How to Manage API Request with AXIOS on a React Native App

APIs can make your life a whole lot easier. With an API, you can send requests to other services and get responses without having to build those requests and responses yourself. But building an API isn't as simple as it sounds. It requires careful planning, testing, and debugging. If you're building an API for the first time, it can feel like an impossible mountain to climb. That's where APIs like axios come in. It has a great API and lots of helpful features. Here in this article you'll understand how to use axios to manage API requests in your React Native app.

What is AXIOS?

Axios is one of the easiest HTTP clients to learn and use. Making an API request is as simple as passing a configuration object to Axios or invoking the appropriate method with the necessary arguments. You will learn the basics of Axios in this section.

Configuring Axios

Type following command on terminal window to install Axios:

NPM Install Axios

How to make requests to an API using Axios

Making a call to an API using Axios, you can pass a configuration object to Axios or invoke a method for the corresponding CRUD operations.

For example, you can make a GET request to the /api/users endpoint in one of the following two ways:

```
import axios from 'axios';
constbaseUrl = 'https://reqres.in';
// Passing configuration object to axios
axios({
method: 'get',
url: `${baseUrl}/api/users/1`,
}).then((response) => {
console.log("<<<<<< Passing configuration object to axios >>>>>>", response.data.data);
});

// Invoking get method to perform a GET request
axios.get(`${baseUrl}/api/users/1`).then((response) => {
console.log("<<<<<< Invoking get method to perform a GET request >>>>>>",
response.data.data);
});
```

```
// Passing configuration object to axios
constfetchUserFirst = async () => {
constconfigurationObject = {
method: 'get',
url: `${baseUrl}/api/users/1`,
  };
const response = await axios(configurationObject);
console.log("<<<<<< Fetch User First >>>>>>", response.data.data);
};

// Invoking get method to perform a GET request
constfetchUserSecond = async () => {
consturl = `${baseUrl}/api/users/2`;
const response = await axios.get(url);
console.log("<<<<<< Fetch User Second >>>>>>", response.data.data);
};
```

How to make multiple concurrent API requests using Axios

We can use the **Promise.all** or **Promise.allSettled** method of the Promise API with Axios to make multiple concurrent API requests from a React Native application.

```
constconcurrentRequests = [
axios.get(`${baseUrl}/api/users/1`),
axios.get(`${baseUrl}/api/users/2`),
axios.get(`${baseUrl}/api/users/3`),
];
// Using Promise.all
Promise.all(concurrentRequests)
.then((result) => {
console.log(result);
})
.catch((err) => {
console.log(err);
});
// Using Promise.allSettled
Promise.allSettled(concurrentRequests)
.then((result) => {
console.log(result);
})
.catch((err) => {
console.log(err);
});
```

How to abort network request in Axios

Axios provides functionality for aborting network requests. A typical use case of this feature in React Native is the cancellation of network requests in the use effect hook when a component is unmounted while data is still in flight.

```
useEffect(() => {
const source = axios.CancelToken.source();
consturl = `${baseUrl}/api/users/${userId}`;
constfetchUsers = async () => {
try {
const response = await axios.get(url, { cancelToken: source.token });
console.log(response.data);
    } catch (error) {
if(axios.isCancel(error)){
console.log('Data fetching cancelled');
}else{
      // Handle error
      }
    }
  };
fetchUsers();
return () =>source.cancel("Data fetching cancelled");
 }, [userId]);
```

How to create an instance of Axios

You can also create an instance of Axios with a custom configuration.Axios will merge the configuration object passed while creating the instance with the configuration passed to the instance method:

```
constaxiosInstance = axios.create({ baseURL: 'https://reqres.in/' });
axiosInstance.get('api/users/1').then((response) => {
console.log(response.data);
});
```

How to make GET request using Axios in React Native

Make a GET request to the /api/users endpoint to retrieve a user and store the user ID in state as shown in the code snippet below. You can change the user ID inside the onPress event handler attached to the Load User button. Changing the user ID will trigger a GET request to the API inside the useEffect hook.

After triggering a network request, we display a loading indicator on the screen. If we fetch the data successfully, we update state and remove the loading indicator. If we fail to retrieve the data for some reason, we stop the loading indicator and display an appropriate error message.

We abort the network request in the clean-up function if the user decides to close the app before getting a response from the server. Check the return value of the effect function in the useEffect hook.

Following is the code in the App.js component:

```
import axios from "axios";
import React, { useState, useEffect } from "react";
import {
StyleSheet,
 Text,
ScrollView,
 View,
 Button,
 Image,
 Platform,
} from "react-native";
import Constants from "expo-constants";
constbaseUrl = "https://reqres.in";
function User({ userObject }) {
return (
<View>
<Image
source={{ uri: userObject.avatar }}
style={{ width: 128, height: 128, borderRadius: 64 }}
    />
<Text style={{ textAlign: "center", color: "white" }}>
     {`${userObject.first_name} ${userObject.last_name}`}
</Text>
</View>
 );
}
export default function App() {
```

```
const [userId, setUserId] = useState(1);
const [user, setUser] = useState(null);
const [isLoading, setIsLoading] = useState(false);
const [hasError, setErrorFlag] = useState(false);
constchangeUserIdHandler = () => {
setUserId((userId) => (userId === 3 ? 1 : userId + 1));
  };
useEffect(() => {
const source = axios.CancelToken.source();
consturl = `${baseUrl}/api/users/${userId}`;
constfetchUsers = async () => {
try {
setIsLoading(true);
const response = await axios.get(url, { cancelToken: source.token });
if (response.status === 200) {
setUser(response.data.data);
setIsLoading(false);
return;
      } else {
throw new Error("Failed to fetch users");
      }
    } catch (error) {
if(axios.isCancel(error)){
console.log('Data fetching cancelled');
}else{
setErrorFlag(true);
setIsLoading(false);
      }
    }
  };
fetchUsers();
return () =>source.cancel("Data fetching cancelled");
  }, [userId]);
return (
<ScrollViewcontentContainerStyle={styles.container}>
<View style={styles.wrapperStyle}>
{!isLoading&& !hasError&& user &&<User userObject={user} />}
</View>
<View style={styles.wrapperStyle}>
      {isLoading&&<Text> Loading </Text>}
{!isLoading&&hasError&&<Text> An error has occurred </Text>}
</View>
<View>
<Button
title="Load user"
onPress={changeUserIdHandler}
```

```
disabled={isLoading}
style={styles.buttonStyles}
      />
</View>
</ScrollView>
 );
}
const styles = StyleSheet.create({
container: {
flex: 1,
backgroundColor: "dodgerblue",
alignItems: "center",
justifyContent: "center",
marginTop: Platform.OS === "ios" ? 0 :Constants.statusBarHeight,
 },
wrapperStyle: {
minHeight: 128,
 },
buttonStyles: {
padding: 100,
 },
});
```

How to make a POST request

POST is the HTTP method you use to send data to the server for updating or creating a resource.

Making a POST request in Axios is similar to making a GET request. Most of the time, POST requests are made with user-generated data submitted using a form. Data requires validation on the client side before it is submitted. Two main React packages for managing forms are Formik and React Hook Form.

React Native form for the user's full name and email in the code snippet below. Both TextInput components are controlled components. After clicking the submit button, the TextInput fields and the submit button are disabled before you display a message to show you are creating the resource. Disabling the submit button ensures the user doesn't make multiple submissions. After successfully submitting a POST request, you display a success message to the user:

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

```
 Text,
ScrollView,
 View,
 Button,
 Platform,
 TextInput,
} from "react-native";
import Constants from "expo-constants";
constbaseUrl = "https://reqres.in";
export default function App() {
const [fullName, setFullName] = useState("");
const [email, setEmail] = useState("");
const [isLoading, setIsLoading] = useState(false);
constonChangeNameHandler = (fullName) => {
setFullName(fullName);
 };
constonChangeEmailHandler = (email) => {
setEmail(email);
 };
constonSubmitFormHandler = async (event) => {
if (!fullName.trim() || !email.trim()) {
alert("Name or Email is invalid");
return;
  }
setIsLoading(true);
try {
const response = await axios.post(`${baseUrl}/api/users`, {
fullName,
email,
    });
if (response.status === 201) {
alert(` You have created: ${JSON.stringify(response.data)}`);
setIsLoading(false);
setFullName('');
setEmail('');
    } else {
throw new Error("An error has occurred");
    }
  } catch (error) {
alert("An error has occurred");
setIsLoading(false);
  }
 };
return (
<ScrollViewcontentContainerStyle={styles.container}>
<View>
```

```
<View style={styles.wrapper}>
        {isLoading ? (
<Text style={styles.formHeading}> Creating resource </Text>
        ) : (
<Text style={styles.formHeading}>Create new user</Text>
        )}
</View>
<View style={styles.wrapper}>
<TextInput
placeholder="Full Name"
placeholderTextColor="#ffffff"
style={styles.input}
value={fullName}
editable={!isLoading}
onChangeText={onChangeNameHandler}
        />
</View>
<View style={styles.wrapper}>
<TextInput
placeholder="Email"
placeholderTextColor="#ffffff"
style={styles.input}
value={email}
editable={!isLoading}
onChangeText={onChangeEmailHandler}
        />
</View>
<View>
<Button
title="Submit"
onPress={onSubmitFormHandler}
style={styles.submitButton}
disabled={isLoading}
        />
</View>
</View>
</ScrollView>
  );
}
const styles = StyleSheet.create({
container: {
flex: 1,
backgroundColor: "#252526",
alignItems: "center",
justifyContent: "center",
marginTop: Platform.OS === "ios" ? 0 :Constants.statusBarHeight,
```

```
  },
formHeading: {
color: "#ffffff",
  },
wrapper: {
marginBottom: 10,
  },
input: {
borderWidth: 2,
borderColor: "grey",
minWidth: 200,
textAlignVertical: "center",
paddingLeft: 10,
borderRadius: 20,
color: "#ffffff",
  },
submitButton: {
backgroundColor: "gray",
padding: 100,
  },
});
```

How to make a DELETE request

DELETE requests using Axios the same way you make POST and PUT requests. DELETE request will delete a resource from the server side. You can replace the onSubmitFormHandler of the code for making a POST request with the event handler below to make a DELETE request.

```
constonSubmitFormHandler = async (event) => {
if (!fullName.trim() || !email.trim()) {
alert("Name or Email is invalid");
return;
    }
setIsLoading(true);
try {
const response = await axios.delete(`${baseUrl}/api/users/2`, {
fullName,
email,
      });
if (response.status === 204) {
alert(` You have deleted: ${JSON.stringify(response.data)}`);
setIsLoading(false);
setFullName('');
setEmail('');
      } else {
```

```
throw new Error("Failed to delete resource");
      }
    } catch (error) {
alert("Failed to delete resource");
setIsLoading(false);
    }
  };
```

How to make a PUT request

Updating a resource requires either the PUT or PATCH method. If a resource exists, using the PUT method completely overwrites it, and creates a new resource if it doesn't. PATCH makes partial updates to the resource if it exists and does nothing if it doesn't.

Making a PUT request to an API is similar to making a POST request. The only difference is the configuration object passed to Axios, or the HTTP method needed to invoke to make a PUT request to the API. Replace the onSubmitFormHandler of the POST request with the code below to make a PUT request.

```
constonSubmitFormHandler = (event) => {
if (!fullName.trim() || !email.trim()) {
alert("Name or Email is invalid");
return;
    }
setIsLoading(true);
constconfigurationObject = {
url: `${baseUrl}/api/users/2`,
method: "PUT",
data: { fullName, email },
    };
axios(configurationObject)
      .then((response) => {
if (response.status === 200) {
alert(` You have updated: ${JSON.stringify(response.data)}`);
setIsLoading(false);
setFullName("");
setEmail("");
       } else {
throw new Error("An error has occurred");
       }
      })
      .catch((error) => {
alert("An error has occurred");
```

```
setIsLoading(false);
    });
 };
```

How to handle errors

React-error-boundary (Simple reusable React error boundary component) is a simple reusable component based on React error boundary API that provides a wrapper around your components and automatically catches all errors from the children's components hierarchy, and also provides a great way to recover your component tree.

Create an Errorhandler component like the following code snippet.

```
import * as React from "react";
import { ErrorBoundary } from "react-error-boundary";
import { View, StyleSheet, Button } from "react-native";
import { Text } from "components";
constmyErrorHandler = (error: Error) => {
  // Do something with the error
functionErrorFallback({ resetErrorBoundary }) {
return (
<View style={[styles.container]}>
<View>
<Text>Something went wrong: </Text>
<Button title="try Again" onPress={resetErrorBoundary} />
</View>
</View>
  );
}
exportconstErrorHandler = ({ children }: { children: React.ReactNode }) => (
<ErrorBoundaryFallbackComponent={ErrorFallback} onError={myErrorHandler}>
   {children}
</ErrorBoundary>
);
const styles = StyleSheet.create({
container: {
flex: 1,
flexDirection: "column",
alignItems: "stretch",
justifyContent: "center",
alignContent: "center",
paddingHorizontal: 12,
  },
});
```

Here you can find the sample code in this Github repository

Best Practices for using AXIOS

Global config

Set up a global configuration that handles all application requests using a standard configuration that is set through a default object that ships with axios.

This object contains:

- **baseURL:** A relative URL that acts as a prefix to all requests, and each request can append the URL
- **headers:** Custom headers that can be set based on the requests
- **Timeout:** The point at which the request is aborted, usually measured in milliseconds. The default value is 0, meaning it's not applicable.
- **With Credentials:** Indicates whether or not cross-site Access-Control requests should be made using credentials. The default is false.
- **Response Type:** Indicates the type of data that the server will return, with options including json (default), arraybuffer, document, text, and stream.
- **Response Encoding:** Indicates encoding to use for decoding responses. The default value is utf8.
- **xsrfCookieName:** The name of the cookie to use as a value for XSRF token, the default value is XSRF-TOKEN.
- **xsrfHeaderName:** The name of the HTTP header that carries the XSRF token value. The default value is X-XSRF-TOKEN.
- **maxContentLength:** Defines the max size of the HTTP response content in bytes allowed
- **maxBodyLength:** Defines the max size of the HTTP request content in bytes allowed

Most of the time, only be using baseURL, header, and maybe timeout. The rest of them are less frequently needed as they have smart defaults, but it's nice to know they are there in case you need to fix up requests.

This is the DRYness at work. For each request, we don't have to repeat the baseURL of our API or repeat important headers that we might need on every request.

Custom instance

Setting up a "custom instance" is similar to a global config, but scoped to specified components so that it's still a DRY technique, but with hierarchy.

Set up a custom instance in a new file (**Ex:** authAxios.js) and import it into the "concern" components.

```
// authAxios.js
import axios from 'axios';
constcustomInstance = axios.create ({
baseURL : 'https://axios-app.firebaseio.com'
})
customInstance.defaults.headers.post['Accept'] = 'application/json'
// Or like this...
constcustomInstance = axios.create ({
baseURL : 'https://axios-app.firebaseio.com',
headers: {'Accept': 'application/json'}
})
```

Then import this file into the "concern" components:

```
// form.js component import from our custom instance
import axios from './authAxios';
export default {
methods : {
onSubmit () {
axios.post('/users.json', formData)
    .then(res => console.log(res))
    .catch(error => console.log(error))
  }
 }
}
```

Axios Verbs

Group the Axios HTTP verbs, like GET, POST, DELETE, and PATCH, in the base config file, as below.

```
export function getRequest(URL) {
returnaxiosClient.get(`/${URL}`).then(response => response);
```

```
}
export function postRequest(URL, payload) {
returnaxiosClient.post(`/${URL}`, payload).then(response => response);
}
export function patchRequest(URL, payload) {
returnaxiosClient.patch(`/${URL}`, payload).then(response => response);
}
export function deleteRequest(URL) {
returnaxiosClient.delete(`/${URL}`).then(response => response);
}
```

Now import the custom functions directly wherever needed to make an API request, as in the code below.

```
import { getRequest } from 'axiosClient';
async function fetchUser() {
try {
const user = await getRequest('users');
} catch(error) {
  //Log errors
 }
}
```

Interceptors

- Interceptors helps with cases where the global config or custom instance might be too generic, in the sense that if you set up a header within their objects, it applies to the header of every request within the affected components. Interceptors have the ability to change any object properties on the fly. For instance, we can send a different headerbased on any condition we choose within the interceptor.
- Interceptors can be in the main.js file or a custom instance file. Requests are intercepted after they've been sent out and allow us to change how the response is handled.

```
// Add a request interceptor
axios.interceptors.request.use(function (config) {
 // Do something before request is sent, like we're inserting a timeout for only requests
with a particular baseURL
if (config.baseURL === 'https://axios-app.firebaseio.com/users.json') {
config.timeout = 4000
 } else {
return config
 }
```

```
 console.log (config)
return config;
 }, function (error) {
 // Do something with request error
returnPromise.reject(error);
});
// Add a response interceptor
axios.interceptors.response.use(function (response) {
 // Do something with response data like console.log, change header, or as we did here
just added a conditional behaviour, to change the route or pop up an alert box, based on
the reponse status
if (response.status === 200 || response.status 201) {
router.replace('homepage') }
else {
alert('Unusual behaviour')
 }
console.log(response)
return response;
}, function (error) {
 // Do something with response error
returnPromise.reject(error);
});
```

Conclusion

For most of your HTTP communication needs, Axios provides an easy-to-use API in a compact package.

There are some alternative libraries for HTTP communication, such as ky, a tiny and elegant HTTP client based on window.fetch; superagent, a small, progressive client-side HTTP request library based on XMLHttpRequest.

But Axios is a better solution for applications with a lot of HTTP requests and for those that need good error handling or HTTP interceptions.