEffect`, the effect will run only when the specified dependencies change. This allows you to control when the effect is executed, optimizing performance and preventing unnecessary re-renders.
- If you omit the dependency array, the effect will run after every render, which can lead to performance issues and infinite loops if not handled carefully. It is generally recommended to use a dependency array to avoid unnecessary executions of the effect.

```
import React from "react";
import { useState } from "react";

const [userName, setUserName] = useState("");
function showUserName() {
  setUserName("Khllaf");
  console.log(`User Name: ${userName}`);
}

useEffect(() => {
  console.log("Hello");
});
```

- What will happen if I don't write a dependency array in useEffect?
    - If you don't write a dependency array in useEffect, the effect will run after every

render of the component. This means that any state or props used inside the effect will always have the latest values, but it can lead to performance issues and unnecessary re-renders. It's generally a good practice to include a dependency array to control when the effect should run.

- This is mean it will work as `componentDidMount` and `componentDidUpdate` together, meaning it will run after the initial render and after every update, which can lead to performance issues if not managed properly and **it will lead to an infinite loop if the effect updates state that is used in the effect itself**.

- If you want the effect to run only once after the initial render (like `componentDidMount`), you should provide an empty dependency array (`[]`). This ensures that the effect runs only once and does not cause unnecessary updates.

---

7 Images In React (JSX)

- To organize your project assets, create an `images` folder inside your `assets` directory and place all your image files there. This makes it easier to manage and reference images throughout your React application.
- You can include images in your React components using the `img` tag.
- The `src` attribute should point to the image URL or import the image file.
- You can also use CSS to style the images, such as setting their width and height.
- The JSX Code is Converted and Transformed to HTML Code, so you can use the `img` tag in JSX just like you would in HTML.
  - So the path to the image should be relative to the file where you are using it, or you can import the image at the top of your file.
- The Best Way to Use Images is to import the image at the top of your file and then use it in the `src` attribute of the `img` tag.

```jsx
import React from "react";
import myImage from "./assets/images/myImage.jpg"; // Import the image
function ImageComponent() {
  return (
    <div>
      <img src={myImage} alt="My Image" />
    </div>
  );
}
```

7.1 require Method is like Importing the Image

- The `require` method is a way to dynamically import images in React. It allows you to specify the path to the image file as a string, and React will resolve the path at runtime.
- Svg Images Doesn't work with the `require` method, so you should use the `import` statement for SVG images.
- How to use the `require` method to import images:

```
import React from "react";
function ImageComponent() {
  return (
    <div>
      <img src={require("./assets/images/myImage.jpg")} alt="My Image" />
    </div>
  );
}
```

7.2 PWAs(Progressive Web Apps) and Service Workers

- **Progressive Web Apps (PWAs)** are web applications that use modern web capabilities to deliver an app-like experience to users. They can work offline, send push notifications, and be installed on the user's device.
- **Service Workers** are scripts that run in the background of a web application, separate from the main browser thread. They enable features like caching, background sync, and push notifications, making PWAs more reliable and performant.
- Service Workers act as a proxy between the web application and the network, allowing developers to intercept network requests and cache responses for offline use.
- Search For This Concept in the Internet:
  - **What is a Progressive Web App (PWA)?**
  - **What is a Service Worker?**
  - **How do Service Workers work in PWAs?**
  - **What are the benefits of using PWAs and Service Workers?**

---

8 Axios and Handling Getting Products from API

8.1 What is Axios?

- **Axios** is a popular JavaScript library used to make HTTP requests from the browser or Node.js. It provides a simple and easy-to-use API for sending asynchronous requests to servers, handling responses, and managing errors.
- Axios is built on top of the XMLHttpRequest API and provides a more powerful and flexible way to handle HTTP requests compared to the native `fetch` API.
- It supports features like request cancellation, interceptors, and automatic JSON data transformation, making it a preferred choice for many developers when working with APIs.
- Axios is often used in React applications to fetch data from APIs, submit forms, and handle other HTTP-related tasks.

8.2 How to Use Axios in React

- To use Axios in a React application, you need to install it first. You can do this using npm or yarn:

```
npm install axios
```

or

```
yarn add axios
```

- Once installed, you can import Axios in your React components and use it to make HTTP requests.

8.3 What is the Difference Between Axios and Fetch?

- Axios and Fetch are both used to make HTTP requests in JavaScript, but they have some key differences.
- Here is a comparison table between Axios and Fetch:

Feature	Axios	Fetch
Syntax	Simpler and more concise	More verbose
Request Cancellation	Yes	No
Interceptors	Yes	No
Automatic JSON Transformation	Yes	No
Browser Support	Older browsers	Modern browsers only
Error Handling	Automatic	Manual
Response Type	JSON by default	Text by default
Timeout	Yes	No
Progress Tracking	Yes	No
Upload/Download Progress	Yes	No

- In summary, Axios provides a more feature-rich and user-friendly API for making HTTP requests compared to the native Fetch API. It simplifies error handling, supports request cancellation, and automatically transforms JSON data, making it a popular choice for many developers when working with APIs in JavaScript applications.
- To Use the Axios In Your Component

```
import axios from "axios";
```

- Home.jsx File

```
import React, { useEffect, useState } from "react";
import Navbar from "../Navbar/Navbar";
import axios from "axios";

export default function Home() {
  async function getAllProducts() {
    const result = await axios.get(
      "https://ecommerce.routemisr.com/api/v1/products"
    );
    console.log(result); // This will log the response from the API
    console.log(result.data); // This will log the data part of the
    ↪  response
    console.log("####################");
```

```
    // I can make it by Object Destructuring
    const { data } = await axios.get(
      "https://ecommerce.routemisr.com/api/v1/products"
    );
    console.log(data); // This will log the data part of the response
  }

  useEffect(() => {
    getAllProducts();

    return () => {
      // Cleanup function
    };
  }, []);

  return <></>;
}
```

- In the above code, we are using Axios to make a GET request to the API endpoint to fetch all products. The response is logged to the console, and you can see the structure of the response data.
- You can also use Axios to handle errors and manage loading states while fetching data from the API.

```
import React, { useEffect, useState } from "react";
import axios from "axios";
export default function Home() {
  const [products, setProducts] = useState([]);
  const [loading, setLoading] = useState(true);
  const [error, setError] = useState(null);

  async function getAllProducts() {
    try {
      const { data } = await axios.get(
        "https://ecommerce.routemisr.com/api/v1/products"
      );
      setProducts(data.data); // Assuming the products are in data.data
    } catch (err) {
      setError(err.message);
    } finally {
      setLoading(false);
    }
  }

  useEffect(() => {
    getAllProducts();
  }, []);

  if (loading) return <p>Loading...</p>;
  if (error) return <p>Error: {error}</p>;
```

```
  return (
    <div>
      <h2>Products</h2>
      <ul>
        {products.map((product) => (
          <li key={product._id}>{product.title}</li>
        ))}
      </ul>
    </div>
  );
}
```

- In this example, we are using Axios to fetch products from the API and handle loading and error states. The products are stored in the `products` state, and we render them in a list once they are fetched successfully. If there is an error, it is displayed to the user.

---

9 Displaying Data With Recapping The Component Life Cycle Methods

9.1 Recap of Component Lifecycle Methods

- **Mounting Phase**: The component is created and inserted into the DOM. Lifecycle methods like `constructor`, `getDerivedStateFromProps`, `render`, and `componentDidMount` are called.
- **Updating Phase**: The component's state or props change, causing it to re-render. Lifecycle methods like `getDerivedStateFromProps`, `shouldComponentUpdate`, `render`, `getSnapshotBeforeUpdate`, and `componentDidUpdate` are called.
- **Unmounting Phase**: The component is removed from the DOM. The `componentWillUnmount` method is called for cleanup tasks.

9.2 Grid & Grid-cols-2 in TailwindCSS

- In TailwindCSS, the `grid` class is used to create a grid layout, and `grid-cols-2` specifies that the grid should have two columns. This allows you to easily create responsive layouts with multiple columns.
- You can use TailwindCSS classes to style your components and create responsive designs without writing custom CSS.

```
<div className="grid grid-cols-2">
  <div className="bg-red-500 p-4">Column 1</div>
  <div className="bg-blue-500 p-4">Column 2</div>
</div>

// Explanation:
// - The `grid` class creates a grid layout.
// - The `grid-cols-2` class specifies that the grid should have two
 ↪  columns.
```

9.3 Displaying Data in a Grid Layout

- Home.jsx File

```
import React, { useEffect, useState } from "react";
import Navbar from "../Navbar/Navbar";
import axios from "axios";

export default function Home() {
  const [allProducts, setAllProducts] = useState(null);
  async function getAllProducts() {
    // const result = await axios.get(
    //   "https://ecommerce.routemisr.com/api/v1/products"
    // );
    // console.log(result); // This will log the response from the API
    // console.log(result.data); // This will log the data part of the
    ↪  response
    // console.log("####################");

    // I can make it by Object Destructuring
    const { data } = await axios.get(
      "https://ecommerce.routemisr.com/api/v1/products"
    );
    setAllProducts(data.data);
  }

  useEffect(() => {
    getAllProducts();

    return () => {
      // Cleanup function
    };
  }, []);

  return (
    <>
      <div className="grid grid-cols-3 gap-2">
        {allProducts?.map((product) => (
          <div
            key={product._id}
            className="bg-white p-4 rounded-lg shadow-md hover:shadow-lg
            ↪  transition-shadow duration-300"
          >
            <img
              src={product.imageCover}
              alt={product.title}
              className="w-full  object-cover rounded-md mb-4"
            />
            <h2 className="text-xl font-semibold mb-2">{product.title}</h2>
            {/* <p className="text-gray-600
            ↪  mb-4">{product.description}</p> */}
```

```
            <p className="text-lg font-bold
            ↪  text-green-600">${product.price}</p>
          </div>
        ))}
      </div>
    </>
  );
}

// allProducts?. : mean "optional chaining", it allows you to safely
↪  access deeply nested properties without having to check if each
↪  reference in the chain is nullish (null or undefined).
```

- In this example, we are fetching products from the API and displaying them in a grid layout using TailwindCSS classes. Each product is displayed in a card-like structure with an image, title, and price.
- The TailwindCSS styling in this code is well-structured and effective. Review the classes used, experiment with them yourself, and adjust as needed for your design preferences. TailwindCSS makes it easy to create visually appealing, responsive layouts with minimal custom CSS.

---

10 Recap and React-Loader-Spinner

- When working with APIs, always initialize your state value as `null` at first. This causes an error if you forget to handle the loading screen, reminding you to implement proper loading state management.
  - **Error Explanation:**
    `TypeError: Cannot read properties of null (reading 'map')` occurs when you attempt to use the `map` method on a value that is `null`. In JavaScript, only arrays have the `map` method, so if your state (e.g., `allProducts`) is initialized as `null` and you try to render with `allProducts.map(...)`, this error will be thrown.
  - **How to Fix:**
    Always ensure that the value you are calling `map` on is an array. You can do this by initializing your state as an empty array (`useState([])`) or by using optional chaining (`allProducts?.map(...)`) or a conditional check (`allProducts && allProducts.map(...)`) in your render logic. This prevents the error and ensures your code runs safely even before the data is loaded.
- **React Loader Spinner**:
  - `react-loader-spinner` is a library that provides various loading spinner components for React applications. It allows you to easily add loading indicators while fetching data from APIs or performing asynchronous operations.
- To use `react-loader-spinner`, you need to install it first:

```
npm install react-loader-spinner
```

- Once installed, you can import the desired spinner component and use it in your React components.

10.1 Loading Screen Without Any Library

- We Can do the Loading Screen Without any Library Like this:

```
import React, { useEffect, useState } from "react";
import Navbar from "../Navbar/Navbar";
import axios from "axios";

export default function Home() {
  const [allProducts, setAllProducts] = useState(null);
  async function getAllProducts() {
    // const result = await axios.get(
    //   "https://ecommerce.routemisr.com/api/v1/products"
    // );
    // console.log(result); // This will log the response from the API
    // console.log(result.data); // This will log the data part of the
    ↪  response
    // console.log("####################");

    // I can make it by Object Destructuring
    const { data } = await axios.get(
      "https://ecommerce.routemisr.com/api/v1/products"
    );
    setAllProducts(data.data);
  }

  useEffect(() => {
    getAllProducts();

    return () => {
      // Cleanup function
    };
  }, []);

  return (
    <>
      {allProducts === null ? (
        <div className="h-screen bg-blue-200 flex justify-center
        ↪  items-center">
          <i className="fa-solid fa-spinner fa-spin text-5xl"></i>
        </div>
      ) : (
        <div className="grid grid-cols-3 gap-2">
          {allProducts?.map((product) => (
            <div
              key={product._id}
              className="bg-white p-4 rounded-lg shadow-md hover:shadow-lg
              ↪  transition-shadow duration-300"
            >
              <img
                src={product.imageCover}
```

```
                alt={product.title}
                className="w-full  object-cover rounded-md mb-4"
              />
              <h2 className="text-xl font-semibold
              ↪  mb-2">{product.title}</h2>
              {/* <p className="text-gray-600
              ↪  mb-4">{product.description}</p> */}
              <p className="text-lg font-bold text-green-600">
                ${product.price}
              </p>
            </div>
          ))}
        </div>
      )}
    </>
  );
}
```

10.2 Loading Screen With `react-loader-spinner`

- Or We Can Use the `react-loader-spinner` Library

```
import React, { useEffect, useState } from "react";
import Navbar from "../Navbar/Navbar";
import axios from "axios";
import { FallingLines } from "react-loader-spinner";

export default function Home() {
  const [allProducts, setAllProducts] = useState(null);
  async function getAllProducts() {
    const { data } = await axios.get(
      "https://ecommerce.routemisr.com/api/v1/products"
    );
    setAllProducts(data.data);
  }

  useEffect(() => {
    getAllProducts();

    return () => {
      // Cleanup function
    };
  }, []);

  return (
    <>
      {allProducts === null ? (
        <div className="h-screen bg-blue-200 flex justify-center
        ↪  items-center">
```

```
          {/* <i className="fa-solid fa-spinner fa-spin text-5xl"></i> */}

          {/* Using react-loader-spinner instead of Icon */}
          <FallingLines
            color="#4fa94d"
            width="100"
            visible={true}
            ariaLabel="falling-circles-loading"
          />
        </div>
      ) : (
        <div className="grid grid-cols-3 gap-2">
          {allProducts?.map((product) => (
            <div
              key={product._id}
              className="bg-white p-4 rounded-lg shadow-md hover:shadow-lg
              ↪  transition-shadow duration-300"
            >
              <img
                src={product.imageCover}
                alt={product.title}
                className="w-full  object-cover rounded-md mb-4"
              />
              <h2 className="text-xl font-semibold
              ↪  mb-2">{product.title}</h2>
              {/* <p className="text-gray-600
              ↪  mb-4">{product.description}</p> */}
              <p className="text-lg font-bold text-green-600">
                ${product.price}
              </p>
            </div>
          ))}
        </div>
      )}
    </>
  );
}
```

---

11 CSS Modules

- **CSS Modules** are a way to write CSS that is scoped locally to the component, preventing styles from leaking into other components. This helps avoid naming conflicts and makes it easier to manage styles in large applications.
- To use CSS Modules in a React application, you need to create a CSS file with the `.module.css` extension. This tells React that the styles in this file should be treated as a CSS Module.
- You can then import the CSS Module in your component and use the styles as an object,

where each class name is a property of the object.

- **Rollup and ESBuild** are modern JavaScript bundlers commonly used in React projects. Both support CSS Modules, enabling you to scope styles locally and avoid naming conflicts.
    - These tools bundle and minify all CSS files into a single optimized file for better performance.
    - If you use plain CSS and have duplicate class names across files, the last definition will overwrite previous ones. Using CSS Modules ensures style isolation and prevents such conflicts.

11.1 Steps To Use CSS Modules

1. Create a CSS file with the `.module.css` extension (e.g., `Home.module.css`).
2. Write your styles in this file, using class names as usual.
3. Import the CSS Module in your React component using the `import` statement.
4. Use the styles as an object, where each class name is a property of the imported object.

- Example of a CSS Module file (`Home.module.css`):

```css
/* Home.module.css */
.container {
  padding: 20px;
  background-color: #f0f0f0;
}
.title {
  font-size: 24px;
  color: #333;
}
#description {
  font-size: 16px;
  color: #666;
}
```

```jsx
import styles from "./Home.module.css";

export default function Home() {
  return (
    <div className={styles.container}>
      {/* Use the styles By Class from the CSS Module */}
      <h1 className={styles.title}>Welcome to My Website</h1>
      <p id={styles.description}>
        {/* Use the styles By ID from the CSS Module */}
        This is a simple example of using CSS Modules in React.
      </p>
    </div>
  );
}
```

11.2 Selecting Elements with Custom Attributes in React

If you want to select an element with a custom attribute like `data-custom`, you can use the `querySelector` method in JavaScript. This is useful for applying styles or manipulating the element directly.

Example: Adding a Custom Attribute in JSX

```
import React from "react";
import styles from "./Home.module.css";

export default function Home() {
  return (
    <div className={styles.container}>
      <h1 className={styles.title}>Welcome to My Website</h1>
      <p className={styles.description} data-custom="myCustomData">
        This is a simple example of using CSS Modules in React.
      </p>
    </div>
  );
}
```

Selecting the Element with JavaScript

```
const customElement =
↪  document.querySelector("[data-custom='myCustomData']");
console.log(customElement); // Logs the <p> element with the custom
↪  attribute
```

Tip: Custom data attributes (`data-*`) are a standard way to store extra information on DOM elements and can be easily accessed or manipulated using JavaScript.