Introduction

Gaurav Singhal

- Introduction
- Making Concurrent HTTP Requests
- Consuming Arrays from Axios Response
- Aborting or Cancelling an Axios Request
- Conclusion
- References
- Top ^

109

All You Need to Know About Axios

Gaurav Singhal

Nov 2, 2020 • 15 Min read • 48,875 Views

Nov 2, 2020 • 15 Min read • 48,875 Views

Web Development React

Introduction

Like the Fetch API, Axios is a promise-based HTTP client for making requests to external servers from the browser. If you have worked with jQuery, you may already know about its `$.ajax()` function which has been a popular choice for frontend developers over the native XML HTTP Request (XHR) interface. The Axios library wraps the complex XHR syntax and provides an abstract and declarative way to make requests from the browser as well as in a node environment.

We have already discussed about how we can make HTTP requests using Axios to get data from the server using `axios.get()` function and post data to the server using `axios.post()` function in an earlier guide. In this guide, we will take a deeper route and learn more about other cool features Axios provides.

Making Concurrent HTTP Requests

More often than not, you will come across a scenario where you'll have to make requests from two endpoints for a particular task. Let's consider that you are working with a CMS REST API and you have to get the current user's data as well as the permissions for that user. You'll probably do something like what is shown below:

jsx

```jsx
class User extends Component {
    constructor(props) {
        super(props);
        this.state = {
            user: {
                data: {},
                permissions: {}
            }
        };
    }

    getUserData = async () => {
```

Introduction


```
            try {
                const {data} = await axios.get(`${ROOT_URL}/profile/${this.pr
                return data;
            } catch (err) {
                console.log(err.message);
            }
        }

        getPermissions = async () => {
            try {
                const {data} = await axios.get(`${ROOT_URL}/permissions/${thi
                return data;
            } catch (err) {
                console.log(err.message);
            }
        }

        async componentDidMount() {
            const userData = await this.getUserData();
            const userPermissions = await this.getPermissions();
            this.setState(
                user: {
                    data: userData,
                    permissions: userPermissions
                }
            );
        }

        render() {
            // render the data
        }

    }
```

109

Instead of making the requests twice, we can call the endpoints simultaneously using the `axios.all()` function.

jsx

```jsx
class User extends Component {
    // ...

    getUserData = () => axios.get(`${ROOT_URL}/profile/${this.props.activ

    getPermissions = () => axios.get(`${ROOT_URL}/permissions/${this.prop

    async componentDidMount() {
        try {
            const [userData, userPermissions] = await axios.all([ this.ge
            this.setState(
                user: {
                    data: userData.data,
                    permissions: userPermissions.data
                }
            );
        }
        catch (err) {
            console.log(err.message);
        }

    }

    // ...
}
```

Note that I have used `async-await` syntax to make the code more readable, instead of handling the promises with `.then()` and `atch()`. Having said that, it's really up to you which syntax you efer, both do the same job of handling promises.

`axios.all()` accepts an array of Axios requests, and returns an array that contains the responses from each Axios request. The problem here is that even if one of the requests fails, by default, all the other requests fail as well and the error from the first failed request will be logged in the `catch()` block.

To get around this problem, we can simply return null from the `catch()` block of each Axios request. So, even if one of the requests fail, it will still consider the `axios.all()` promise to be resolved. The return value from the failed request will be `null`, hence we need to have additional checks for if the data in the returned array is `null` or not.

jsx

```jsx
class User extends Component {
    // ...

    getUserData = () => axios.get(`${ROOT_URL}/profile/${this.props.activ

    getPermissions = () => axios.get(`${ROOT_URL}/permissions/${this.prop

    async componentDidMount() {
        try {
```


```
            const [userData, userPermissions] = await axios.all([ this.ge
            this.setState(
                user: {
                    data: userData && userData.data,
                    permissions: userPermissions && userPermissions.data
                }
            );
        }
        catch (err) {
            console.log(err.message);
        }
    }

    // ...
}
```

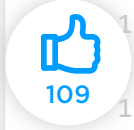
We can further refactor the above code, as follows:

jsx

```
class User extends Component {
    constructor(props) {
        super(props);
        this.state = {
            user: {
                data: {},
                permissions: {}
            }
        };
    }

    async componentDidMount() {
        const URLs = [ `${ROOT_URL}/profile/${this.props.activeUserId}`,
```

Introduction




```
14
15          const requests = URLs.map(URL => axios.get(URL).catch(err => null
16
17          try {
18              const [userData, userPermissions] = await axios.all(requests
19              this.setState(
20                  user: {
21                      data: userData && userData.data,
22                      permissions: userPermissions && userPermissions.data
23                  }
24              );
25          }
26          catch (err) {
27              console.log(err.message);
28          }
29      }
30
31      render() {
32          // render the data
33      }
34
35  }
```

Consuming Arrays from Axios Response

APIs often return an array that contains objects of data. For example, when you want to retrieve all posts from your CMS. To

109

consume the array, we have to loop over the response using the `Array.map()` function which can be done as follows:

jsx

```jsx
class Posts extends Component {
    constructor(props) {
        super(props);
        this.state = { posts: [] }
    }

    async componentDidMount() {
        try {
            const {data} = await axios.get(`${ROOT_URL}/posts`);
            this.setState({
                posts: data
            })
        } catch (err) {
            console.log(err.message)
        }
    }

    render() {
        return (
            <div className="container">
            {  this.state.posts && this.state.posts.length !== 0 ?
                this.state.posts.map(post => <Card title={post.title}>{po
                <Loading/> }
            </div>
        );
    }
}
```

The `Array.map()` function iterates over each element of the array and returns a new array; in our case, the array contains JSX

109

ments, the `<Card />` component. Note that here, I'm using the orthand version, a more complete implementation is as follows:

jsx

```jsx
// ...
render() {
    return (
        <div className="container">
        {  this.state.posts && this.state.posts.length !== 0 ?
            this.state.posts.map((post, index) => {
                    const { title, content } = post;
                    return <Card title={title}>{content}</Card>;
                }) :
            <Loading/> }
        </div>
    );
}
//..
```

For each element in the array, the `map()` function provides us with an anonymous function that accepts two arguments, the item itself and an optional index argument which contains the position value of the current item. We can also do some transformation of data before returning it with JSX. For example we can transform the title to uppercase, as follows:

jsx

```
// ...
        this.state.posts.map((post, index) => {
            const { title, content } = post;
            const transformedTitle = title.toUpperCase();
            return <Card title={transformedTitle}>{content}</Card>;
        })
        //..
```


For conditional rendering in JSX, you cannot use an `if-else` block. Hence, we have used the ternary operators. Also notice that, when we try to run the above code in the browser, we get a warning message in the console which says: **Each child in an array or iterator should have a unique "key" prop.**

The `key` prop is used by React to keep track of items in an array. In the above example, `this.state.posts.map()` will result in an array, hence each JSX element must have a key prop associated with it. Not including a `key` prop will lead to unexpected results and bugs.

In our case, if we do not specify the `key` prop in our `<Card />` component, React wouldn't know how to keep track of the posts and, hence, when the state changes it would re-render the whole array again instead of updating the changes. This is undesirable because it will affect the performance of the application. Therefore,

it's important to have at least one unique key value (in our case, post id) for array item.

109

jsx

```
//...
    this.state.posts.map((post, index) => {
        const { id, title, content } = post;
        return <Card key={id} title={title}>{content}</Card>;
    })
//...
```

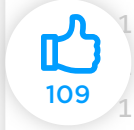
Aborting or Cancelling an Axios Request

You can also cancel or abort a request, if you no longer require the data. To cancel a request, we need to create a cancel token using the `CancelToken.source()` factory method. Let's say that we want to cancel the request when the user navigates from the current page to another page, we can write something as follows:

jsx

```
const NavBar = props => (
    <Nav>
        <NavItem onClick={() => props.navigate('/home') }> Home </NavIter
        <NavItem onClick={() => props.navigate('/about') }> About </NavIt
        <NavItem onClick={() => props.navigate('/contact') }> Contact </N
    </Nav>
)
```

Introduction




```
8
9       class Posts extends Component {
10          constructor(props) {
11              super(props);
12              this.state = { posts: [] }
13          }
14
15          navigate = url => {
16              // cancel the request
17              this.source.cancel('User navigated to different page');
18
19              // assuming we are using React-Router
20              this.props.history.push(url);
21          }
22
23          async componentDidMount() {
24              const CancelToken = axios.CancelToken;
25              // create the source
26              this.source = CancelToken.source();
27              try {
28                  const {data} = await axios.get(`${ROOT_URL}/posts`, {
29                      cancelToken: this.source.token
30                  });
31                  this.setState({
32                      posts: data
33                  })
34              } catch (err) {
35                  // check if the request was cancelled
36                  if(axios.isCancel(err)) {
37                      console.log(err.message);
38                  }
39                  console.log(err.message)
40              }
41          }
42
```

```
        render() {
            return (
                <div className="container">
                <NavBar navigate={this.navigate}/>
                {  this.state.posts && this.state.posts.length !== 0 ?
                    this.state.posts.map(post => <Card key={post.id} title={p
                    <Loading/> }
                </div>
            );
        }
    }
```

109

The `CancelToken.source` factory provides us with two main requirements for cancelling the Axios request, the cancel token and the `cancel()` function. We need to pass the cancel token as a config to the `axios.get()` function and call the `cancel()` function whenever we need to cancel the previous Axios request. We can use the same cancel token for multiple Axios requests.

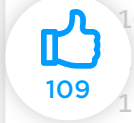
We can also create a cancel token by passing an `executor()` function to the `CancelToken` constructor.

jsx

```
    navigate = url => {
        this.cancelRequest && this.cancelRequest('User navigated to different

        // assuming we are using React-Router
        this.props.history.push(url);
    }
```

Introduction




```
async componentDidMount() {
    const CancelToken = axios.CancelToken;
    try {
        const {data} = await axios.get(`${ROOT_URL}/posts`, {
            cancelToken: new CancelToken(function executor(c) {
                this.cancelRequest = c;
            })
        });
        this.setState({
            posts: data
        })
    } catch (err) {
        if(axios.isCancel(err)) {
            console.log(err.message);
        }
        console.log(err.message);
    }
}
```

Let's check out another use case for cancelling the request. Consider the scenario where we have a search component that retrieves the results from the search API. As the user types in the input field, we need to cancel the previous axios request.

jsx

```
class Search extends Component {
    constructor(props) {
        super(props);
        this.state = { value: null, results: [] };
    }
```

109

```
6
7           search = async () => {
8               const CancelToken = axios.CancelToken;
9               try {
10              const {data} = await axios.get(`${ROOT_URL}/search/q=${this.state
11                  cancelToken: new CancelToken(function executor(c) {
12                      this.cancelRequest = c;
13                  });
14              });
15              } catch (err) {
16                  if(axios.isCancel(thrown)) {
17                      console.log(thrown.message);
18                  }
19                  console.log(err.message)
20              }
21          }
22
23          handleChange = e => {
24              this.cancelRequest && this.cancelRequest();
25              if(e.target.value !== "") {
26                  this.setState({ value: e.target.value }, async () => await th
27              }
28          }
29
30          render() {
31              return <input type="text" onChange={this.handleChange} value={thi
32          }
33      }
```

Here, we are cancelling the request when the value of the input field changes. An important point to note in the above code is that we are passing a callback as a second parameter to the `setState()`

109

function. This is because the state does not change immediately and, hence, to avoid any unexpected outcomes, we are calling the `arch()` function in the callback instead of calling it directly inside `handleChange()` method. In a more real-world use case, the search input would be debounced but, for brevity purposes, I'm not including it here.

Conclusion

In this guide, we looked up various important use cases of Axios and how it's really a great library for making HTTP requests in React (or any other JavaScript Framework like Vue). I hope you had enough of Axios today, and do checkout the Axios documentation that will be linked in the references.

Please follow my other React guides to learn more. If you have any queries, feel free to ask at Codealphabet

References

Axios documentation

Introduction