Axios is a promise-based HTTP client for JavaScript, often used with React to make API requests. It works in both the browser and Node.js environments, making it a versatile tool for interacting with external APIs or backend services.

**Key Features of Axios:**

1. **Promise-based**: Axios makes use of JavaScript's Promise API to handle asynchronous operations, allowing for cleaner and more readable code.
2. **Client-side and Server-side**: It can be used in both browser-based and Node.js environments.
3. **Interceptors**: Axios supports request and response interceptors, which allow you to modify requests or handle responses globally.
4. **Automatic JSON Data Transformation**: Axios automatically transforms JSON data when sending or receiving it.
5. **Cancellation**: You can cancel requests using Axios.

Using a GET request

import React, { useEffect, useState } from 'react';

import axios from 'axios';

const FetchDataComponent = () => {

const [data, setData] = useState([]);

const [loading, setLoading] = useState(true);

const [error, setError] = useState(null);

useEffect(() => {

const fetchData = async () => {

try {

const response = await axios.get('https://jsonplaceholder.typicode.com/posts');

setData(response.data);

setLoading(false);

} catch (error) {

setError(error);

setLoading(false);

}

};

fetchData();

}, []);

if (loading) return <p>Loading...</p>;

if (error) return <p>Error: {error.message}</p>;

return (

<div>

<h1>Posts</h1>

<ul>

{data.map(post => (

<li key={post.id}>{post.title}</li>

))}

</ul>

</div>

);

};

export default FetchDataComponent;

Using a POST request

import React, { useState } from 'react';

import axios from 'axios';

const SubmitFormComponent = () => {

const [title, setTitle] = useState('');

const [body, setBody] = useState('');

const [response, setResponse] = useState(null);

const handleSubmit = async (e) => {

e.preventDefault();

try {

const result = await axios.post('https://jsonplaceholder.typicode.com/posts', {

title,

body,

});

setResponse(result.data);

} catch (error) {

console.error("There was an error!", error);

}

};

return (

<div>

<h1>Submit a Post</h1>

<form onSubmit={handleSubmit}>

<div>

<label>Title:</label>

<input

type="text"

value={title}

onChange={(e) => setTitle(e.target.value)}

/>

</div>

<div>

<label>Body:</label>

<textarea

value={body}

onChange={(e) => setBody(e.target.value)}

/>

</div>

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

</form>

{response && (

<div>

<h2>Response</h2>

<pre>{JSON.stringify(response, null, 2)}</pre>

</div>

)}

</div>

);

};

export default SubmitFormComponent;

### Axios Interceptors

Axios interceptors are functions that Axios runs before a request is sent or after a response is received. They are useful for tasks such as adding authentication tokens to headers, logging request and response details, or handling errors globally.

#### Request Interceptor

A request interceptor runs before a request is sent. You can use it to modify the request, for example, by adding authentication headers.

#### Response Interceptor

A response interceptor runs after the response is received but before it is processed by the .then() or .catch() methods. You can use it to handle responses globally, log data, or handle errors.

### Example: Setting Up Axios Interceptors

Here's how to set up Axios interceptors:

import axios from 'axios';

// Add a request interceptor

axios.interceptors.request.use(

(config) => {

// Do something before request is sent, such as adding a token

console.log('Request Interceptor:', config);

config.headers.Authorization = 'Bearer your\_token\_here';

return config;

},

(error) => {

// Do something with request error

return Promise.reject(error);

}

);

// Add a response interceptor

axios.interceptors.response.use(

(response) => {

// Do something with response data

console.log('Response Interceptor:', response);

return response;

},

(error) => {

// Do something with response error

if (error.response.status === 401) {

// handle unauthorized errors globally

console.error('Unauthorized access - maybe redirect to login?');

}

return Promise.reject(error);

}

);

### Cancelling Requests

Sometimes, you might need to cancel an ongoing request. This is useful in scenarios where the user navigates away from a component, or when making new requests that render previous ones obsolete.

To cancel a request in Axios, you use a CancelToken.

### Example: Cancelling a Request

#### Step 1: Create a Component with a Cancellable Request

import React, { useEffect, useState } from 'react';

import axios from 'axios';

const CancelRequestComponent = () => {

const [data, setData] = useState([]);

const [loading, setLoading] = useState(true);

useEffect(() => {

const source = axios.CancelToken.source();

const fetchData = async () => {

try {

const response = await axios.get('https://jsonplaceholder.typicode.com/posts', {

cancelToken: source.token,

});

setData(response.data);

setLoading(false);

} catch (error) {

if (axios.isCancel(error)) {

console.log('Request canceled', error.message);

} else {

// handle error

}

}

};

fetchData();

// Cleanup function to cancel the request

return () => {

source.cancel('Operation canceled by the user.');

};

}, []);

if (loading) return <p>Loading...</p>;

return (

<div>

<h1>Posts</h1>

<ul>

{data.map(post => (

<li key={post.id}>{post.title}</li>

))}

</ul>

</div>

);

};

export default CancelRequestComponent;

**Explanation:**

1. **Creating a CancelToken**:
   * const source = axios.CancelToken.source(); creates a new cancel token source. This token can be used to cancel the request.
2. **Making a Request with a CancelToken**:
   * The fetchData function makes an Axios GET request and includes the cancelToken: source.token in the request configuration. This ties the request to the cancel token.
3. **Handling Cancellation**:
   * If the request is canceled, Axios throws an error, which can be identified using axios.isCancel(error). This allows you to differentiate between a cancellation and other types of errors.
4. **Cleanup Function**:
   * The useEffect cleanup function calls source.cancel('Operation canceled by the user.'); to cancel the request if the component unmounts before the request completes. This prevents trying to update state on an unmounted component, which would result in a memory leak.