API Calls with Axios

Introduction to API Calls

Understanding API Calls

In modern web development, applications often need to interact with external servers or databases to fetch data. APIs (Application Programming Interfaces) are sets of rules and protocols that allow different software applications to communicate with each other. API calls are requests made by your Vue.js application to an external API to retrieve data, perform actions, or receive information.

Example Scenario:

Imagine you are building a Vue.js weather app that needs to display the current weather for a city. To get this data, your application would make an API call to a weather service API, sending a request for the weather information of the desired city. The API would then respond with the relevant weather data, such as temperature, humidity, and weather condition.

RESTful APIs

REST (Representational State Transfer) is a common architectural style for designing APIs. RESTful APIs use standard HTTP methods like GET, POST, PUT, and DELETE to perform operations on resources. For example, GET is used to retrieve data, POST is used to create new data, PUT is used to update existing data, and DELETE is used to remove data.

Installing Axios and Setting up AxiosInstance

In this step, we'll install Axios and set up a single Axios instance with all API requests as properties. This will allow us to manage API calls more efficiently in our Vue.js application.

Installing Axios

To get started, make sure you have Node.js and npm (Node Package Manager) installed. If you are using Vue CLI, Axios is not included by default, so you need to install it manually.

Open your terminal or command prompt, navigate to your Vue.js project directory, and run the following command to install Axios:

```
npm install axios
```

Setting up AxiosInstance

Next, let's create a separate file named `axiosInstance.js` in your project's `src` directory. This file will contain the Axios instance with all API requests as properties.

Example - axiosInstance.js:

```
import axios from 'axios';

const axiosInstance = axios.create({
```

```
  baseURL: '<https://jsonplaceholder.typicode.com>', // Replace with your API base URL
  timeout: 5000, // Set a timeout (in milliseconds) for the requests
});

export const api = {
  getPosts() {
    return axiosInstance.get('/posts');
  },
  createPost(postData) {
    return axiosInstance.post('/posts', postData);
  },
  updatePost(postId, postData) {
    return axiosInstance.put(`/posts/${postId}`, postData);
  },
  deletePost(postId) {
    return axiosInstance.delete(`/posts/${postId}`);
  }
};

export default axiosInstance;
```

In this example, we created the `axiosInstance` using `axios.create()` with the base URL and timeout configuration. Then, we defined the API requests as properties of the `api` object.

Using the API Functions in a Vue Component

Now, let's use the API functions in a Vue component to demonstrate how to make API calls using the Axios instance.

Example - PostList.vue:

```
<template>
  <div>
    <h1>Post List</h1>
    <button @click="getPosts">Get Posts</button>
    <div v-if="isLoading">Loading...</div>
    <div v-else v-if="posts">
      <div v-for="post in posts" :key="post.id">
        <h2>{{ post.title }}</h2>
        <p>{{ post.body }}</p>
      </div>
    </div>
  </div>
</template>

<script>
import { api } from '../axiosInstance';

export default {
  data() {
    return {
```

```
      posts: null,
      isLoading: false
    };
  },
  methods: {
    async getPosts() {
      this.isLoading = true;
      try {
        const response = await api.getPosts();
        this.posts = response.data;
        this.isLoading = false;
      } catch (error) {
        console.error('Error fetching posts:', error);
        this.isLoading = false;
      }
    }
  }
};
</script>
```

Explanation:

- We created a single Axios instance named `axiosInstance` in `axiosInstance.js`, which contains the base URL and timeout configuration for all requests.
- We defined the API requests as properties of the `api` object, making them easily accessible throughout the application.
- In the Vue component, we imported the `api` object and used the `api.getPosts()` function to fetch posts when the button is clicked.
- The `async/await` syntax is used to handle the asynchronous nature of the API call, and we store the response data in the `posts` data property.
- Error handling is included in case the API call encounters an error.

Note:

- The example uses the JSONPlaceholder API (`https://jsonplaceholder.typicode.com`) as a placeholder for the actual API you will use in your project.
- You can add other request configurations to the `axiosInstance`, such as headers or interceptors, to customize the behavior of all requests made using this instance.

By creating a single Axios instance with all API requests as properties, we centralize and organize API calls in a clean and maintainable way. This approach makes it easier to manage request configurations and promotes code reusability throughout your Vue.js application.

GET Requests with Axios

In this step, we'll focus on making GET requests with Axios, retrieving data from an API, and handling the responses in a Vue.js application.

Making GET Requests with AxiosInstance:

In the `axiosInstance.js` file, we can add functions to handle GET requests using Axios.

Example - axiosInstance.js:

```
import axios from 'axios';

const axiosInstance = axios.create({
  baseURL: '<https://jsonplaceholder.typicode.com>', // Replace with your API base URL
  timeout: 5000, // Set a timeout (in milliseconds) for the requests
  retry: {
    retries: 3, // Number of retries for failed requests
    retryDelay: (retryCount) => retryCount * 1000 // Delay between retries (in millise
conds)
  }
});

// ... (interceptors and other configurations remain the same)

export const api = {
  // ... (other API functions remain the same)

  // Example of handling a GET request
  async getUser(userId) {
    try {
      const response = await axiosInstance.get(`/users/${userId}`);
      return response.data;
    } catch (error) {
      console.error('Error fetching user:', error);
      throw error; // Re-throw the error to be handled at the component level
    }
  },

  // Example of handling a GET request for a list of users
  async getUsers() {
    try {
      const response = await axiosInstance.get('/users');
      return response.data;
    } catch (error) {
      console.error('Error fetching users:', error);
      throw error; // Re-throw the error to be handled at the component level
    }
  }
};
```

```
export default axiosInstance;
```

Explanation:

- We added two new functions to the `api` object: `getUser(userId)` and `getUsers()`.
- The `getUser(userId)` function is an example of fetching a specific user by their ID, and the `getUsers()` function fetches a list of all users from the server.
- Each function uses Axios' `axiosInstance.get()` method to make the corresponding GET request.

Using GET Requests in a Vue Component:

Now, let's use these functions to perform GET requests in a Vue component.

Example - UserList.vue:

```
<template>
  <div>
    <h2>User List</h2>
    <ul v-if="isLoading">
      <li v-for="user in users" :key="user.id">{{ user.name }}</li>
    </ul>
    <p v-else>No users found.</p>
  </div>
</template>

<script>
import { api } from '../axiosInstance';

export default {
  data() {
    return {
      users: [],
      isLoading: true
    };
  },
  async created() {
    try {
      this.users = await api.getUsers();
      this.isLoading = false;
    } catch (error) {
      console.error('Error fetching users:', error);
      this.isLoading = false;
      // Handle the error gracefully (e.g., show an error message to the user)
    }
  }
};
</script>
```

Explanation:

- In the `UserList.vue` component, we use `api.getUsers()` to fetch the list of all users from the server.
- The `v-if` directive is used to conditionally render either the user list or a message if there are no users.
- The `v-for` directive is used to loop through the `users` array and display each user's name in a list item.

Note:

- The examples provided in this step use the JSONPlaceholder API (`https://jsonplaceholder.typicode.com`) as a placeholder for the actual API you will use in your project.

With the addition of the GET request handling, we now have a comprehensive understanding of making various types of API calls with Axios in Vue.js applications. We can fetch data, update resources, patch data, and delete resources using the Axios instance, which allows us to manage our API calls efficiently and handle errors gracefully. In the next steps, we'll explore advanced features, such as interceptors, canceling requests, and handling authentication.

Handling PUT, PATCH, and DELETE Requests with AxiosInstance

In the `axiosInstance.js` file, we can add functions to handle PUT, PATCH, and DELETE requests using Axios.

Example - axiosInstance.js:

```
import axios from 'axios';

const axiosInstance = axios.create({
  baseURL: 'https://jsonplaceholder.typicode.com', // Replace with your API base URL
  timeout: 5000, // Set a timeout (in milliseconds) for the requests
  retry: {
    retries: 3, // Number of retries for failed requests
    retryDelay: (retryCount) => retryCount * 1000 // Delay between retries (in millise
conds)
  }
});

// ... (interceptors and other configurations remain the same)

export const api = {
  // ... (other API functions remain the same)
```

```
  // Example of handling a PUT request
  async updateUser(userId, userData) {
    try {
      const response = await axiosInstance.put(`/users/${userId}`, userData);
      return response.data;
    } catch (error) {
      console.error('Error updating user:', error);
      throw error; // Re-throw the error to be handled at the component level
    }
  },

  // Example of handling a PATCH request
  async patchUser(userId, userData) {
    try {
      const response = await axiosInstance.patch(`/users/${userId}`, userData);
      return response.data;
    } catch (error) {
      console.error('Error patching user:', error);
      throw error; // Re-throw the error to be handled at the component level
    }
  },

  // Example of handling a DELETE request
  async deleteUser(userId) {
    try {
      const response = await axiosInstance.delete(`/users/${userId}`);
      return response.data;
    } catch (error) {
      console.error('Error deleting user:', error);
      throw error; // Re-throw the error to be handled at the component level
    }
  }
};

export default axiosInstance;
```

Explanation:

- We added three new functions to the `api` object: `updateUser(userId, userData)`, `patchUser(userId, userData)`, and `deleteUser(userId)`.
- These functions are examples of handling PUT, PATCH, and DELETE requests, respectively, for updating user data, patching user data, and deleting a user from the server.
- Each function uses Axios' `axiosInstance.put()`, `axiosInstance.patch()`, and `axiosInstance.delete()` methods to make the corresponding requests.

Using PUT, PATCH, and DELETE Requests in a Vue Component:

Now, let's use these functions to perform PUT, PATCH, and DELETE requests in a Vue component.

Example - UserEdit.vue:

```
<template>
  <div>
    <h2>Edit User</h2>
    <form @submit.prevent="updateUser">
      <div>
        <label for="name">Name:</label>
        <input type="text" id="name" v-model="user.name" required>
      </div>
      <div>
        <label for="email">Email:</label>
        <input type="email" id="email" v-model="user.email" required>
      </div>
      <button type="submit">Update User</button>
    </form>
    <button @click="patchUser">Patch User</button>
    <button @click="deleteUser">Delete User</button>
  </div>
</template>

<script>
import { api } from '../axiosInstance';

export default {
  data() {
    return {
      user: null,
      userId: 1 // Replace with the ID of the user you want to edit
    };
  },
  async created() {
    try {
      this.user = await api.getUser(this.userId);
    } catch (error) {
      console.error('Error fetching user:', error);
      // Handle the error gracefully (e.g., show an error message to the user)
    }
  },
  methods: {
    async updateUser() {
      try {
        const updatedUser = await api.updateUser(this.userId, this.user);
        console.log('Updated User:', updatedUser);
      } catch (error) {
        console.error('Error updating user:', error);
        // Handle the error gracefully (e.g., show an error message to the user)
      }
    },
    async patchUser() {
      try {
        const patchedUser = await api.patchUser(this.userId, { email: 'updated@exampl
e.com' });
```

```
        console.log('Patched User:', patchedUser);
      } catch (error) {
        console.error('Error patching user:', error);
        // Handle the error gracefully (e.g., show an error message to the user)
      }
    },
    async deleteUser() {
      try {
        await api.deleteUser(this.userId);
        console.log('User deleted successfully.');
      } catch (error) {
        console.error('Error deleting user:', error);
        // Handle the error gracefully (e.g., show an error message to the user)
      }
    }
  }
};
</script>
```

Explanation:

- In the `UserEdit.vue` component, we have a form to update the user's name and email fields. We also have two buttons, one for patching the user's email and the other for deleting the user.
- The form uses the `updateUser` method to handle the submit event and perform the PUT request to update the user's data.
- The `patchUser` method is called when the "Patch User" button is clicked, and it performs a PATCH request to update the user's email.
- The `deleteUser` method is called when the "Delete User" button is clicked, and it performs a DELETE request to delete the user from the server.

Interceptors and Request Configurations

In this step, we'll learn about Axios interceptors and how to configure requests globally using the Axios instance. Interceptors allow us to intercept requests and responses, which can be useful for adding headers, handling errors, or performing transformations.

Using Interceptors in AxiosInstance

In the `axiosInstance.js` file, we can add interceptors to customize request configurations globally.

Example - axiosInstance.js:

```
import axios from 'axios';

const axiosInstance = axios.create({
  baseURL: '<https://jsonplaceholder.typicode.com>', // Replace with your API base URL
  timeout: 5000, // Set a timeout (in milliseconds) for the requests
});

// Request Interceptor: Add an authorization header to all requests (if needed)
axiosInstance.interceptors.request.use(
  (config) => {
    config.headers.Authorization = 'Bearer your-access-token'; // Replace with your ac
cess token
    return config;
  },
  (error) => {
    return Promise.reject(error);
  }
);

// Response Interceptor: Handle common error responses (e.g., unauthorized)
axiosInstance.interceptors.response.use(
  (response) => {
    return response;
  },
  (error) => {
    if (error.response.status === 401) {
      // Handle unauthorized error (e.g., redirect to login page)
    }
    return Promise.reject(error);
  }
);

export const api = {
  //... (other API functions remain the same)
};

export default axiosInstance;
```

Explanation:

- We added a request interceptor using `axiosInstance.interceptors.request.use()`. This interceptor adds an authorization header to all requests. You can replace `'your-access-token'` with your actual access token or any other headers you need to include.
- We also added a response interceptor using `axiosInstance.interceptors.response.use()`. This interceptor allows us to handle common error responses. For example, if the server returns a 401 status code (unauthorized), we can handle it globally, such as redirecting the user to the login page.

Handling Request and Response Transformations

In this step, we'll learn how to handle request and response transformations using Axios interceptors. Request transformations allow us to modify the outgoing request data before it is sent to the server, while response transformations allow us to modify the response data before it is passed to the calling code.

Using Transformations in AxiosInstance

In the `axiosInstance.js` file, we can add interceptors to handle request and response transformations.

Example - axiosInstance.js

```
import axios from 'axios';

const axiosInstance = axios.create({
  baseURL: '<https://jsonplaceholder.typicode.com>', // Replace with your API base URL
  timeout: 5000, // Set a timeout (in milliseconds) for the requests
});

// Request Interceptor: Add a request transformation to all POST requests
axiosInstance.interceptors.request.use(
  (config) => {
    if (config.method === 'post') {
      config.data = {
        ...config.data,
        timestamp: Date.now() // Add a timestamp to the request data
      };
    }
    return config;
  },
  (error) => {
    return Promise.reject(error);
  }
);

// Response Interceptor: Handle response transformation for specific endpoints
axiosInstance.interceptors.response.use(
  (response) => {
    if (response.config.url === '/posts') {
      // Modify the response data for the '/posts' endpoint
      response.data = response.data.map(post => ({
        ...post,
        title: post.title.toUpperCase() // Convert post titles to uppercase
      }));
    }
    return response;
  },
  (error) => {
```

```
    return Promise.reject(error);
  }
);

export const api = {
  //... (other API functions remain the same)
};

export default axiosInstance;
```

Explanation:

- We added a request interceptor using `axiosInstance.interceptors.request.use()`. This interceptor checks if the request method is 'post', and if so, it adds a timestamp to the request data. This is just an example; you can perform any required transformations on the request data.
- We also added a response interceptor using `axiosInstance.interceptors.response.use()`. This interceptor allows us to modify the response data for specific endpoints. In this example, we modify the response data for the '/posts' endpoint by converting post titles to uppercase.

Using the Updated AxiosInstance in Vue Components:

Now, when making API requests with the Axios instance, the transformations added through interceptors will be applied automatically.

Example - PostList.vue:

```
<template>
  <!-- ... (rest of the template remains the same) -->
</template>

<script>
import { api } from '../axiosInstance';

export default {
  // ... (rest of the component options remain the same)
  methods: {
    async getPosts() {
      this.isLoading = true;
      try {
        const response = await api.getPosts();
        this.posts = response.data;
        this.isLoading = false;
      } catch (error) {
        console.error('Error fetching posts:', error);
        this.isLoading = false;
      }
    },
```

```
    async createPost() {
      const newPost = {
        title: 'New Post',
        body: 'This is a new post body'
      };
      try {
        await api.createPost(newPost);
        // Successfully created the post
        this.getPosts(); // Refresh the posts after creation
      } catch (error) {
        console.error('Error creating post:', error);
      }
    }
  }
};
</script>
```

Explanation:

- In the Vue component, we continue to use the `api.getPosts()` function to fetch posts, and we also added a `createPost()` function that uses `api.createPost()` to create a new post with the given data.
- The request transformation added through the request interceptor will automatically add a timestamp to the request data when we make a POST request in the `createPost()` function.
- The response transformation added through the response interceptor will automatically convert post titles to uppercase for the '/posts' endpoint when we fetch posts in the `getPosts()` function.

Note:

- The examples provided in the previous steps and this step use the JSONPlaceholder API (`https://jsonplaceholder.typicode.com`) as a placeholder for the actual API you will use in your project.
- You can add more request and response transformations based on your specific application needs.

By using Axios interceptors for request and response transformations, we can customize data before sending it to the server and modify the server's response before using it in the application. This provides greater flexibility and control over data handling and ensures that the API communication aligns with our application requirements. In the next steps, we'll explore advanced Axios features like request cancellation and progress monitoring.

Advanced Axios Features - Request Cancellation and Progress Monitoring

In this step, we'll explore two advanced features of Axios: request cancellation and progress monitoring. Request cancellation allows us to cancel ongoing API requests, and progress monitoring allows us to track the progress of requests that involve uploading or downloading data.

Request Cancellation with AxiosInstance

In the `axiosInstance.js` file, we can add the ability to cancel requests using Axios' `CancelToken`.

Example - axiosInstance.js

```
import axios from 'axios';

const axiosInstance = axios.create({
  baseURL: '<https://jsonplaceholder.typicode.com>', // Replace with your API base URL
  timeout: 5000, // Set a timeout (in milliseconds) for the requests
});

// Create a CancelToken source
const cancelTokenSource = axios.CancelToken.source();

export const api = {
  // ... (other API functions remain the same)

  // Example of making a request with cancellation
  async getPostWithCancellation(postId) {
    try {
      const response = await axiosInstance.get(`/posts/${postId}`, {
        cancelToken: cancelTokenSource.token // Add cancelToken to the request config
      });
      return response.data;
    } catch (error) {
      if (axios.isCancel(error)) {
        console.log('Request canceled:', error.message);
      } else {
        console.error('Error fetching post:', error);
      }
    }
  },

  // Cancel the ongoing request
  cancelRequest() {
    cancelTokenSource.cancel('Request canceled by user.');
  }
};

export default axiosInstance;
```

Explanation:

- We create a `CancelToken` source using `axios.CancelToken.source()` in the `axiosInstance.js` file. This source is used to generate a cancel token for a request that can be canceled later.
- In the `api` object, we added a new function called `getPostWithCancellation(postId)`. This function makes a GET request to retrieve a specific post, and we pass the `cancelToken` to the request config to enable cancellation.
- To cancel the request, we call `cancelRequest()` and provide a cancel message. When the request is canceled, the promise will be rejected with an object containing the `message` property.

Using Cancellation in a Vue Component

Now, let's use the cancellation feature in a Vue component.

Example - PostDetail.vue:

```
<template>
  <div>
    <h2>{{ post.title }}</h2>
    <p>{{ post.body }}</p>
    <button @click="cancelRequest">Cancel Request</button>
  </div>
</template>

<script>
import { api } from '../axiosInstance';

export default {
  data() {
    return {
      post: null
    };
  },
  async created() {
    const postId = this.$route.params.id;
    this.post = await api.getPostWithCancellation(postId);
  },
  methods: {
    cancelRequest() {
      api.cancelRequest();
    }
  }
};
</script>
```

Explanation:

- In the Vue component, we use `api.getPostWithCancellation(postId)` to fetch a specific post using the ID from the route parameters. The cancellation feature allows us to cancel this request if needed.
- When the component is created, it fetches the post with the given ID using the `getPostWithCancellation()` function.
- The "Cancel Request" button allows the user to cancel the ongoing request when clicked, and the corresponding message will be logged to the console.

Progress Monitoring with AxiosInstance:

Axios also provides the ability to monitor the progress of requests that involve uploading or downloading data. This is especially useful for handling file uploads or downloads.

Example - axiosInstance.js:

```
import axios from 'axios';

const axiosInstance = axios.create({
  baseURL: '<https://jsonplaceholder.typicode.com>', // Replace with your API base URL
  timeout: 5000, // Set a timeout (in milliseconds) for the requests
});

export const api = {
  // ... (other API functions remain the same)

  // Example of making a request with progress monitoring
  async uploadFile(file, onUploadProgress) {
    const formData = new FormData();
    formData.append('file', file);

    try {
      const response = await axiosInstance.post('/upload', formData, {
        onUploadProgress // Add onUploadProgress to the request config
      });
      return response.data;
    } catch (error) {
      console.error('Error uploading file:', error);
    }
  }
};

export default axiosInstance;
```

Explanation:

- In the `api` object, we added a new function called `uploadFile(file, onUploadProgress)`. This function is an example of making a file upload request.
- We use `FormData` to prepare the data to be uploaded.
- The `onUploadProgress` callback is passed to the request config to monitor the upload progress. This callback will be called periodically with progress information as the upload proceeds.

Using Progress Monitoring in a Vue Component

Now, let's use the progress monitoring feature in a Vue component.

Example - FileUpload.vue:

```
<template>
  <div>
    <input type="file" @change="handleFileChange" />
    <button @click="uploadFile">Upload</button>
    <div v-if="uploadProgress !== null">Progress: {{ uploadProgress }}%</div>
  </div>
</template>

<script>
import { api } from '../axiosInstance';

export default {
  data() {
    return {
      file: null,
      uploadProgress: null
    };
  },
  methods: {
    handleFileChange(event) {
      this.file = event.target.files[0];
    },
    async uploadFile() {
      if (!this.file) return;
      const onUploadProgress = (progressEvent) => {
        this.uploadProgress = Math.round((progressEvent.loaded * 100) / progressEvent.
total);
      };
      try {
        await api.uploadFile(this.file, onUploadProgress);
        this.uploadProgress = null; // Reset progress after successful upload
      } catch (error) {
        console.error('Error uploading file:', error);
        this.uploadProgress = null; // Reset progress after error
      }
    }
  }
};
</script>
```

Explanation

- In the Vue component, we use the `uploadFile()` function to initiate the file upload. The `onUploadProgress` callback is provided to monitor the progress of the upload.
- When a file is selected using the input field, the `handleFileChange()` function updates the `file` data property.
- The progress information received through the `onUploadProgress` callback is used to update the `uploadProgress` data property, which is then displayed to the user.

Note

- The examples provided in this step use the JSONPlaceholder API (`https://jsonplaceholder.typicode.com`) as a placeholder for the actual API you will use in your project.
- Progress monitoring is typically more relevant for handling file uploads or large data transfers.

By using request cancellation and progress monitoring, we can enhance the control and user experience of our Vue.js applications. Request cancellation allows us to handle situations where we might need to cancel ongoing requests, and progress monitoring allows us to track the progress of requests that involve uploading

Error Handling and Global Error Handling

In this step, we'll focus on error handling with Axios and implementing global error handling to centralize error handling logic in one place.

Error Handling with AxiosInstance

In the `axiosInstance.js` file, we can handle errors that occur during API requests using Axios' error handling features.

Example - axiosInstance.js

```
import axios from 'axios';

const axiosInstance = axios.create({
  baseURL: '<https://jsonplaceholder.typicode.com>', // Replace with your API base URL
  timeout: 5000, // Set a timeout (in milliseconds) for the requests
});
```

```
// ... (interceptors and other configurations remain the same)

export const api = {
  // ... (other API functions remain the same)

  // Example of error handling in an API function
  async getPost(postId) {
    try {
      const response = await axiosInstance.get(`/posts/${postId}`);
      return response.data;
    } catch (error) {
      if (error.response) {
        // The request was made, but the server responded with an error
        console.error('Server Error:', error.response.data);
      } else if (error.request) {
        // The request was made, but no response was received
        console.error('No Response:', error.request);
      } else {
        // Something happened in setting up the request that triggered an Error
        console.error('Request Error:', error.message);
      }
      throw error; // Re-throw the error to be handled at the component level
    }
  }
};

export default axiosInstance;
```

Explanation

- In the `api` object, we added a new function called `getPost(postId)`, which is an example of fetching a specific post by its ID.
- We used a `try...catch` block to handle any errors that may occur during the API request.
- If `error.response` exists, it means the request was made, but the server responded with an error (e.g., 404 or 500 status code). In this case, we log the server error data to the console.
- If `error.request` exists, it means the request was made, but no response was received (e.g., the server is down). In this case, we log the error request details to the console.
- If neither `error.response` nor `error.request` exists, it means something went wrong in setting up the request (e.g., a network error or incorrect URL). We log the request error message to the console.
- We re-throw the error using `throw error` to allow components to handle the error individually.

Global Error Handling with Vue Error Handler

In the `main.js` file, we can set up a global error handler to catch and handle errors that occur throughout the entire Vue application.

Example - main.js

```
import Vue from 'vue';
import App from './App.vue';
import router from './router';
import { api } from './axiosInstance';

Vue.config.productionTip = false;

// Global Error Handler
Vue.config.errorHandler = (error, vm, info) => {
  console.error('Global Error:', error);
  // You can perform additional actions here, like logging errors to a server or showi
ng error messages to users
};

new Vue({
  router,
  render: (h) => h(App),
}).$mount('#app');
```

Explanation

- In the `main.js` file, we define a global error handler using `Vue.config.errorHandler`.
- The global error handler will be called whenever an unhandled error occurs in any component throughout the application.
- In this example, we log the error to the console using `console.error()`. You can perform additional actions in the error handler, such as logging the error to a server, showing error messages to users, or performing analytics on errors.

Using Error Handling in a Vue Component

Now, let's use the error handling feature in a Vue component.

Example - PostDetail.vue

```
<template>
  <div>
    <h2>{{ post.title }}</h2>
    <p>{{ post.body }}</p>
  </div>
</template>
```

```
<script>
import { api } from '../axiosInstance';

export default {
  data() {
    return {
      post: null
    };
  },
  async created() {
    const postId = this.$route.params.id;
    try {
      this.post = await api.getPost(postId);
    } catch (error) {
      console.error('Error fetching post:', error);
      // Handle the error gracefully (e.g., show an error message to the user)
    }
  }
};
</script>
```

Explanation

- In the `PostDetail.vue` component, we use `api.getPost(postId)` to fetch a specific post by its ID.
- If an error occurs during the API request, the `try...catch` block will catch the error, and we can handle it gracefully. In this example, we log the error to the console, but you can display an error message to the user or take other appropriate actions.

Note

- The examples provided in this step use the JSONPlaceholder API (`https://jsonplaceholder.typicode.com`) as a placeholder for the actual API you will use in your project.
- By implementing global error handling, we can manage and centralize error handling logic for the entire Vue application, providing a more robust and consistent error handling experience.

With error handling in place, we can ensure that our Vue.js application handles errors gracefully, preventing unexpected behaviors and providing valuable feedback to users when something goes wrong during API requests or other operations. In the next steps, we'll explore more advanced Axios features like request retries and authentication handling.

Request Retries and Authentication Handling

In this step, we'll cover two essential features related to handling request retries and implementing authentication with Axios.

Request Retries with AxiosInstance

In the `axiosInstance.js` file, we can configure Axios to automatically retry failed requests a certain number of times before giving up.

Example - axiosInstance.js

```
import axios from 'axios';

const axiosInstance = axios.create({
  baseURL: '<https://jsonplaceholder.typicode.com>', // Replace with your API base URL
  timeout: 5000, // Set a timeout (in milliseconds) for the requests
  retry: {
    retries: 3, // Number of retries for failed requests
    retryDelay: (retryCount) => retryCount * 1000 // Delay between retries (in millise
conds)
  }
});

// Add a request interceptor
axiosInstance.interceptors.request.use(
  (config) => {
    // Attach the cancel token to the request config
    config.cancelToken = cancelTokenSource.token;
    return config;
  },
  (error) => {
    console.error('Request Interceptor Error:', error);
    return Promise.reject(error);
  }
);

// Add a response interceptor
axiosInstance.interceptors.response.use(
  (response) => {
    // Return the response data
    return response.data;
  },
  (error) => {
    // Handle response errors here
    console.error('Response Interceptor Error:', error);

    const { config, response } = error;
    if (response && response.status >= 500) {
      // Retry the request if it's a server error
      config.__retryCount = config.__retryCount || 0;
      if (config.__retryCount >= axiosInstance.defaults.retry.retries) {
        return Promise.reject(error);
      }
      config.__retryCount += 1;
      return new Promise((resolve) => {
```

```
        setTimeout(() => resolve(axiosInstance(config)), axiosInstance.defaults.retry.
retryDelay(config.__retryCount));
      });
    }

    return Promise.reject(error);
  }
);

// ... (other configurations remain the same)

export const api = {
  // ... (API functions remain the same)
};

// Function to cancel ongoing requests
export function cancelRequests() {
  cancelTokenSource.cancel('Request canceled by user');
}

export default axiosInstance;
```

Explanation

- We added a response interceptor using `axiosInstance.interceptors.response.use()`.
- The first argument of `use()` is a callback function that will be called when the response is received successfully.
- The second argument of `use()` is another callback function that will be called if there is an error in the response.
- In this example, we handle response errors in the response interceptor.
- If the response status code is 500 or greater (indicating a server error), we retry the request up to the specified number of times (`axiosInstance.defaults.retry.retries`) with a delay between retries.

Authentication Handling with AxiosInstance

In the context of our Axios instance, we can handle authentication by adding custom headers to our requests.

Example - axiosInstance.js:

```
import axios from 'axios';

const axiosInstance = axios.create({
  baseURL: '<https://jsonplaceholder.typicode.com>', // Replace with your API base URL
  timeout: 5000, // Set a timeout (in milliseconds) for the requests
  retry: {
```

```
    retries: 3, // Number of retries for failed requests
    retryDelay: (retryCount) => retryCount * 1000 // Delay between retries (in millise
conds)
  }
});

// Add a request interceptor
axiosInstance.interceptors.request.use(
  (config) => {
    // Attach the cancel token to the request config
    config.cancelToken = cancelTokenSource.token;

    // Add your authentication logic here (e.g., adding an authentication token to hea
ders)
    const authToken = localStorage.getItem('authToken');
    if (authToken) {
      config.headers['Authorization'] = `Bearer ${authToken}`;
    }

    return config;
  },
  (error) => {
    console.error('Request Interceptor Error:', error);
    return Promise.reject(error);
  }
);

// ... (response interceptor and other configurations remain the same)

export const api = {
  // ... (API functions remain the same)
};

// Function to cancel ongoing requests
export function cancelRequests() {
  cancelTokenSource.cancel('Request canceled by user');
}

export default axiosInstance;
```

Explanation

- We modified the request interceptor to add an `Authorization` header to the requests if there is an authentication token available in local storage.
- You can replace the logic inside the `axiosInstance.interceptors.request.use()` callback to handle your specific authentication mechanism (e.g., token-based authentication or session-based authentication).

Using Request Retries and Authentication in a Vue Component

Now, let's see how we can use request retries and authentication in a Vue component.

Example - UserList.vue

```
<template>
  <div>
    <h2>User List</h2>
    <button @click="loadUsers">Load Users</button>
    <ul v-if="isLoading">
      <li v-for="user in users" :key="user.id">{{ user.name }}</li>
    </ul>
    <p v-else>No users found.</p>
    <button @click="cancelRequest">Cancel Request</button>
  </div>
</template>

<script>
import { api, cancelRequests } from '../axiosInstance';

export default {
  data() {
    return {
      users: [],
      isLoading: false
    };
  },
  methods: {
    async loadUsers() {
      try {
        this.isLoading = true;
        this.users = await api.getUsers();
        this.isLoading = false;
      } catch (error) {
        console.error('Error fetching users:', error);
        this.isLoading = false;
        // Handle the error gracefully (e.g., show an error message to the user)
      }
    },
    cancelRequest() {
      cancelRequests();
      this.isLoading = false;
      this.users = [];
    }
  }
};
</script>
```

Explanation

- The example in the `UserList.vue` component remains the same as in the previous step. We use the `loadUsers()` method to fetch users and the `cancelRequest()` method to cancel the ongoing request.

With the addition of request retries and authentication handling, our Axios instance becomes more robust and secure. Request retries help us handle temporary server

failures gracefully, while authentication handling allows us to add custom headers and manage user authentication effectively. These features contribute to a smoother and more reliable