**Fetching Data from APIs in React: Best Practices and Methods**

[[Anurag Joshi](https://medium.com/@itsanuragjoshi?source=post_page-----e959e92206f4--------------------------------)](https://medium.com/@itsanuragjoshi?source=post_page-----e959e92206f4--------------------------------)

[Anurag Joshi](https://medium.com/@itsanuragjoshi?source=post_page-----e959e92206f4--------------------------------)

·

Follow

9 min read

·

Oct 15

31

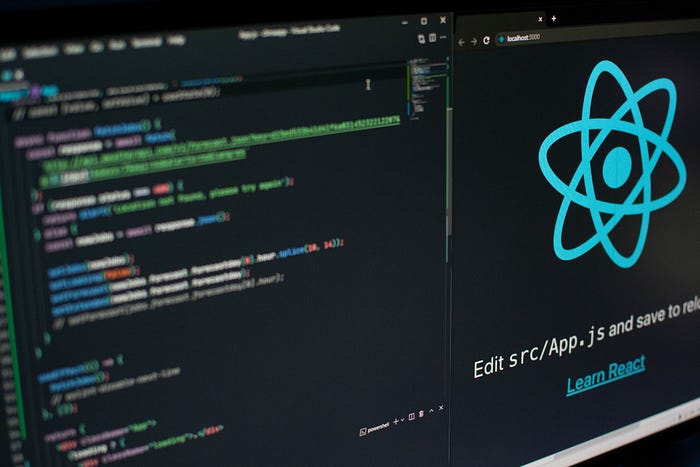


Photo by [Lautaro Andreani](https://unsplash.com/@lautaroandreani?utm_source=medium&utm_medium=referral) on [Unsplash](https://unsplash.com/?utm_source=medium&utm_medium=referral)

**Introduction**

In today’s web development world, fetching data from an API is a fundamental task. In a React application, this process can be achieved in various ways, each with its own advantages and considerations. Moreover, it’s essential to address loading indicators, error handling, and prevent race conditions to ensure a smooth user experience. In this tutorial, we’ll explore four methods for fetching API data in React and cover best practices, including managing loading states, handling errors, and avoiding race conditions using the AbortController.

**Method 1: Fetch API with Promises**

In this method, we utilize the Fetch API in conjunction with Promises to fetch data from an API. This approach is characterized by its simplicity and straightforwardness. Promises provide an elegant way to handle asynchronous operations. The Fetch API, which is available in modern browsers, allows us to make network requests with ease. By combining the two, we can create a clean and concise method for fetching data. This method is suitable for developers who prefer traditional promise-based syntax and want to avoid using additional libraries.

import { useState, useEffect } from "react";  
  
const FetchApi = () => {  
 const BASE\_URL = "https://jsonplaceholder.typicode.com/posts";  
  
 // State to hold fetched data  
 const [posts, setPosts] = useState([]);  
  
 useEffect(() => {  
   
 // Fetch data using Promise with the Fetch API  
 const fetchUsingPromiseWithFetchApi = () => {  
 fetch(BASE\_URL) // Fetch data based on the current page  
 .then((response) => response.json()) // Parse the response as JSON  
 .then((data) => {  
 setPosts(data); // Set the fetched data  
 });  
 };  
  
 // Trigger fetching method on component mount  
 fetchUsingPromiseWithFetchApi();  
  
 }, []); // Run the effect only once on component mount  
  
 return (  
 <div className="container">  
 <h1>Fetching Data in React</h1>  
  
 {/\* Display the fetched data \*/}  
 {posts.map((post) => (  
 <div className="post" key={post.id}>  
 <h3>{post.title}</h3>  
 <p>{post.body}</p>  
 </div>  
 ))}  
 </div>  
 );  
};  
  
export default FetchApi;

**Method 2: Fetch API with Async/Await**

This method introduces a variation in the use of the Fetch API, leveraging the power of async/await. With async/await, we simplify asynchronous code, making it more readable and maintainable. This method is a preferred choice for developers who favor modern JavaScript features and want to take advantage of the concise and sequential structure that async/await offers. It’s especially beneficial when dealing with complex asynchronous operations and multiple API requests.

import { useState, useEffect } from "react";  
  
const FetchApi = () => {  
 const BASE\_URL = "https://jsonplaceholder.typicode.com/posts";  
  
 // State to hold fetched data  
 const [posts, setPosts] = useState([]);  
  
 useEffect(() => {  
  
 // Fetch data using async/await with the Fetch API  
 const fetchUsingAsyncAwaitWithFetchApi = async () => {  
 const response = await fetch(BASE\_URL);  
 const data = await response.json();  
 setPosts(data);  
 };  
  
 // Trigger fetching method on component mount  
 fetchUsingAsyncAwaitWithFetchApi();  
  
 }, []); // Run the effect only once on component mount  
  
 return (  
 <div className="container">  
 <h1>Fetching Data in React</h1>  
  
 {/\* Display the fetched data \*/}  
 {posts.map((post) => (  
 <div className="post" key={post.id}>  
 <h3>{post.title}</h3>  
 <p>{post.body}</p>  
 </div>  
 ))}  
 </div>  
 );  
};  
  
export default FetchApi;

**Method 3: Axios with Promises**

This method involves the use of the [Axios library](https://www.npmjs.com/package/axios), known for its ease of use and extensive capabilities, in combination with Promises. Axios simplifies HTTP requests and provides a wide range of features. This method is well-suited for developers who prefer using a dedicated HTTP client library that simplifies common tasks. It offers promise-based syntax, making it easy to handle network requests and API responses with clarity and precision.

import { useState, useEffect } from "react";  
import axios from "axios"; // Import the Axios library  
  
const FetchApi = () => {  
 const BASE\_URL = "https://jsonplaceholder.typicode.com/posts";  
  
 // State to hold fetched data  
 const [posts, setPosts] = useState([]);  
  
 useEffect(() => {  
  
 // Fetch data using Promise with Axios  
 const fetchUsingPromiseWithAxios = () => {  
 axios  
 .get(BASE\_URL) // Fetch data based on the current page  
 .then(({ data }) => {  
 setPosts(data); // Set the fetched data  
 });  
 };  
  
 // Trigger fetching method on component mount  
 fetchUsingPromiseWithAxios();  
  
 }, []); // Run the effect only once on component mount  
  
 return (  
 <div className="container">  
 <h1>Fetching Data in React</h1>  
  
 {/\* Display the fetched data \*/}  
 {posts.map((post) => (  
 <div className="post" key={post.id}>  
 <h3>{post.title}</h3>  
 <p>{post.body}</p>  
 </div>  
 ))}  
 </div>  
 );  
};  
  
export default FetchApi;

**Method 4: Axios with Async/Await**

This method extends the use of Axios, this time embracing the async/await pattern. With Axios, we can enjoy the benefits of an HTTP client library while adopting the cleaner and more structured approach of async/await. This combination is ideal for developers who want the best of both worlds: the robustness of Axios and the modern simplicity of async/await. It streamlines the process of fetching data and handling responses, making it a reliable choice for complex applications.

import { useState, useEffect } from "react";  
import axios from "axios"; // Import the Axios library  
  
const FetchApi = () => {  
 const BASE\_URL = "https://jsonplaceholder.typicode.com/posts";  
  
 // State to hold fetched data  
 const [posts, setPosts] = useState([]);  
  
 useEffect(() => {  
  
 // Fetch data using async/await with Axios  
 const fetchUsingAsyncAwaitWithAxios = async () => {  
 const { data } = await axios.get(BASE\_URL);  
 setPosts(data); // Set the fetched data  
 };  
  
 // Trigger fetching method on component mount  
 fetchUsingAsyncAwaitWithAxios();  
  
 }, []); // Run the effect only once on component mount  
  
 return (  
 <div className="container">  
 <h1>Fetching Data in React</h1>  
  
 {/\* Display the fetched data \*/}  
 {posts.map((post) => (  
 <div className="post" key={post.id}>  
 <h3>{post.title}</h3>  
 <p>{post.body}</p>  
 </div>  
 ))}  
 </div>  
 );  
};  
  
export default FetchApi;

**Best Practices**

**Adding Loading State**

To enhance the user experience, it’s essential to provide loading indicators while the data is being fetched. This helps users understand that something is happening in the background.

**Handling Fetch Errors**

Errors can occur during the API request process, such as network issues or incorrect endpoints. Proper error handling ensures that your application remains stable and user-friendly. We’ve included error handling in all four methods to handle and display errors gracefully.

**Avoiding Race Conditions using AbortController**

Race conditions can occur when multiple API requests are in progress, and we need to ensure that only the most recent request’s data is used. We’ve addressed this issue by using the AbortController, a feature provided by the Fetch API. It allows us to cancel ongoing requests when new requests are made, preventing conflicts and ensuring that the data displayed is the most up-to-date.

**Code Example**

In this example, we explore a React component that efficiently manages loading states with loading indicators, gracefully handles errors with clear error messages, and effectively prevents race conditions using the AbortController and signal. Let's dive into the code to understand how these features are implemented.

import { useState, useEffect } from "react";  
import axios from "axios";  
  
const FetchApi = () => {  
 const BASE\_URL = "https://jsonplaceholder.typicode.com/posts";  
  
 // State to hold fetched data, loading status, and error  
 const [posts, setPosts] = useState([]);  
 const [isLoading, setIsLoading] = useState(true);  
 const [error, setError] = useState(null);  
 const [page, setPage] = useState(0);  
  
 useEffect(() => {  
 // Create an AbortController for managing the requests  
 const abortController = new AbortController();  
 const signal = abortController.signal;  
  
 // Fetch data using Promise with the Fetch API  
 const fetchUsingPromiseWithFetchApi = () => {  
 fetch(`${BASE\_URL}?page=${page}`, { signal }) // Fetch data based on the current page  
 .then((response) => response.json()) // Parse the response as JSON  
 .then((data) => {  
 setPosts(data); // Set the fetched data  
 setError(null); // Clear any previous errors  
 })  
 .catch((error) => {  
 if (error.name === "AbortError") {  
 console.log("Fetch aborted"); // Log a message when the request is intentionally aborted  
 return; // Exit the function to prevent further error handling  
 }  
 setError(error.message); // Handle and set the error message  
 })  
 .finally(() => setIsLoading(false)); // Turn off loading indicator  
 };  
  
 // Fetch data using async/await with the Fetch API  
 const fetchUsingAsyncAwaitWithFetchApi = async () => {  
 try {  
 const response = await fetch(`${BASE\_URL}?page=${page}`, { signal }); // Fetch data based on the current page  
 const data = await response.json();  
 setPosts(data);  
 setError(null);  
 } catch (error) {  
 if (error.name === "AbortError") {  
 console.log("Fetch aborted"); // Log a message when the request is intentionally aborted  
 return; // Exit the function to prevent further error handling  
 }  
 setError(error.message);  
 } finally {  
 setIsLoading(false);  
 }  
 };  
  
 // Fetch data using Promise with Axios  
 const fetchUsingPromiseWithAxios = () => {  
 axios  
 .get(`${BASE\_URL}?page=${page}`, { signal }) // Fetch data based on the current page  
 .then(({ data }) => {  
 setPosts(data); // Set the fetched data  
 setError(null); // Clear any previous errors  
 })  
 .catch((error) => {  
 if (axios.isCancel(error)) {  
 console.log("Fetch aborted"); // Log a message when the request is intentionally aborted  
 return; // Exit the function to prevent further error handling  
 }  
 setError(error.message); // Handle and set the error message  
 })  
 .finally(() => setIsLoading(false)); // Turn off loading indicator  
 };  
  
 // Fetch data using async/await with Axios  
 const fetchUsingAsyncAwaitWithAxios = async () => {  
 try {  
 const { data } = await axios.get(`${BASE\_URL}?page=${page}`, {  
 signal,  
 }); // Fetch data based on the current page  
 setPosts(data); // Set the fetched data  
 setError(null); // Clear any previous errors  
 } catch (error) {  
 if (axios.isCancel(error)) {  
 console.log("Fetch aborted"); // Log a message when the request is intentionally aborted  
 return; // Exit the function to prevent further error handling  
 }  
 setError(error.message); // Handle and set the error message  
 } finally {  
 setIsLoading(false); // Turn off loading indicator  
 }  
 };  
  
 // Trigger all fetching methods on component mount  
 fetchUsingPromiseWithFetchApi();  
 fetchUsingAsyncAwaitWithFetchApi();  
 fetchUsingPromiseWithAxios();  
 fetchUsingAsyncAwaitWithAxios();  
  
 // Cleanup: Abort the controller and set loading to true when the component unmounts  
 return () => {  
 abortController.abort(); // Cancel any ongoing requests  
 setIsLoading(true); // Reset loading state  
 };  
 }, [page]); // Re-run the effect when the 'page' state changes  
  
 return (  
 <div className="container">  
 <h1>Fetching Data in React</h1>  
 {/\* Button to load more data \*/}  
 <button onClick={() => setPage(page + 1)}>Fetch API ({page})</button>  
  
 {/\* Display errors if any \*/}  
 {error && <div>{error}</div>}  
  
 {/\* Display loading indicator \*/}  
 {isLoading && <div>Loading...</div>}  
  
 {/\* Display the fetched data \*/}  
 {posts.map((post) => (  
 <div className="post" key={post.id}>  
 <h3>{post.title}</h3>  
 <p>{post.body}</p>  
 </div>  
 ))}  
 </div>  
 );  
};  
  
export default FetchApi;

The code shared above handles loading, errors, and prevents race conditions effectively. Let’s break down how it accomplishes these tasks:

**Loading Handling**

* The isLoading state variable is used to manage the loading state. It is initially set to true, indicating that data is being fetched.
* When a request is initiated, isLoading remains true, displaying a "Loading..." message.
* Once the data is successfully fetched and processed, or if an error occurs, the finally block in each of the fetch method sets isLoading to false, indicating that the loading process is complete.
* This ensures that a loading indicator is shown while the data is being fetched, and it disappears once the data is ready, providing a better user experience.

**Error Handling**

* The error state variable is used to store and display error messages.
* If an error occurs during a fetch request, it is caught and handled within the catch block.
* The error message is then set to the error state variable.
* The error message is displayed in the component only when error is not null, showing users what went wrong.

**Preventing Race Conditions**

* To prevent race conditions, the code utilizes the AbortController and its associated signal. This is crucial when the "Fetch API" button is spammed, causing rapid requests for the next page of data.
* Before making a new request, the code cancels any ongoing requests by calling abortController.abort(). This action effectively aborts the previous request initiated by the previous button click.
* The code then proceeds to make a new request for the updated page.
* If a previous request is canceled, a “Fetch aborted” message is logged in the console. This ensures that only the most recent request is processed and displayed.
* As a result, the code addresses race conditions by ensuring that only one request (the most recent one) is active at any given time. This prevents conflicts and guarantees that the displayed data is always the most up-to-date.

In summary, the code effectively handles loading by displaying a loading indicator while data is being fetched. It handles errors by capturing and displaying error messages. It prevents race conditions by canceling ongoing requests and ensuring that only the most recent request’s data is displayed. These practices ensure a smoother and more reliable user experience when working with dynamic data loading in a React application.

**Conclusion**

In this tutorial, we explored four different methods for fetching API data in React, each with its own strengths and use cases. We emphasized the importance of adding loading states, handling errors, and avoiding race conditions using the AbortController. By following these best practices, you can create a more robust and user-friendly application that provides a seamless experience when working with external data sources.

Whether you choose the simplicity of the Fetch API or the feature-rich Axios library, understanding the nuances of data fetching in React is crucial for building modern web applications. The choice of method ultimately depends on your project’s requirements and your development preferences.