[[](https://delivery.smashing.services/ball?uri=//srv.buysellads.com/ads/click/x/GTND42QNF6YIL53LCKY4YKQMCYAIKKQJCTSI5Z3JCWSDL2JEC67DVKQKC6BIK5QLCV7DTK3EHJNCLSIZ)](https://delivery.smashing.services/ball?uri=//srv.buysellads.com/ads/click/x/GTND42QNF6YIL53LCKY4YKQMCYAIKKQJCTSI5Z3JCWSDL2JEC67DVKQKC6BIK5QLCV7DTK3EHJNCLSIZ)

[Slack](https://delivery.smashing.services/ball?uri=//srv.buysellads.com/ads/click/x/GTND42QNF6YIL53LCKY4YKQMCYAIKKQJCTSI5Z3JCWSDL2JEC67DVKQKC6BIK5QLCV7DTK3EHJNCLSIZ)

[Bring your team together with Slack, the collaboration hub for work.](https://delivery.smashing.services/ball?uri=//srv.buysellads.com/ads/click/x/GTND42QNF6YIL53LCKY4YKQMCYAIKKQJCTSI5Z3JCWSDL2JEC67DVKQKC6BIK5QLCV7DTK3EHJNCLSIZ)

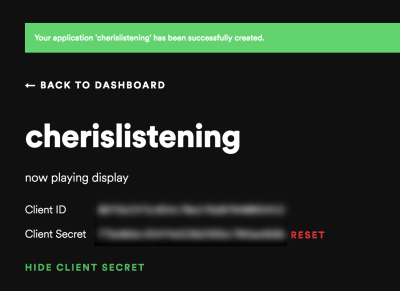
[Learn More](https://delivery.smashing.services/ball?uri=//srv.buysellads.com/ads/click/x/GTND42QNF6YIL53LCKY4YKQMCYAIKKQJCTSI5Z3JCWSDL2JEC67DVKQKC6BIK5QLCV7DTK3EHJNCLSIZ)

**Sign Up For A Spotify Developer Account**

This requires a Spotify account. Make note that every use of Spotify’s API must adhere to their [brand guidelines](https://developer.spotify.com/branding-guidelines/).

Create a Client ID at <https://developer.spotify.com/dashboard/applications>.

Take the **Client ID** and the **Client Secret**, which you can find if you click on the green card into your new application’s details, and export them to Heroku as configuration variables. **Keep these safe and secret!** If you believe your client secret has been exposed, you can get a new one, but you’ll need to update your application’s configuration as well.

[](https://cloud.netlifyusercontent.com/assets/344dbf88-fdf9-42bb-adb4-46f01eedd629/b5d7ddf4-c47d-462d-b822-edcf180bdd59/never-share-client-id-secret-tokens.png)Never share your client ID and client secret tokens! ([Large preview](https://cloud.netlifyusercontent.com/assets/344dbf88-fdf9-42bb-adb4-46f01eedd629/b5d7ddf4-c47d-462d-b822-edcf180bdd59/never-share-client-id-secret-tokens.png))

$ heroku config:set CLIENT\_ID=<CLIENT\_ID> CLIENT\_SECRET=<CLIENT\_SECRET>

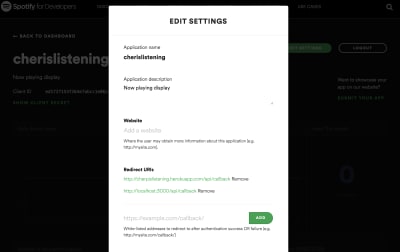
Setting CLIENT\_ID, CLIENT\_SECRET and restarting ⬢ cherislistening... done, v3

CLIENT\_ID: <CLIENT\_ID>

CLIENT\_SECRET: <CLIENT\_SECRET>

Copy

On the top right side of the application dashboard, there is a Settings button. Click that and add in two callback URLs for whitelisting. You’ll need a local callback URL and one for your production server (the Heroku URL we got during setup).

[](https://cloud.netlifyusercontent.com/assets/344dbf88-fdf9-42bb-adb4-46f01eedd629/38924ece-c565-4358-b2d0-57bfe6151320/spotify-settings-whitelisted-callback-url.png)Whitelisted callback URLs in Spotify’s dashboard ([Large preview](https://cloud.netlifyusercontent.com/assets/344dbf88-fdf9-42bb-adb4-46f01eedd629/38924ece-c565-4358-b2d0-57bfe6151320/spotify-settings-whitelisted-callback-url.png))

Spotify has fantastic [developer documentation](https://developer.spotify.com/documentation/web-api/), including a great [reference interface](https://developer.spotify.com/documentation/web-api/reference/) for testing endpoints. We’ll need to get our user ID to save to our configuration variables, so let’s do that with [Get Current User’s Profile](https://developer.spotify.com/console/get-current-user/). Get an auth token from their console, selecting the user-read-private scope. Click “Try It”, and in the right-hand column look for your ID. We’ll use this identifier to make sure no one else can sign into our app.

$ heroku config:set SPOTIFY\_USER\_ID=<SPOTIFY\_USER\_ID>

Setting SPOTIFY\_USER\_ID and restarting ⬢ cherislistening... done, v4

SPOTIFY\_USER\_ID: <SPOTIFY\_USER\_ID>

Copy

As we discussed, we will have data we wouldn’t want exposed to the public. Two of these are clientId and clientSecret we were given by Spotify, and another which Heroku exported for us to access our Redis cache on the server. We’ll need to grab those for our local development as well.

$ heroku config

=== cherislistening Config Vars

API\_URL: /

CLIENT\_URL: http://cherislistening.herokuapp.com

HOST: 0.0.0.0

NODE\_ENV: production

NPM\_CONFIG\_PRODUCTION: false

REDIS\_URL: <REDIS\_URL>

SPOTIFY\_CLIENT\_ID: <SPOTIFY\_CLIENT\_ID>

SPOTIFY\_CLIENT\_SECRET: <SPOTIFY\_CLIENT\_SECRET>

SPOTIFY\_USER\_ID: <SPOTIFY\_USER\_ID>

$ touch .env

Copy

We’ll transfer the credentials Heroku returned in our terminal to our new file, *.env*, and we’ll make our client URL our local server, http://localhost:3000/. We’ll need to make our Redis URL point to our local instance, too, which by default is redis://127.0.0.1:6379. This file will be ignored by git.

CLIENT\_URL=http://localhost:3000/

REDIS\_URL=redis://127.0.0.1:6379

SPOTIFY\_CLIENT\_ID=<SPOTIFY\_CLIENT\_ID>

SPOTIFY\_CLIENT\_SECRET=<SPOTIFY\_CLIENT\_SECRET>

SPOTIFY\_USER\_ID=<SPOTIFY\_USER\_ID>

Copy

*.env*

In order to access the configuration on our local server, we’ll need to update the nuxt config. We’ll add another item to our modules array: @nuxtjs/dotenv. We’ll also need to import two of the variables we will need available on the client-side of our application. We’ll add an env object following modules.

/\*

\*\* Nuxt.js modules

\*/

modules: [

// Doc: https://axios.nuxtjs.org/usage

'@nuxtjs/axios',

'@nuxtjs/dotenv'

],

env: {

spotifyId: process.env.SPOTIFY\_CLIENT\_ID,

clientUrl: process.env.CLIENT\_URL

}

Copy

*nuxt.config.js*

**Building Our API Layer**

**Middleware**

Nuxt has two separate methods for executing server-side code.

In a single-file component (SFC), you have access to the [middleware property](https://nuxtjs.org/api/pages-middleware#the-middleware-property), which corresponds with the middleware folder in your scaffolding. The drawback with this middleware for our use-case is that while it will execute server-side when your page is loaded or refreshed, it will execute client-side once your app is mounted, and when you navigate with nuxt’s routes.

The other option is what we’re looking for. We’ll create our own directory and add it as [serverMiddleware](https://nuxtjs.org/api/configuration-servermiddleware/) to our config. Nuxt creates its own express instance, so we can write middleware registered to its stack that will only run on the server. This way, we can protect our private data from exploitation. Let’s add an api folder and *index.js* to handle our API endpoints.

$ mkdir api

$ touch api/index.js

Copy

Next, we’ll need to add our directory to our config so it registers when we start our server. Let’s open up the file *nuxt.config.js* at the root of our app. This file gives us our HTML <head>, as well as connecting anything to our client at build time. You can read more about the config in the [docs](https://nuxtjs.org/guide/configuration/).

We’ll add our api directory to our config file,

},

serverMiddleware: ['~/api']

}

Copy

*nuxt.config.js*

While we are developing, our changes will require rebuilds and server reboots. Since we don’t want to have to do this manually, nuxt installs nodemon for us, which is a “hot reload” tool. This just means it will reboot the server and rebuild our app when we save our changes.

Since we’ve added our API as serverMiddleware to Nuxt’s, we’ll need to add our directory to the config. We’ll add watch to our build object, and add the relative path from root.

\*/\*\*

\*\*\* Build configuration\*

\*\*/\*

build: {

watch: ['api'],

\*/\*\*

\*\*\* You can extend webpack config here\*

\*\*/\*

extend(config, ctx) {},

serverMiddleware: ['~/api']

}

Copy

*nuxt.config.js*

We’ll also need to change our dev script in *package.json* to restart the server. We’ll need to make it nodemon --watch api --exec \"nuxt\":

"scripts": {

"dev": "nodemon --watch api --exec \"nuxt\"",

"build": "nuxt build",

"start": "nuxt start",

"generate": "nuxt generate",

"heroku-postbuild": "npm run build"

},

Copy

*package.json*

Now we don’t have to worry about rebooting and restarting our server manually every time we make a change. 🎉

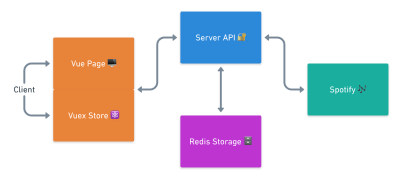
Let’s start our local development server.

$ npm run dev

Copy

**Data Flow, Storage, And Security**

Before we start writing our API layer, we’ll want to plan how we move data from external sources to our client. We’ve set up a Redis cache server, signed up for Spotify API, and set up a structure that has a client layer and a server layer. The client has pages and a store where we can store and render our data. How do these work together to keep our authentication data safe and drive our Now Playing component?

[](https://cloud.netlifyusercontent.com/assets/344dbf88-fdf9-42bb-adb4-46f01eedd629/9a9d5b3b-e460-4cea-8de1-190e9a950835/data-flow-plan-in-a-chart.png)Data flow plan in a chart ([Large preview](https://cloud.netlifyusercontent.com/assets/344dbf88-fdf9-42bb-adb4-46f01eedd629/9a9d5b3b-e460-4cea-8de1-190e9a950835/data-flow-plan-in-a-chart.png))

Any information we want to keep long-term, or for new incoming connections, we’ll want to store on the server. We can’t log into Spotify when other users visit our app, so we’ll need want to ensure that new client connections can bypass authentication by accessing our special service token. We’ll want to keep track of our own Spotify login so that only our own connection is approved by the API, and we’ll want a track ready to show in case we can’t connect to Spotify’s API for some reason.

So, we’ll need to plan on storing our Spotify refresh\_token, our Spotify userId, and our lastPlayedTrack in our Redis Cache.

Everything else can safely be stored in our client’s vuex store. The store and the pages (including their components) will pass data back and forth using nuxt’s architecture, and we’ll talk to the Redis cache and Spotify’s API via our own server’s API.