The Practical Guide to Building LiveChat Apps: LiveChat APIs in practice

Repository overview

Hello! This is Chloma from the LiveChat Developer Program, and welcome back to the Practical Guide to Building LiveChat Apps! In today's video, we're moving straight to the practical part of the course. That's right — today, we are coding!

We prepared a training app for you, which after the previous video, you should already have fetched on your machine. If you haven't done that yet, now's the time to do it.

Naturally, you'll find the link to the LiveChat Tag Master training repository in our resources file.

So, as a result of today's video, you'll know how to implement a working authorization using the redirect authorization method, and also, how to add functional API methods to your apps. Our application's name is Tag Master, which means that our app will allow for extended tag management. However, as an extra, the final application file will also allow for configuring canned responses.

To give you a broader view of what's already inside our app, in the repository, you'll see that the app has already been divided into three separate branches we'll be opening in a certain order. Each of these branches includes a specific part of code indicated by its name.

So some of the coding that's not essential to the topic of this video — like implementing the app's frontend design or certain UI components — has been done off-screen.

Now, let's open the repository and check the first branch called `base`. In here, we have two files - one is the `package.json`, and the second one is a very simple `index.js` file. Let's dive into these and see what they do.

Base branch rundown and initial application run

So the `package.json` file imports the necessary libraries we use in our project. There, you'll find the Accounts SDK library responsible for authorization with the LiveChat API, and the LiveChat Design System, which we'll use components from. We also use Axios, a library for communication with APIs, the Material UI library, which imports different icons, and the styled-components library, which is a library for application styling. Naturally, since this is a React application, we also added relevant React dependencies.

We also modified the command `start` by adding an HTTPS flag to it so we can obtain a secure encrypted HTTPS connection instead of just an HTTP one.

We also have the `index.js` file, which at the moment, doesn't really contain much code. So in here, we imported React and ReactDOM, as well as the LiveChat Design System. We also created a simple JavaScript function that returns a *React JSX* code. And, this function is a React component that we render on the page.

So to begin, we need to install the required packages and dependencies using the `npm install` command in the repository file directory in your terminal. Next, to run the app, simply type `npm start`. And, as soon as we do that, we can check in our browser if our app runs fine.

So, we see that's working! Now, we can move on to set up our application in the Developer Console.

Configuring the app in LiveChat Developer Console

So after creating a new app on our account, we'll have to do some simple configuration that will allow us to code the authorization mechanism in our app, and as a result of that, communicate with the LiveChat API.

For that, we'll need to add building blocks. And the first building block that we'll have add to our app is the *Agent App Widgets*. That's a block for displaying custom embedded widgets in the LiveChat Agent App. Currently, we're using a local environment to run our app, so we'll put the localhost address into the widget source URL. We'll also use the default Chat Details placement for the app.

Our second building block would be the App Authorization. We're coding a JavaScript app, so that's what we select. And inside of this block, we'll have to also whitelist our localhost, so we're allowed to receive the access token there.

The last thing we have to do is to add relevant scopes for the API methods we want to use in our application. Since our app allows for tag and canned responses management, we'll select scopes responsible for creating, reading, and modifying tags. And the same for canned responses.

After we save the changes, we can install the app privately to check whether our local environment displays properly in the *Agent App*. Later, this will also allow for eventual further testing and debugging.

And if you want to know more details about building an app in Developer Console from start to finish, we recommend checking our recent video on this subject. You'll find this video on the LiveChat Insider YouTube channel.

Alright! And once we move on to the *Agent App* to check our sample app, we can see that it looks great — exactly the same as on the localhost. To differentiate our app from other apps we've installed on our test license, we uploaded a custom icon for it off-screen. So, now we can move on to our code editor and continue with coding.

Implementing authorization with LiveChat Accounts SDK

We'll be starting our work on the base branch, and the final code of the authorization part will be available for you on the branch named authorization.

So in this branch, off-screen, we added a simple spinner component to indicate a loading state, as well as a global styling to remove unwanted scrollbar in the app's iframe. And the styling has already been implemented in the app's main file. Naturally, both of these files are available on the authorization branch. We also started our app using the `npm start` command in the terminal.

So the first step to implement LiveChat authorization is to create a configuration file, where we'll store basic information about our authorization. In this file, we'll create a JavaScript object that accepts various key-value pairs. We'll prepare keys for storing our app's `client_id` and the `accounts_url`, which is a URL of the LiveChat Authorization server. You'll find the `client_id` and URL in our documentation. The `client_id` is unique for your app, and you can find it in your Authorization block in the Developer Console.

So once we do that, we want to export the config object so we can use it in other files. When we have that, we can go to the main file. This is where we'll create a new instance of the authorization with the LiveChat Accounts instance.

So, we'll create an arrow function hook that will accept two parameters — our app's `client_id` and `accounts_url`. We want to define a state for this hook, so we'll need to add the `useState` React hook in our imports as well. And the default value of our hook will be a `null`.

Now, we'll also import the LiveChat Accounts SDK to our file so that we can create an instance for the Accounts SDK. The Accounts SDK instance will accept `client_id` and `server_url` as parameters.

For authorization, we need to define a `useEffect` hook. So we'll also need to add the `useEffect` to our React imports. As a parameter for this hook will accept a function and an array. In our `useEffect` hook, we want to define an async authorization function, and we'll use the try-catch construction for it.

So for our authorization, we'll be using the redirect authorization method. To give you a background of how this works, this method redirects our app to the LiveChat Accounts server, where we authorize, then, we go back to the app with the authorization data found in the URL.

And if you want to read more about this authorization method, be sure to have a look at our explainer article, where we dive deeper into authorization methods available in the Accounts SDK.

So as mentioned, the first thing we want to do is to check if the authorization data is present in the URL. We'll use the `async` Accounts SDK method for this, which returns some authorization data for us. And since this is an async method, we'll have to use the await construction. After that, we'll use our predefined AccountsSDK instance, on behalf of which we'll use the redirect authorization method, on which we'll use the `authorize` method.

To verify the validity of our token, we'll use the Accounts SDK `verify` method. So the `verify` function accepts the `authorizeData` from our URL as its parameter. We do this as, for example, our token might be already expired, or else, have the wrong scopes assigned to it — there are a couple of things that could go wrong here, so it's the best practice to check the token's validity.

And, if the authorization data is correct and valid, we can save this to the state of our hook. Here we'll save the access token from the `authorizeData`, and set this in the hook's state.

And since this function can result in errors, we're using the try-catch construction. If the authorization doesn't go right, we'll have to handle an error. For that, we'll redirect the user to the authorization server again with our `auth()` function.

And currently, the authorization function isn't called anywhere outside of here. So we'll need to find a place to call this function. Ultimately, we want to return the accessToken from the `useEffect` hook, and use this hook in our application. Our function accepts config as its parameter, so we'll have to import the config to this file as well. And then, input this config as a parameter.

Let's also have to handle a situation where we don't have an access token yet. So, if the token's not available yet, we'll return the imported spinner component we created off-screen as the loading state. So, if there's no access token, we show a loading spinner.

Now we can check if our authorization works, so let's move to our app. As you can expect, currently, we're authorized, so we don't see any specific error or anything out of the ordinary happening in our browser. But to be 100% sure everything works right, we can consider log our access token and check in the browser's console whether this works as we expected. And, as you can see, everything works correctly!

Coding the LiveChat Configuration API methods to our app

So, now we can go to the real deal and implement code communication with the LiveChat API. For that, we'll open the `apiCalls` branch. In this branch, off-screen, we also implemented some code modifications. Those would be a user interface and components that we'll be using in our further code work.

So the first change is that we extracted the body of our main app to a separate file in the components folder. In this file, we utilized tabs from the LiveChat Design System to separate tags and canned responses that our app will allow to modify.
Since in this video we'll be working just with the tags, we assigned the tags tab to a relevant, separate component. And in the Tags component, we'll find the tags list that's visible in the app. So far, what we have here is just one static tag, but in a moment, we'll show you how to fetch the tags from our LiveChat license using an appropriate API method.

In the Tags component, we also implemented two modals — for adding and removing tags. Both modals are coded to be fired on a button click.

To start working with the LiveChat API, we'll need to update our configuration file. So we'll add a base URL of the LiveChat API. To do that, we need to add a `livechatAPI` key with the LiveChat API URL as the value.

Our next step would be to create an instance for the API, which we'll use to send requests. For that, we'll create an `apiJs` file, in which we'll import the Axios library. As a reminder, it is the Axios library that allows for communication between external APIs. Also, we'll import the config to this file.

So the first thing we want to do here is to make a function for creating HTTP requests. This will be an arrow function, that accepts an HTTP method as its first parameter - for example, `GET`, `POST` and so on. Secondly, it will accept a parameter to specify a route to the API we want to use. As a third parameter, we'll pass the access token for our authorization. And ultimately, as another parameter, we might want to pass the payload — so the body of our request. And we'll call this parameter as data.

So from this function, we want to return the Axios instance. The first element that our Axios configuration will accept is the method that we pass in the function as its argument. Then, we need to specify the URL, and the URL will consist of two elements. The first element is the base URL that we can read from our config file, and that's the livechatAPI address with a specified route to the method we want to use. Next, we want to specify the headers for this request with key-value pairs as the value.

The first pair would be the authorization, which in the case of LiveChat is a Bearer token, followed by the access token. The LiveChat API also requires the `Content-Type` header, so we need to specify this as well. Additionally, we'll pass the accept key with the same `application/json` value as `content-type`. And there's that for the headers. As the last parameter we pass to our HTTP request is our data payload, so the possible body of our request.

Now, we also need to consider that sometimes requests return errors, and we need to handle them as well. So we'll create an arrow function with error catch construction that will print the error in the browser's console.

And that's how our whole function for the HTTP requests would look like. Now we need to create an API instance that would include the requests we want to send. So, we'll create an API object which will accept different functions.

The first function we want to create is a function to fetch tags from the LiveChat Configuration API. So we'd create a function called `fetchTags` that will accept our authorization access token as one of the parameters, and will call the `createApiRequest` method we created just before. As a first element, we'll need to pass the name of the method. To keep the React-like approach, we'll define a `POST` const for this. Then, we can send this const as the first parameter in our function. The next parameter will be the route to our API directory. The next thing is to pass the access token, and in the case of this exact method we're using for fetching tags, the payload will be an empty object. And, the last thing we need to do is to export this API instance.

Now, we want to go to the Tags component. Here, we'll call the method we just created. And the first thing we want to do is to import our `apiJs` file from the utils folder.

Next, we want to define the method that will fetch the tags from our LiveChat license. The access token is one of the props of this component, and after fetching the tags, we want to save those in the state of our component too.

To use this method, we need to use the `useEffect` function. As a first parameter, this function will accept another arrow function, which we'll call updateTags. As a second parameter, it will accept the dependency array, which in our case, is empty.

Previously, as an initial parameter for the tags view, we mentioned that we display some mockup data. We don't need this anymore, as we implemented a dedicated function to fetch them directly from the Configuration API. So now, we can initialize our tags with an empty `null`.

And, now's the high time to check in the Agent App if all we did even works! And, as we can see, there are definitely some tags fetched from the API, so it means it works well. Also, the exact same data is visible on the localhost. But to be 100% sure, we can check in our browser's network if the tags are fetched correctly.

So since this works fine, we can move on to creating other requests our app will utilize. So we'll create another HTTP request — now to create tags.
This function will accept two parameters - a tag and an access token. We'll also use the `createApiRequest` that we defined prior. Same as before, we'll use our method, a `POST` const, a route to the API directory, the access token, and the payload object, which will accept the `name` key and tag name we want to pass as its value.

So now we'll use this new method in our main Tags component file. For that, we'll have to import the API object here as well. We'll implement this method on the onSubmit function, which will call the HTTP request to the LiveChat Configuration API.
So the first thing we want to do is to apply the event prevent default, and then, create a loading state for this component. Next, we'll call the `createTag` API method we created. As a tag, it accepts the state of our tag component, and the access token has already been passed as a props.

So after we add a new tag, we'll want to update the state of our app with the newly fetched tags. This way, we can make sure that the state of our app will be the same as the backend data. And for that, we'll need to call the updateTags function.

So what we have to do now is to call the update function inside of our component — here. As props, we want to pass the `updateTags` function we created. So after sending a request to create a new tag, we want to call the update function.
Ultimately, we want to reset the state of our app. For that, we'll use an arrow function, that will both close the modal for adding tags, stop the loading state, and reset the tags we've been creating before.

Alright, so let's test that and try to create some new sample tag. It looks like everything's working correctly as well!

So, the last HTTP request we'll create in our API file today is a request to remove tags. And we'll call this request `removeTag`. Similarly to the `createTag` function, the `removeTag` will accept the tag we want to remove as a parameter and the access token. So as you can see, this function has pretty much the same pattern as the `createTag` function.
For this one, we also need to use the `createApiRequest` method. Then, we'll need to define the method, which is again a `POST` const, the route to the API, the access token, and lastly, the name of the tag we want to delete in the request's payload.

To put this method to use, in our Tags component, we'll define a tag removal mechanism on the `onDelete` event. So the first thing we'll do is to set the state of our application, which will accept the previous values, and the tag to remove as its parameters. Thanks to that, we'll know which tags we want to remove — as we might want to remove more than one tag at a time.

Next, we'll start the loading state and utilize our imported API to remove the tag. Here, we'll pass the tag we want to remove, as well as the access token. After removing a tag, we want to call the update function again to make sure the tags are up to date.
And ultimately, we also want to reset the state of our component with an update function. Inside of this function, we'll set the tag to remove as a `null` and stop the loading state, just as we did previously with the createTag function.

So let's check if this works in our application on the localhost and try to remove a tag. It appears that it works just fine! Now we can test how it works in the Agent App as well. We'll try to add a new tag. Good, works well! Now we'll see if our app removes the tag without any errors. Seems to be all good, great!

Outro

Oof, that was something! Now, we have a fully working and running application. If you want to update your code to the most recent state with canned responses methods implemented as well, you can simply switch to the master branch where it's all available for you to check.

In the next video, we'll teach you how to deploy this application to an external hosting. We'll explain how to use three different solutions - Firebase, Netlify, and Heroku. You'll also receive a great insight into what hosting you should choose for your app, depending on what you're particularly interested in building.

So, thanks for sticking with us, and see you in the next one!