How To Combine REST API calls with JavaScript Promises in node.js or OpenWhisk

A familiar scenario for node.js developers: you want to make an HTTP request to a REST API, and depending on the response of the first request, you need to make more requests.

Lars Trieloff

Mar 27, 2017 · 5 min read

I came across a scenario like this just the other day when I read Stephen O'Grady's blog post on the state of Open Source licensing.

> *the unfortunate reality is that increasingly open source repositories are populated not by code carrying one of the aforementioned types of licenses, but no license at all.*

Thinking of my own GitHub repositories, I wondered how many would not have a license, and if there was an easy way to find out. I decided against the obvious solution and set out to write some code to find the answer.

The traditional approach would have been to simply make a request, handle the response in a callback function, make another request, handle the response in a nested callback function, and keep on nesting callbacks as deeply as necessity requires or sanity permits.

With my exploration of OpenWhisk, I wanted to see if there was a more elegant way to solve this problem. And there is: the request-promise library (a variation of the popular request HTTP client library for node.js).

Getting Started with request-promise

To start: note that request-promise requires the Bluebird promise implementation and the original request library. If you don't already have them, install using npm.

```
npm install request request-promise
```

With the dependencies installed, you can make your first request.

All this does is set up a `GET` request to the GitHub API, without any authorization, and very little handling of the response. The interesting thing here is `then`, which is the main interface of the Promises API.

Next, add code so that you have an easier time running and testing the code. (You should also go to your GitHub user settings and acquire a personal access token).

What is happening here is that we define a `github` object that serves as a holder for our token. The token gets passed as a command line argument into `main`, where most of the action will be. You will also notice the added `getUser` function, which is using authorization to get the user profile of the currently logged in user as a JSON object. You can call this for testing from the command line like this:

```
$ node authenticated-request.js <your-token>
{
  "login": "trieloff",
  "id": 39613,
  "avatar_url": "https://avatars0.githubusercontent.com/u/39613?v=3",
  "gravatar_id": "",
  "url": "https://api.github.com/users/trieloff",
  "html_url": "https://github.com/trieloff",
  "repos_url": "https://api.github.com/users/trieloff/repos"
…
```

At this point, all we do with the result is to log it. We are just getting started.

Chaining Processing with Promises

This leads us to the first pattern: **if you want to make one request, and then process the result using the response of the first request as input, just use `then`.**

To chain the requests, we define a new function `getUserReposUrl`, which is using the `repos_url` key on the passed JSON object.

In the next step, we pull the actual list of repositories by making a second HTTP request. To make things a bit more useful for developers that have lots of repositories on GitHub, the GitHub API uses pagination. This allows us to try the next pattern right away:

Recursive Request Chaining with Promises

Recursive request chaining means nothing more than that we call a function that makes a request, which might call itself to make another request, and so on. All that's being returned are either promises or data, once the promise has been fulfilled and the request has been made.

The `getUserRepos` function that we will call next looks relatively familiar, but at line 13, a bit more complicated response handling starts. The first thing it does is to check if a list of retrieved `repos` has been passed to it, and if not, it will initialize that list.

Next, it goes on to check if the `Link` header of the response (note we used `resolveWithFullResponse` to get the headers, not just the body) indicates that there are more pages to load, and triggers the `getUserRepos` function, i.e. itself, once again.

This time, `repos` gets passed, and will be built up with subsequent recursions. As a result, we end up with a full list of repositories for the current user. Next, we refine that

list.

Mapping & Filtering Responses with Promises

Now that we have the full list returned, we can run `map` and `filter` on the returned array of repositories. These two filter calls will weed out all private repositories (only I care about the license for these) and forked repositories (it's not my place to assign them a license).

`map` will call the function `licenseUrl` for each remaining repository and returns the URI for the license file, should it exist.

In the next step, we **check existence for each URL in the resulting array, which, again, can be done using `map`.**

Fanning out Requests with Promises

In this (almost final) step, we replace the returned URL with more useful info: whether a `GET` request to this URL yields an error or not.

Just like we used `map` before to transform data, we will do it again, except in this case, the function `checkLicense` performs a request and returns a promise.

The only slightly counter-intuitive aspect about `checkLicense` **is that it returns `false` when the request succeeds**. As our goal is to find repositories that have no license (where the request will yield a status code 404), the function to handle the rejected promise is slightly more interesting. It returns the URL of the missing license file.

Final Clean-Up

In the final steps, we `filter` (using `isMissing`) all repositories that have a license out and `map` (using `createLicenseLink`) the API URL for the license with a convenient link to the GitHub web interface that allows you to pick a license for the repository.

Altogether, it took no more than 100 lines of code to get a full list of public repositories that are missing a license. Not only did it take a relatively trivial amount of time to develop this code (scarcely more than checking manually), but we've also built (without noticing) a serverless function that is ready to run on OpenWhisk.

Here's what returned for my GitHub account:

```
node get-repos.js a4abc39daefa2a26b10cdefabcdefa5bac6defab
30 repos so far
There is more.
37 repos so far
[ 'https://github.com/trieloff/creative-lights/new/master?filename=LICENSE',
  'https://github.com/trieloff/excelsior/new/master?filename=LICENSE',
  'https://github.com/trieloff/katacoda-scenarios/new/master?filename=LICENSE',
  'https://github.com/trieloff/medium-feedreader/new/master?filename=LICENSE',
  'https://github.com/trieloff/medium-tools/new/master?filename=LICENSE',
  'https://github.com/trieloff/openlike/new/master?filename=LICENSE',
  'https://github.com/trieloff/openwhisk-cljs/new/master?filename=LICENSE',
  'https://github.com/trieloff/stackoverflow-bigqueries/new/master?filename=LICENSE' ]
```

By the time you read this, the repositories above should all have their missing LICENSE files.

Give the code a try on your own account!

JavaScript Nodejs Promises Openwhisk Github

WRITTEN BY

Lars Trieloff

Principal at Adobe

Follow

Adobe Tech Blog

News, updates, and thoughts related to Adobe, developers, and technology.

Follow

See responses (4)