UISNG AXIOS WITH INTERCEPTORS

The following code implements Axios Interceptors:

1. Imports the axios library.
2. Creates an instance of axios with a base URL for your API. This is optional but recommended for cleaner code.
3. **Request interceptor:**
   - The `api.interceptors.request.use()` function adds an interceptor that will be called before any request is sent.
   - The first function in `use()` handles successful requests. Here, you can log the request config and modify it if needed (e.g., add authorization headers).
   - The second function handles request errors.
4. **Response interceptor:**
   - The `api.interceptors.response.use()` function adds an interceptor that will be called after a response is received.
   - The first function handles successful responses (status code 2xx).
   - The second function handles error responses (status code other than 2xx).
5. Use the `api` instance to make a GET request.
6. Emulates a long loading process by adding a timer. You can cancel it.
7. Loading message.

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

const api = axios.create({
  baseURL: 'https://jsonplaceholder.typicode.com'
});

function App() {
  const [data, setData] = useState(null);
  const [loading, setLoading] = useState(false);
  const [cancelTokenSource, setCancelTokenSource] =
useState(null);

  useEffect(() => {
    const source = axios.CancelToken.source();
    setCancelTokenSource(source);

    return () => {
```

```
      source.cancel('Component unmounted');
    };
  }, []);

  const fetchData = async () => {
    setLoading(true);

    try {
      // Introduce an artificial delay for demonstration
purposes
      await new Promise(resolve => setTimeout(resolve,
2000)); // Wait 2 seconds

      const response = await api.get('/posts/1', {
        cancelToken: cancelTokenSource.token,
      });
      setData(response.data);
    } catch (error) {
      if (axios.isCancel(error)) {
        console.log('Request canceled', error.message);
      } else {
        console.error('Error fetching data:', error);
      }
    } finally {
      setLoading(false);
    }
  };

  const cancelRequest = () => {
    if (cancelTokenSource) {
      cancelTokenSource.cancel('Request canceled by user');
    }
  };

  return (
    <div>
      <button onClick={fetchData} disabled={loading}>
        {loading ? 'Loading...' : 'Fetch Data'}
      </button>
      <button onClick={cancelRequest} disabled={!loading}>
```

```
        Cancel
      </button>

      {loading && <p>Fetching data...</p>} {/* Loading
indicator */}

      {data && (
        <div>
          <h2>Post Data:</h2>
          <pre>{JSON.stringify(data, null, 2)}</pre>
        </div>
      )}
    </div>
  );
}

export default App;
```

.

Axios interceptors vs other tools

- **Benefits of using Axios interceptors:**
    - **Centralized:** Handle logic in one place, keeping your components clean.
    - **Reusable:** Apply the same behavior to multiple requests.
    - **Powerful:** Can do more than just loading and canceling (e.g., authentication, logging).

Alternatives to using Axios interceptors:

1. **Write directly in `async/await` functions:**
    - **Pros:** Simple and straightforward for basic cases.
    - **Cons:** Can lead to code duplication if you have many requests.

```
const fetchData = async () => {
  setLoading(true);
  try {
    const response = await api.get('/posts/1');
    setData(response.data);
  } catch (error) { /* ... */ }
  finally { setLoading(false); }
```

```
};
```

2. **Write in custom hooks:**
    - **Pros:** Encapsulate fetching logic, including loading and canceling, into reusable hooks.
    - **Cons:** Might require more initial setup.

```
function useFetchData(url) {
  const [data, setData] = useState(null);
  const [loading, setLoading] = useState(false);
  // ... (cancellation logic using AbortController) ...

  useEffect(() => {
    fetchData(); // Function to fetch data and handle
loading/canceling
  }, [url]);

  return { data, loading };
}
```

Using the right tools for the job: Axios interceptors can handle loading and canceling, there are scenarios where hooks or state management libraries might be a better fit.

1. Component-Specific Logic

- **Hooks:** If the loading and canceling behavior is tightly coupled to a specific component and not needed globally, a custom hook can encapsulate that logic neatly within the component.
- **Example:** A component that fetches user data might have a `useUserData` hook that handles loading state and allows canceling the fetch request if the user navigates away.

2. Complex State Interactions

- **State Management (Redux):** When you have multiple components interacting with the same data and need to manage loading state, errors, and cancellation in a centralized and predictable way, a state management library can be very helpful.

- **Example:** An e-commerce app with a shopping cart. Adding/removing items, updating quantities, and applying discounts can all trigger API requests with loading states and potential cancellations. A state management library helps coordinate this complex state.

3. Improved Code Organization

- **Hooks:** Hooks can help you extract and reuse fetching logic across different components, making your code more organized and maintainable.

- **State Management:** For large applications, state management libraries provide a structured way to organize data fetching, updates, and related logic, improving overall code architecture.

4. Testability

- **Hooks:** Custom hooks are easier to test in isolation compared to interceptors, as they are more focused and have well-defined inputs and outputs.

- **State Management:** State management libraries often come with tools and patterns for testing actions, reducers, and state changes, making it easier to test your data fetching logic.

Axios interceptors are great for global, request-level concerns, but hooks and state management tools are good when:

- You need component-specific loading/canceling.
- You have complex state interactions related to data fetching.
- You want to improve code organization and reusability.
- You need better testability for your data fetching logic.