Example project: Scraping XML sitemaps

> The code in this guide scrapes an artificial sitemap for a non-existent site, that was created specifically for this article. When scraping anything on a site that you do not run yourself, you should always check the robots.txt to determine whether the owner is okay with you scraping it, and avoid collecting personal information. There are some rare cases where it makes sense to ignore robots.txt, but especially if you are doing something commercial, your case almost certainly isn't one of them. **Permission matters!**

In this Promistreams tutorial, we'll build a small sitemap scraper that can recursively follow XML sitemaps, and collect every page it finds along the way, printing the URLs of all the pages to the terminal. This kind of code might come in handy when building a search engine, for example.

The sitemaps that we want to scrape start at https://fake-sitemap.slightly.tech/. These sitemaps were created specifically for this guide, and you can freely experiment with scraping them, just try not to break the server. They do not point to any valid page URLs, but all the other sitemap files referenced in them are valid.

What you will need

- A Node.js version that is still being updated
- A JS package manager such as pnpm or npm (we will use pnpm in this guide, but npm will work too)
- A code editor of some kind, whichever one you prefer working in
- A working internet connection
- The ability to install native Node modules that require compilation (this usually means having your operating system's build tools installed)
- A Linux-style terminal (WSL should work too)
- A few dependencies from npm, but these will be installed throughout the guide.

> **This guide will provide example code for the project.** While you *could* just copy-paste the code into your file, I recommend typing it out manually instead. In years of tutoring developers, I've found that this really helps people to remember how things work, and it makes it easier to apply the ideas to your own projects later on.

>

> **Are you using npm instead?** Then the `pnpm add` command will be `npm install` instead, and the `pnpm init` command will be `npm init`.

Creating the project

In your terminal, create a new folder for your project, move into it, and initialize the project:

```
mkdir sitemap-scraper
cd sitemap-scraper
pnpm init
```

Making a HTTP request

To scrape these sitemaps, we will need to make a couple of HTTP requests. First of all, let's make a single one to check that everything works. First of all, install the `bhttp` dependency that we will need - it is a HTTP client, and will handle the HTTP request for us:

```
pnpm add bhttp
```

> When running `pnpm add` or `npm install` commands, always make sure that your terminal is currently inside of the project directory, otherwise the package will be installed in the wrong place.

Then, create a new file in your project folder (eg. `index.js`), and write the following code in it:

```
let bhttp = require("bhttp");
let assert = require("node:assert");

(async () => { // async IIFE
  let response = await bhttp.get("https://fake-sitemap.slightly.tech/sitemap.xml");
  assert(response.statusCode === 200);

  console.log(response.body.toString());
})();
```

All this does is make an asynchronous HTTP request, wait for it to complete, verify that it responded with a HTTP 200 status code (throwing an error if not), and then read the response body as a string and print that to the terminal. Try running this code - if it's working correctly, you should see some XML appear on your terminal.

>

> **Why is there a weird async function around the code?** This is the asynchronous equivalent of an IIFE, an Immediately Invoked Function Expression. It defines an asynchronous arrow function, and immediately calls it. This allows us to use `async`/`await` in what would otherwise be top-level code, which is not possible in all JS environments.

> **What does XML look like?** If you're not familiar with XML, it looks something like `<foo>bar</foo>`, ie. text and tags enclosed in angle brackets. The precise structure can vary from file to file - XML is just a data encoding, it doesn't define *what kind* of data an XML document can contain, and we don't need to care about the structure for the purpose of this guide.

Making a queue

Okay, so we've made one HTTP request, and that works. However, there's a peculiarity about XML sitemaps: *they can be recursive*. It's possible for one XML sitemap file to not just list page URLs, but also *other* sitemap files. To correctly discover all of the page URLs on a site through its sitemap, you will often have to follow these references and scrape those referenced sitemap files too!

An easy way to solve this problem is to work with a **queue**. A queue does exactly what it sounds like; it keeps a list of all the things that need to be done (sitemap files that need to be scraped, in this case), and works through them one by one. In JS, the simplest possible implementation of a queue is an array. The queue will be forgotten once you exit the process, but because there are usually only a few sitemap files and then the task is done, that is okay here.

So, let's add a queue to our code:

```
let bhttp = require("bhttp");
let assert = require("node:assert");

(async () => { // async IIFE
  let queue = [];
  queue.push("https://fake-sitemap.slightly.tech/sitemap.xml");

  let response = await bhttp.get(queue[0]);
  assert(response.statusCode === 200);

  console.log(response.body.toString());
})();
```

Wait, that's it? Yep! Well, sort of. Our queue isn't really doing any queue-things yet - it's still making a single request and then exiting, but that request is "in a queue" now. So, let's make the

queue actually *useful*:

```
let bhttp = require("bhttp");
let assert = require("node:assert");

(async () => { // async IIFE
  let queue = [];
  queue.push("https://fake-sitemap.slightly.tech/sitemap.xml");

  async function doTask() {
    if (queue.length > 0) {
      let response = await bhttp.get(queue.shift());
      assert(response.statusCode === 200);

      console.log(response.body.toString());
      return doTask();
    } else {
      // Do nothing, process will exit by itself.
    }
  }

  doTask();
})();
```

We've now added *recursion* to our request code. Take your time, give it a careful read, and try to follow the flow of the code. At the end of the IIFE, `doTask` is called. If there's at least one item in the queue, it takes it from the queue (note how it now uses `.shift()` instead of `[0]`!), and uses it as the URL to fetch. Then after it has logged the response, it *calls itself* again, to process the next item in the queue, if any. If there are no items left, it will do nothing, `doTask` will return undefined, and the stack of recursive function calls will unravel to nothing, causing the process to eventually exit.

So now a queue has been added, and we're taking URLs from the queue to fetch, but we are *still* only making a single request and then exiting. We're not actually using that recursive logic for anything useful yet. For the next step, we're going to have to actually *parse* the contents of the sitemap file, not just print it to the terminal, and that is where Promistreams come in.

Parsing sitemap files

So, what *is* a sitemap file actually? In concept, it's really simple - it's just a list of other URLs, sometimes sitemaps and sometimes webpages, with a bunch of optional extra metadata to know what kind of content is going to be on the page. Sitemaps are mainly used by search engines to

know which pages it needs to look at, and add to its search index.

Sitemaps can take a few forms, but the typical machine-readable form (which is the one we're interested in here!) is in XML, and it's human-readable too - in fact, you can just look at the file that we're going to start scraping from by clicking that link. Your browser will most likely show you a bunch of XML.

First of all, we need to install a few packages:

```
pnpm add @promistream/pipe @promistream/from-node-stream @promistream/decode-string
@promistream/parse-sitemap @promistream/simple-sink
```

Whew, that's a lot of packages at once! Let's have a look at what each package does for us:

- `@promistream/pipe`: This is probably the most important Promistream package; it gives you the `pipe` function, which you use to pipe together different streams into a *pipeline*. It's what makes Promistreams work!
- `@promistream/from-node-stream`: This package will let you convert a Node.js stream into a Promistream automatically. We need this because bhttp will only give us a Node stream, not a Promistream.
- `@promistream/decode-string`: Remember how in the very first version of our code, we called `.toString()` on the response body? This does the same thing, but for a stream of Buffers (which is what we will get from bhttp). It converts raw binary data into readable text according to some sort of *text encoding*, UTF-8 in our case. We have to do that because the sitemap parser doesn't accept Buffers, only strings.
- `@promistream/parse-sitemap`: This is the central point of this example project; it's the stream that actually parses the sitemap contents, turning it from a stream of XML (strings) into a stream of objects with sitemap/page URLs.
- `@promistream/simple-sink`: Finally, this is what 'drives' the whole pipeline. A sink will keep requesting values and process them with some custom callback, forever. The custom callback is where we do something *with* the objects we got from `@promistream/parse-sitemap`.

> **About Buffers and strings:** In Node streams, it is common for many streams to accept either strings or Buffers, and convert between the two automatically. In Promistreams, this is not the case - because that kind of automatic conversion can introduce subtle bugs, it is avoided in stream implementations and you are expected to do the conversation yourself using an encoding or decoding stream. A Promistream package will typically accept either Buffers *or* strings - read the documentation of a stream package carefully to learn what sort of values it expects.

Then, let's add the sitemap parsing to our code!

```
let bhttp = require("bhttp");
let assert = require("node:assert");

const pipe = require("@promistream/pipe");
const fromNodeStream = require("@promistream/from-node-stream");
const decodeString = require("@promistream/decode-string");
const parseSitemap = require("@promistream/parse-sitemap");
const simpleSink = require("@promistream/simple-sink");

(async () => { // async IIFE
  let queue = [];
  queue.push("https://fake-sitemap.slightly.tech/sitemap.xml");

  async function doTask() {
    if (queue.length > 0) {
      let response = await bhttp.get(queue.shift(), { stream: true });
      assert(response.statusCode === 200);

      await pipe([
        fromNodeStream.fromReadable(response),
        decodeString("utf8"),
        parseSitemap(),
        simpleSink((item) => {
          if (item.type === "sitemap") {
            queue.push(item.url);
          } else if (item.type === "url") {
            console.log("URL found:", item.url);
          }
        })
      ]).read();

      return doTask();
    } else {
      // Do nothing, process will exit by itself.
    }
  }

  doTask();
})();
```

We've changed two things here:

1. In the `bhttp.get` call, we have added a `stream: true` option; this is what tells bhttp to give us a Node stream instead of a string, and that will also prevent it from reading the whole response in memory at once.
2. We've used the packages that we just installed to construct and use a pipeline, to read out the entire response - in a streaming manner, so not everything is loaded into memory at once! - and process the results item-by-item, either adding them to the queue (if the URL is a sitemap URL) or printing them to the terminal (if it's a page URL instead).

What is actually happening here, is that when you call `.read()` on a pipeline (as we do here), that call is forwarded to the last stream in the pipeline - in our case, that is the `simpleSink` stream, which is designed to start reading from its upstream forever as soon as it is read from once. Or well, "forever" - at least until the stream runs out of values.

This is how every Promistream pipeline works! Sink streams, as a category, are designed as the streams that 'drive' a pipeline; they are responsible for, in some way, reading from the upstream continuously. You will usually use either a `collect` or a `simpleSink` stream, which read forever and then return a Promise, but sink streams are technically allowed to use any read pattern and return any value, and that is how adapters to other stream implementations work too.

The Promise returned from a `simpleSource` stream will resolve when the stream has completely run out of items to process (ie. when our queue is empty), or reject when an error occurs *anywhere within the pipeline*. This means you can await the return value of `pipe(...).read()`, like we're doing here, and the rest of the code will wait for the streaming process to complete, or it will throw an error if something goes wrong. Just like any other async code! This is what makes Promistreams so easy to integrate into other code; it's just an asynchronous function call, really.

After running this code, you should get output that looks something like this:

```
URL found: https://fake-sitemap.slightly.tech/page-0a.html
URL found: https://fake-sitemap.slightly.tech/page-0b.html
URL found: https://fake-sitemap.slightly.tech/page-0c.html
URL found: https://fake-sitemap.slightly.tech/page-0d.html
URL found: https://fake-sitemap.slightly.tech/page-0e.html
URL found: https://fake-sitemap.slightly.tech/page-1a.html
URL found: https://fake-sitemap.slightly.tech/page-1b.html
URL found: https://fake-sitemap.slightly.tech/page-1c.html
URL found: https://fake-sitemap.slightly.tech/page-1d.html
URL found: https://fake-sitemap.slightly.tech/page-1e.html
URL found: https://fake-sitemap.slightly.tech/page-2a.html
URL found: https://fake-sitemap.slightly.tech/page-2b.html
URL found: https://fake-sitemap.slightly.tech/page-2c.html
URL found: https://fake-sitemap.slightly.tech/page-2d.html
```

```
URL found: https://fake-sitemap.slightly.tech/page-2e.html
URL found: https://fake-sitemap.slightly.tech/page-1aa.html
URL found: https://fake-sitemap.slightly.tech/page-1ab.html
URL found: https://fake-sitemap.slightly.tech/page-1ac.html
URL found: https://fake-sitemap.slightly.tech/page-1ad.html
URL found: https://fake-sitemap.slightly.tech/page-1ae.html
URL found: https://fake-sitemap.slightly.tech/page-1ba.html
URL found: https://fake-sitemap.slightly.tech/page-1bb.html
URL found: https://fake-sitemap.slightly.tech/page-1bc.html
URL found: https://fake-sitemap.slightly.tech/page-1bd.html
URL found: https://fake-sitemap.slightly.tech/page-1be.html
```

We're done! Right? Well, not quite, because there's still a problem - we're absolutely *hammering* the server hosting the sitemaps. We're making a lot of requests immediately after each other! With a small sitemap like this it's not a *huge* problem, but this can cause serious issues for real-world sites, which is why **a scraper should always be rate-limited**.

> **Rate-limiting is not optional.** Many websites will ban you if you make repeated requests without a rate limit, often automatically. They may not be willing to unban you, so always make sure *before* you run your scraper code, that you have properly rate-limited it. A good way to do this is to temporarily add a `console.log` before the line of code that makes the HTTP request, and look at the terminal to see whether the delay you are expecting is actually there.

Restricting our scraping speed

There are many ways to implement a rate limit, but the simplest one is to simply have a fixed delay between each request, a time during which nothing happens. Your code would then look something like this:

```
let bhttp = require("bhttp");
let assert = require("node:assert");
const { setTimeout } = require("node:timers/promises");

const pipe = require("@promistream/pipe");
const fromNodeStream = require("@promistream/from-node-stream");
const decodeString = require("@promistream/decode-string");
const parseSitemap = require("@promistream/parse-sitemap");
const simpleSink = require("@promistream/simple-sink");

(async () => { // async IIFE
```

```
  let queue = [];
  queue.push("https://fake-sitemap.slightly.tech/sitemap.xml");

  async function doTask() {
    if (queue.length > 0) {
      let response = await bhttp.get(queue.shift(), { stream: true });
      assert(response.statusCode === 200);

      await pipe([
        fromNodeStream.fromReadable(response),
        decodeString("utf8"),
        parseSitemap(),
        simpleSink((item) => {
          if (item.type === "sitemap") {
            queue.push(item.url);
          } else if (item.type === "url") {
            console.log("URL found:", item.url);
          }
        })
      ]).read();

      await setTimeout(1000);
      return doTask();
    } else {
      // Do nothing, process will exit by itself.
    }
  }

  doTask();
})();
```

We've only really changed one thing here; we're using a special `setTimeout` version provided by Node.js, that returns a Promise (this is not the standard one in JS!). We could've also used any other timed Promise, such as `Promise.delay` in Bluebird. We `await` this timed Promise *just* before we recursively call `doTask` again, and that's all we need - it will now wait a second before processing the next queue item. If you run your code now, you will see that it spits out a set of URLs approximately once every second!

Conclusion

And that's it! You now have a working XML sitemap scraper using Promistreams, that can handle *very large* sitemaps of many gigabytes, without filling up your memory - because everything is

handled in a streaming manner. For now, this guide will leave off here, but it may be expanded at some later date with some extra features, like making *the task queue itself* a Promistream too. If you're curious and want more practice, try to see if you can make that change yourself using `@promistream/simple-queue` and `@promistream/rate-limit`!

Revision #2
Created 2024-12-24 16:02:16 UTC by joepie91
Updated 2024-12-24 20:57:32 UTC by joepie91