Projects

Documentation about various (technical) projects that I run or am involved in in some way.

seekseek.org

A collection of specialized search engines

What is seekseek?

SeekSeek is a search engine project. But instead of being one general search engine, the goal is to build many smaller, highly specialized search engines with specially designed search interfaces, to work optimally for the specific kind of information you're looking for. The focus is on types information which are currently hard to navigate.

There is a common philosophy to all of these search engines: they must be fast, free of clutter, free of advertising and sponsored placements, with an openly available dataset and codebase so that anyone can replicate the work and improve on it.

The first, and currently only, search engine on seekseek is the **datasheet search engine**, which lets you find datasheets very quickly by entering a model number - it finds results as you type, and directly presents you with a download button.

Most search engines on seekseek will be scraping data from many different sources, with custom-made scrapers to ensure data quality. In the future, there will be a mechanism for people to contribute their own data to an open dataset.

The next planned search engine to implement, is a price comparison engine. The work on this is still in progress, and if you know of sources for price feeds, please contact me!

How does it work?

> The following text was originally published on the seekseek website, at https://seekseek.org/technology.

> The information on this page is currently changing. While the current deployment of seekseek still does use the technology as described here, a next version is currently being tested which has some significant architectural changes to better achieve the goals stated below, and to be more maintainable in the long term.

The technology

So... what makes SeekSeek tick? Let's get the boring bits out of the way first:

- The whole thing is written in Javascript, end-to-end, including the scraper.
- Both the scraping server and the search frontend server run on NixOS.
- PostgreSQL is used as the database, both for the scraper and the search frontends (there's only one frontend the time of writing).
- The search frontends use React for rendering the UI; server-side where possible, browser-side where necessary.
- Server-side rendering is done with a fork of `express-react-views`.
- *Most* scraping tasks use bhttp as the HTTP client, and cheerio (a 'headless' implementation of the jQuery API) for data extraction.

None of that is really very interesting, but people always ask about it. Let's move on to the interesting bits!

The goal

Before we can talk about the technology, we need to talk about what the technology was built *for*.

SeekSeek is radical software. From the ground up, it was designed to be FOSS, collaborative and community-driven, non-commercial, ad-free, and to improve the world - in the case of SeekSeek specifically, to improve on the poor state of keyword-only searches by providing highly specialized search engines instead!

But... that introduces some unusual requirements:

- **It needs to be resource-conservative:** While it doesn't need to be *perfectly* optimized, it shouldn't require absurd amounts of RAM or CPU power either. It should be possible to run *the whole thing* on a desktop or a cheap server - the usual refrain of "extra servers are cheaper than extra developers", a very popular one in startups, does not apply here.
- **It needs to be easy to spin up for development:** The entire codebase needs to be self-contained as much as reasonably possible, requiring not much more than an `npm install` to get everything in place. No weirdly complex build stacks, no assumptions about how the developer's system is laid out, and things need to be debuggable by someone who has never touched it before. It needs to be possible for *anybody* to hack on it, not just a bunch of core developers.
- **It needs to be easy to deploy and maintain:** It needs to work with commodity software on standard operating systems, including in constrained environments like containers and VPSes. No weird kernel settings, no complex network setup requirements. It needs to Just Work, and to *keep* working with very little maintenance. Upgrades need to be seamless.
- **It needs to be flexible:** Time is still a valuable resource in a collaborative project - unlike a company, we can't assume that someone will be able to spend a working day restructuring the entire codebase. Likewise, fundamental restructuring causes coordination issues across the community, because a FOSS community is not a centralized entity with a manager who decides what happens. That means that the core (extensible) architecture needs to be right *from the start*, and able to adapt to changing circumstances, more so because scraping is involved.
- **It needs to be accessible:** It should be possible for *any* developer to build and contribute to scrapers; not just specialized developers who have spent half their life working on this sort of thing. That means that the API needs to be simple, and there needs to be space for someone to use the tools they are comfortable with.

At the time of writing, there's only a datasheet search engine. However, the long-term goal is for SeekSeek to become a large *collection* of specialized search engines - each one with a tailor-made UI that's ideal for the thing being searched through. So all of the above needs to be satisfied not just for a datasheet search engine, but for a *potentially unlimited* series of search engines, many of which are not even on the roadmap yet!

And well, the very short version is that *none* of the existing options that I've evaluated even came *close* to meeting these requirements. Existing scraping stacks, job queues, and so on tend to very much be designed for corporate environments with tight control over who works on what. That wasn't an option here. So let's talk about what we ended up with instead!

The scraping server

The core component in SeekSeek is the 'scraping server' - an experimental project called srap that was built specifically for SeekSeek; though also designed to be more generically useful. You can

think of srap as **a persistent job queue that's optimized for scraping**.

So what does that mean? The basic idea behind srap is that you have a big pile of "items" - each item isn't much more than a unique identifier and some 'initial data' to represent the work to be done. Each item can have zero or more 'tags' assigned, which are just short strings. Crucially, none of these items *do* anything yet - they're really just a mapping from an identifier to some arbitrarily-shaped JSON.

The real work starts with the **scraper configuration**. Even though it's called a 'configuration', it's really more of a *codebase* - you can find the configuration that SeekSeek uses here. You'll notice that it defines a number of tasks and seed items. The seed items are simply inserted automatically if they don't exist yet, and define the 'starting point' for the scraper.

The tasks, however, define what the scraper *does*. Every task represents one specific operation in the scraping process; typically, there will be multiple tasks per source. One to find product categories, one to extract products from a category listing, one to extract data from a product page, and so on. Each of these tasks has its own concurrency settings, as well as a TTL (Time-To-Live) that defines after how long the scraper should revisit it.

Finally, what wires it all together are the *tag mappings*. These define what tasks should be executed for what tags - or more accurately, for all the items that are tagged *with* those tags. Tags associated with items are dynamic, they can be added or removed by any scraping task. This provides a *huge* amount of flexibility, because any task can essentially queue any *other* task, just by giving an item the right tag. The scraping server then makes sure that it lands at the right spot in the queue at the right time - the task itself doesn't need to care about any of that.

Here's a practical example, from the datasheet search tasks:

- The initial seed item for LCSC is tagged as `lcsc:home`.
- The `lcsc:home` tag is defined to trigger the `lcsc:findCategories` task.
- The `lcsc:findCategories` task fetches a list of categories from the source, and creates an item tagged as `lcsc:category` for each.
- The `lcsc:category` tag is then defined to trigger the `lcsc:scrapeCategory` task.
- The `lcsc:scrapeCategory` task (more or less) fetches all the products for a given category, and creates items tagged as `lcsc:product`. Importantly, because the LCSC category listings *already* include the product data we need, these items are immediately created with their full data - there's no separate 'scape product page' task!
- The `lcsc:product` tag is then defined to trigger the `lcsc:normalizeProduct` task.
- The `lcsc:normalizeProduct` task then converts the scraped data to a standardized representation, which is stored with a `result:datasheet` tag. The scraping flows for *other* data sources *also* produce `result:datasheet` items - these are the items that ultimately end up in the search frontend!

One thing that's not mentioned above is that `lcsc:scrapeCategory` doesn't *actually* scrape all of the items for a category - it just scrapes a specific page of them! The initial `lcsc:findCategories` task would

have created as many of such 'page tasks' as there are pages to scrape, based on the amount of items a category is said to have.

More interesting, though, is that the scraping flow doesn't *have* to be this unidirectional - if the total amount of pages could only be learned from scraping the first page, it would have been entirely possible for the `lcsc:scrapeCategory` task to create *additional* `lcsc:category` items! The tag-based system makes recursive discovery like this a breeze, and because everything is keyed by a unique identifier and persistent, loops are automatically prevented.

You'll probably have noticed that none of the above mentions HTTP requests. That's because srap doesn't care - it has no idea what HTTP even is! All of the actual scraping *logic* is completely defined by the configuration - and that's what makes it a codebase. This is the scraping logic for extracting products from an LCSC category, for example. This is also why each page is its own item; that allows srap to rate-limit requests despite having absolutely no hooks into the HTTP library being used, by virtue of limiting each task to 1 HTTP request.

There are more features in srap, like deliberately invalidating past scraping results, item merges, and 'out of band' task result storage, but these are the basic concepts that make the whole thing work. As you can see, it's highly flexible, unopinionated, and easy to collaboratively maintain a scraper configuration for - every task functions more or less independently.

The datasheet search frontend

If you've used the datasheet search, you've probably noticed that it's *really* fast, it almost feels like it's all local. But no, your search queries really *are* going to a server. So how can it be that fast?

It turns out to be surprisingly simple: by default, the search is a *prefix search* only. That means that it will only search for items that *start with* the query you entered. This is usually what you want when you search for part numbers, and it *also* has some very interesting performance implications - because a prefix search can be done entirely on an index!

There's actually very little magic here - the PostgreSQL database that runs behind the frontend simply has a (normalized) index on the column for the part number, and the server is doing a `LIKE 'yourquery%'` query against it. That's it! This generally yields a search result in under 2 milliseconds, ie. nearly instantly. All it has to do is an index lookup, and those are *fast*.

On the browser side, things aren't much more complicated. Every time the query changes, it makes a new search request to the server, cancelling the old one if one was still in progress. When it gets results, it renders them on the screen. That's it. There are no trackers on the site, no weird custom input boxes, nothing else to slow it down. The result is a search that feels local :)

The source code

Right now, the source code for all of these things lives across three repositories:

- joepie91/srap - the scraping server.
- seekseek/scraper-config - the configuration and scraping logic that's used for SeekSeek.
- seekseek/ui - the search frontend (including search server!) for SeekSeek.

At the time of writing, documentation is still pretty lacking across these repositories, and the code in the srap and UI repositories in particular is pretty rough! This will be improved upon quite soon, as SeekSeek becomes more polished.

Final words

Of course, there are many more details that I haven't covered in this post, but hopefully this gives you an idea of how SeekSeek is put together, and why!

Has this post made you interested in working on SeekSeek, or maybe your own custom srap-based project? Drop by in the chat! We'd be happy to give you pointers :)

Validatem

An ergonomic and modular validation system for Javascript code - argument validation, arbitrary value validation, everything.

What is Validatem?

> This article is derived from the documentation at https://www.npmjs.com/package/@validatem/core.

The last validation library you'll ever need.

- Does **every kind of validation**, and does it well: it doesn't matter whether you're validating function arguments, form data, JSON request bodies, configuration files, or whatever else. As long as it's structured data of some sort, Validatem can deal with it.
- Supports the notion of **virtual properties** in validation errors, which means that even if your data *isn't* already structured data (eg. an encoded string of some sort), you can bring your own parser, and have it integrate cleanly.
- **Easy to read**; both the code that *uses* Validatem, and the validation error messages that it produces! Your validation code doubles as in-code format documentation, and users get clear feedback about what's wrong.
- Fully **composable**: it's trivial to use third-party validators, or to write your own (reusable!) validators, whether fully custom or made up of a few other validators chained together.
- Supports **value transformation**, which means that you can even encode things like "this value defaults to X" or "when this value is a number, it will be wrapped like so" in your validation code; this can save you a bunch of boilerplate, and makes your validation code *even more complete* as format documentation.
- Validatem has a **small and modular core**, and combined with its composability, this means you won't pull any more code into your project than is strictly necessary to make your validators work! This is also an important part of making Validatem suitable for use in libraries, eg. for argument validation.
- Many **off-the-shelf validators** are already available! You can find the full list here.
- Extensively **documented**, with clear documentation on what is considered valid, and what is not. Likewise, the plumbing libraries that you can use to write your own validators and combinators, are also well-documented.

While Validatem is suitable for any sort of validation, this unique combination of features and design choices makes it *especially* useful for validating arguments in the public API of libraries, unlike other validation libraries!

For example, you might write something like the following (from the `icssify` library):

```
module.exports = function (browserify, options) {
  validateArguments(arguments, {
    browserify: required,
    options: allowExtraProperties({
      mode: oneOf([ "local", "global" ]),
      before: arrayOf([ required, isPostcssPlugin ]),
      after: arrayOf([ required, isPostcssPlugin ]),
      extensions: arrayOf([ required, isString ])
    })
  });

  // Implementation code goes here ...
};
```

And calling it like so:

```
icssify(undefined, {
  mode: "nonExistentMode",
  before: [ NaN ],
  unspecifiedButAllowedOption: true
})
```

... would then produce an error like this:

```
ValidationError: One or more validation errors occurred:
 - At browserify: Required value is missing
 - At options -> mode: Must be one of: 'local', 'global'
 - At options -> before -> 0: Must be a PostCSS plugin
```

Why are there so many packages?

> This article is derived from the documentation at https://www.npmjs.com/package/@validatem/core.

Dependencies often introduce a lot of unnecessary complexity into a project. To avoid that problem, I've designed Validatem to consist of a lot of small, separately-usable pieces - even much of the core plumbing has been split up that way, specifically the bits that may be used by validator and combinator functions.

This may sound counterintuitive; doesn't more dependencies mean *more* complexity? But in practice, "a dependency" in and of itself doesn't have a complexity cost at all; it's the code that is *in* the dependency where the complexity lies. The bigger a dependency is, the more complexity there is in that dependency, and the bigger the chance that some part of that complexity isn't even being used in your project!

By packaging things as granularly as possible, you end up only importing code into your project *that you are actually using*. Any bit of logic that's never used, is somewhere in a package that is never even installed. As an example: using 10 modules with 1 function each, will add less complexity to your project than using 1 module with 100 functions.

This has a lot of benefits, for both you and me:

- **Easier to audit/review:** When only the code you're actually using is added to your project, there will be less code for you to review. And because each piece is designed to be loosely coupled and extensively documented, you can review each (tiny) piece in isolation; without having to trawl through mountains of source code to figure out how it's being called and what assumptions are being made there.
- **Easier to version and maintain:** Most of the modules for Validatem will be completely done and feature-complete the moment they are written, never needing any updates at all. When occasionally a module *does* need an update, it will almost certainly not be one that breaks the API, because the API for each module is so simple that there isn't much *to* break.
- **Easier to upgrade:** Because of almost nothing ever breaking the API, it also means that you'll rarely need to manually upgrade anything, if ever! The vast majority of module updates can be automatically applied, even many years into the future, *even* if a new

(breaking) version of `validatem/@core` is ever released down the line.
- **Easier to fork:** If for any reason you want to fork any part of Validatem, you can do so easily - *without* also having to maintain a big pile of validators, combinators, internals, and so on. You only need to fork and maintain the bit where you want to deviate from the official releases.

Of course, there being so many packages means it can be more difficult to find the specific package you want. That is why the Validatem website has an extensive, categorized list of all the validators, combinators and utilities available for Validatem. Both officially-maintained ones, and third-party modules!

Promistreams

Easy-to-use, composable, universal streams for Javascript.

What are Promistreams?

> This article (and most of the others in this chapter) were derived from a formerly-private draft. It is still subject to change, as Promistreams are polished further towards their first stable release. Despite this, you can test out Promistreams today!

> Promistreams have entered the beta-testing phase! Make sure to read "What to expect from the Promistreams beta phase" before you start using them, so that you know what to expect, and where to report problems. On your left, you will find a menu with several other articles to help you get started.

This is a brief explanation of how Promistreams work, and how to use them, in their current early-ish state.

The core model is pretty well-defined by this point, and major changes in internal structure are not expected to happen. Compatibility is unlikely to be broken in major ways up to the 1.0.0 release, but occasionally you may need to update a few of the libraries at once to keep things working together perfectly. This will typically only involve a version bump, no code changes.

What to expect

Promistreams are pull-based streams. This means that a Promistream does nothing until a value is requested from it, directly or indirectly. This mirrors how streams work in many other languages, and is *unlike* Node.js streams.

Another thing that is unlike Node.js streams, is that a Promistream is always piped into exactly one other stream at a time; though which stream this is, may change over time.

Additionally, Promistreams provide the following features:

- Promise-oriented; 'reading'/running a pipeline returns a Promise, and all internal callbacks work with Promises (including async/await) out of the box. No manual callback wiring!
- Well-defined and consistent error handling.
- Well-defined and consistent cancellation/abort behaviour, including automatically on error conditions.
- Safe concurrent use, ie. having multiple "in-flight" values in the pipeline at once.

- Full interoperability with arbitrary other stream/sequence-shaped types; currently iterables, async iterables and Node.js streams are implemented, but others will follow (please let me know if you need any particular ones!).
- Branching and converging/merging streams, supporting arbitrary distribution patterns, to accommodate cases where one stream needs to be piped into multiple other streams or vice versa.
- Each stream has precise control over when it reads from upstream, and provides results to downstream.
- Composability of streams.

What *not* to expect

There are a few things that Promistreams do *not* prioritize in their design.

Maximum performance. While of course stream implementations will be optimized as much as possible, and a simple core model helps to ensure that, Promistreams don't aim for performance as the #1 goal. This is because doing so would require serious tradeoffs on usability and reliability. In practice, the performance is still quite good, and more than enough for the majority of real-world usecases. Likewise, a lot of work has gone into not making the internals more complex than necessary. But if you are trying to minimize every byte of memory and clock cycle, Promistreams are probably not the right choice for you.

API familiarity. While Promistreams take inspiration from a number of other stream implementations - most notable `pull-streams` and `futures.rs` streams - they do not aim to mirror any one particular streams API. Instead, the API is optimized for usability of the specific model that Promistreams implement, and specifically their use in Javascript.

Aesthetics. While the API is designed to be predictable and easy to reason about, and to roughly represent a pipeline, it does not necessarily look *aesthetically* nice, and some patterns - particularly branching and diverging - may look at bit strange or ugly. The choice was made to optimize for predictability and ergonomics over aesthetic quality, where these goals conflict.

Seekable streams. Like most streams implementations, Promistreams are read-to-end streams, and do not support seeking within streams (although they do support infinite streams, including queues!). If you need seeking capabilities, I would recommend to create a custom abstraction that eg. lets you specify a starting offset and then dispenses a Promistream that starts reading from that offset, and uses happy aborts to stop reading. This way, you get the ergonomics of a Promistream, but can still read specific segments of a resource.

Non-object mode. Node.js streams have two modes; 'regular' mode (Buffers/strings only) and 'object' mode (everything else). Promistreams only have object mode. You can still work with Buffers and strings as before, the streams design is just not specially aware of them, and how they are handled is entirely decided by the specific streams you use.

So this is a library?

Not exactly. I am building this as an interoperable specification, and it's designed so that no libraries are *required* to make use of them; the internal structure of a Promistream is very simple, and contains the absolute minimum complexity to have reliably composable streams.

That having been said, libraries are provided at every level of abstraction, which implement this specification - including high-level streams for specific usecases but also low-level utilities. This means that you can use it *like* a library if you want to, but you can also use it as a spec. These libraries are highly modular; you only install the parts you actually need. Other future implementations of the specification may make different distribution choices.

Currently, the spec is not complete, and is still subject to change. In practice that means that only use-as-a-library is currently viable for real-world projects. The spec is partly written and is pending more real-world experimentation with the current implementation, to find the rough edges and polish them before publishing the specification and 'locking in' the design.

Likewise, many of the libraries are currently missing proper documentation. However, almost every Promistream library that currently exists includes an `example.js` that demonstrates how to use that particular library or stream in your code. Combined with the introduction in this post, that should get you quite far! And if you're stuck, don't hesitate to ask - those questions also help to build out the documentation better.

Known issues

Currently there is a single known issue: a design change was fairly recently made, in the process of formalizing the spec, where it was decided that a 'happy abort' (ie. a cancellation under expected circumstances, like the successful completion of a process that did not need to consume all upstream data) should be exposed by the pipeline as an `EndOfStream` (as if the source stream ran out of data entirely) instead of an `Aborted`.

You may run into some older implementations of source streams that still implement the old behaviour, as this change has not been applied everywhere yet. Please report it when this happens, and they will be fixed promptly!

How do I use Promistreams?

Here's a simple example of a valid Promistream pipeline:

```
"use strict";

const pipe = require("@promistream/pipe");
const fromIterable = require("@promistream/from-iterable");
const map = require("@promistream/map");
const collect = require("@promistream/collect");

(async function() {
  let numbers = await pipe([
    fromIterable([ 1, 2, 3, 4 ]),
    map((number) => number * 2),
    collect()
  ]).read();

  console.log(numbers); // [ 2, 4, 6, 8 ]
})();
```

That's it! That's all there is to it. The call to `pipe` returns a Promise, and you can `await` it like any other Promise - if something goes wrong anywhere inside the pipeline, it automatically aborts the stream, running any teardown logic for each stream in the process, and then throws the original error that caused the failure. Otherwise, you get back whatever output the `collect` stream produced - which is simply an array of every value it has read from upstream.

In this example, the `fromIterable` is what is known as the **source stream** - it provides the original data - and the `collect` stream is what's known as the **sink stream**, which is responsible for reading out the entire pipeline until it is satisfied, which usually means "the source stream has run out of data" (but it *can* choose to behave differently!). The streams inbetween, just `map` in this case, are **transform streams**.

Look carefully at the `pipe` invocation, and how `read` is called on it. This is necessary to 'kickstart' the pipeline. The only thing that `pipe` does is to compose a series of Promistreams into one combined stream, automatically wiring up both ends, and you still need to call `read` on the result to cause the *last* stream in that pipeline to start reading stuff. Doing so is basically equivalent to calling `read` on the `collect` stream directly, with the streams before it as an argument, the `pipe`

function just wires this up for you.

Now this example is not very interesting, because everything is synchronous. But it still works the exact same if we do something asynchronous:

```
"use strict";

const pipe = require("@promistream/pipe");
const fromIterable = require("@promistream/from-iterable");
const map = require("@promistream/map");
const collect = require("@promistream/collect");

(async function() {
  let numbers = await pipe([
    fromIterable([ 1, 2, 3, 4 ]),
    map(async (number) => await doubleNumberRemotely(number)), // I have no idea why you would want to do this
    collect()
  ]).read();

  console.log(numbers); // [ 2, 4, 6, 8 ]
})();
```

We've replaced the `map` callback with one that is asynchronous, and it still works the exact same way! Of course in a real-world project it would be absurd to use a remote service for doubling numbers, but this keeps the example simple to follow. The asynchronous logic could be *anything* - as long as it returns a Promise.

What if we want to reuse this logic in multiple pipelines? We could just have a function that generates a custom `map` stream on-demand. It would need to be a function that *creates* one, because each pipeline would still need its own `map` instance - streams are single-use! It might look like this:

```
"use strict";

const pipe = require("@promistream/pipe");
const fromIterable = require("@promistream/from-iterable");
const map = require("@promistream/map");
const collect = require("@promistream/collect");

function double() {
  // We're using the synchronous version here again, because doubling numbers remotely was a terrible idea!
```

```
    return map((number) => number * 2);
}

(async function() {
    let numbers = await pipe([
        fromIterable([ 1, 2, 3, 4 ]),
        double()
        collect()
    ]).read();

    console.log(numbers); // [ 2, 4, 6, 8 ]
})();
```

As you can see, it's all just functions returning Promises (or values, or throwing errors - the rules are the same as within any `async` context).

But what if we want to make a more complicated custom stream? Let's filter out all the numbers that are a multiple of ten!

```
"use strict";

const pipe = require("@promistream/pipe");
const fromIterable = require("@promistream/from-iterable");
const map = require("@promistream/map");
const filter = require("@promistream/filter");
const collect = require("@promistream/collect");

function doubleAndFilter() {
    return pipe([
        map((number) => number * 2),
        filter((number) => number % 10 > 0)
    ]);
}

(async function() {
    let numbers = await pipe([
        fromIterable([ 1, 2, 3, 4 ]),
        doubleAndFilter()
        collect()
    ]).read();
```

```
  console.log(numbers); // [ 2, 4, 6, 8 ]
})();
```

Wait, what's `pipe` doing there? Well, think back to this earlier comment:

> The only thing that `pipe` does is to compose a series of Promistreams into one combined stream, automatically wiring up both ends, and you still need to call `read` on the result to cause the *last* stream in that pipeline to start reading stuff.

We're now getting to the reason *why* this is the case, and why the composing and reading of pipelines are split up! Because the composing doesn't cause any reads, and it doesn't require a complete pipeline (with a data source and a sink), you can also use it to compose together multiple streams into a single reusable stream, that can then be inserted into another pipeline!

Since we're now both mapping and filtering, this is *exactly* what we need - some way to represent those two streams, in sequence, with pre-specificed callbacks, exposed externally as a single custom stream - and so we use `pipe` to accomplish that. The resulting pipeline works just as if you were to manually insert both of those streams after one another, with the same behaviours and error handling. This is the key to what makes Promistreams composable.

This pretty much covers the basic use of Promistreams. There are many different types of streams, including branching streams, that change the exact behaviour of the pipeline; but the basic operation always looks like this. Pipe together array of streams, call `read` on the resulting pipeline once it's finalized, or return the un-read pipeline if it's meant to be used as a composite stream. Even the more complex pipelines still work like this.

If you want to see what a more real-world example would look like, you may also want to look at the example project tutorial.

The behaviours and responsibilities of different types of streams

While you don't need to know much about the internals of Promistreams to use the libraries, there are a few things that are useful to know, mostly around which streams are responsible for what. In the Promistreams design, much of the behaviour is 'emergent'; it's not enforced by some central runtime or orchestrator, but rather is the emergent result of different parts of the system behaving in certain defined ways.

For example, you might think that the `pipe` function does error handling, but it doesn't! All of the error handling is emergent from the design, and simply a result of how Promises work - a pipeline is essentially just a very long chain of nested Promise callbacks, internally. All that `pipe` does is a bit of `bind` magic to pass the previous stream into the next one.

However, some things *do* need to be defined to make things like error handling and concurrency work correctly. The decision was made to shift this burden to the source and sink streams, as these are the least likely to require a custom implementation - the result is that a transform stream is not much more than an `async` function, and does not need to care about error handling at all if it doesn't want to act on those errors. Errors will simply propagate through them with the usual throw/rejection mechanisms of Promises.

The source and sink streams need to do a bit more; they are responsible for emitting 'markers' and handling rejections, respectively. The 'markers' are `EndOfStream` and `Aborted`, and these are rejected and propagated like an error would be, but they are specially recognized by (some of the) streams inbetween, as well as the sink stream. They're used for teardown code and, in the case of the sink stream, to generate the appropriate 'consumer-facing' error to throw from the pipeline as a whole.

The basic read process looks like this: you call the `read` function on the pipeline, which calls the `read` function on the last stream in it, the **sink stream**. The **sink stream** is responsible for 'driving' the pipeline in some way, though exactly what that looks like will depend on the stream implementation. It is *valid* for a read on the pipeline to only trigger a single upstream read, but that is generally not useful - more typically, the sink stream will start a **read loop**. The stream upstream from it will call `read` on *its* upstream, and so on, recursively, until a value is read from the **source stream**. Any stream inbetween may modify the result, discard values, combine them,

read more times, read less times, and so on. Once the **source stream** runs out of values, it will start dispensing `EndOfStream` markers, which will propagate down like an error, and ultimately signal to the **sink stream** that it should stop any read loops.

The basic abort process looks like this: `abort` is called on any stream in the pipeline, that stream calls `abort` on its upstream, which does the same recursively, until it ends up at the **source stream**. The **source stream** internally 'latches' into 'aborted' mode, and starts dispensing `Aborted` markers on subsequent reads, which are thrown/rejected and therefore propagate back *downstream*, until they eventually end up at the **sink stream**, which unpacks the original error stored within the `Aborted` marker and throws it from its `read` call (and therefore the pipeline's `read` call). Subsequent attempts at reading the sink stream will throw the `Aborted` marker itself, so the original error is not duplicated.

(The details are more complicated, and if the `abort` is a *happy* abort, rather than one based on an Error, the same latching occurs but with `EndOfStream` instead of `Aborted`. Further details will be in the spec.)

When any stream in the pipeline throws an error or rejects a Promise in its `read` callback, this propagates downstream like any error would, until it is received by the **sink stream**. It then initiates the abort process described above.

Note that because `Aborted` and `EndOfStream` markers are thrown/rejected, transform streams inbetween the source and the sink do not need to care about them, unless they intend to implement some kind of teardown logic, in which case they can be intercepted and then re-thrown. But normally they propagate like any rejection would in a chain of Promises, because that is essentially what they are!

Missing from this documentation

Here are some of the things not (yet) covered in this documentation:

- Interoperating with Node streams (see `example.js` in `@promistream/from-node-stream` - this is pretty trivial and it's even entirely valid to *only* use Node streams in your pipeline, using this wrapper)
- Concurrency (this is illustrated in the `example.js` for the `@promistream/parallelize` package)
- Branching (this is illustrates in the `example.js` of various `@promistream/fork-*` packages)
- Converging/merging (currently only illustrated in the `example.js` of the `@promistream/merge-concatenate` package - merge streams are still being worked on)
- The exact details of what source/sink streams are responsible for (part of the unfinished spec)
- How the internal `peek` API works (this is responsible for making concurrency work reliably)

Commonly useful Promistream packages

All of the existing Promistream packages can be found in the package list, but they're not very well-explained. The majority of these should be functional and have an `example.js` demonstrating their use.

Here's a selection of the packages you are most likely to need:

Common cases

- `@promistream/pipe`: the core package that pipes streams together into a pipeline. Technically optional but strongly recommended to use.
- `@promistream/debug`: transform stream that simply prints everything that goes through it, optionally with a label. Only for pipeline debugging use.
- `@promistream/simple-source`: low-level source stream abstraction. Implements the specification responsibilities, leaving you to worry only about how to produce values. Suitable for the majority of usecases.
- `@promistream/simple-sink`: low-level sink stream abstraction. Same as above, but on the other end.
- `@promistream/map`: like the array method, but as a Promistream. Also functions as a general-purpose low-level transform stream.
- `@promistream/filter`: like the array method, but as a Promistream.
- `@promistream/collect`: high-level sink stream, that simply read-loops and collects all read values into an array, then resolves with that array. Often what you want. Also a good example of `@promistream/simple-sink` use, internally.
- `@promistream/from-node-stream`: source/sink/transform wrappers for all types of Node.js streams, to integrate them with a Promistream pipeline.
- `@promistream/from-iterable`: creates source stream from a synchronous or asynchronous iterable (including arrays).
- `@promistream/range-numbers`: high-level source stream, generates numbers in a specified range.

Complex cases

- `@promistream/buffer`: reads an array (of 0 or more items) from upstream, and then dispenses the values in that array (if any) one by one on subsequent reads by its downstream. Often composed with others.
- `@promistream/dynamic`: lets you pass a value through different streams/pipelines depending on the value. Finicky and high-overhead; usually fork-and-merge is a better option.
- `@promistream/parallelize`: lets you run N reads (up to and including `Infinity`) simultaneously.
- `@promistream/sequentialize`: forces inbound reads from downstream to occur sequentially, 'protecting' its upstream. **Mandatory** to use if your stream does not support parallel operation and you intend to publish it, or you use `parallelize` elsewhere in your pipeline.
- `@promistream/rate-limit`: as the name implies, throttles reads going through it but leaves results otherwise unmodified.
- `@promistream/simple-queue`: high-level source stream that functions as a task queue; items can be added externally.

Specific usecases

- `@promistream/read-file`: as the name implies. Produces buffers.
- `@promistream/decode-string`: as the name implies. Takes buffers, produces strings.
- `@promistream/split-lines`: as the name implies. Takes strings.
- `@promistream/parse-xml`: streaming XML parser.

There are other Promistream packages, and there will be many more! These are just some of the ones currently available, that you're likely to need at some point.

Troubleshooting

My process just exits and/or my pipeline doesn't run!

You most likely forgot to call `.read()` on the pipeline. This is easy to forget. I still do it regularly!

I get a weird error!

All `@promistream` libraries are meant to produce clear and understandable errors. If they do not, that is a bug. Please report it!

(Currently this is most likely to happen because of a library not being updated for a newer revision of the specification; I'll help figure out what's going on if you report it.)

How do I...?

Is there an off-the-shelf package for it? Give that a shot first. If there isn't, or it doesn't work as you expect, please let me know and I'll help you figure it out!

Specification (draft)

> This is a draft. It is neither complete nor, probably, correct.

Core concept

A Promistream is, at its core, simply a chain of Promises established through recursive asynchronous calls. A read is requested from the last stream in a pipeline (the "sink"), it requests a read from the stream upstream from it, which requests a read from the stream upstream to *it*, and so on - until the first stream in the pipeline, the "source", is reached.

Each stream along the way may choose to modify both the value itself, as well as the characteristics of it being produced; it may be delayed, filtered, turned into multiple values, or any other transformation that the project requires.

Therefore, the flow (simplified) is as follows:

```
USER CODE  --read-->  SINK STREAM  --read-->  TRANSFORM STREAM(S)  --read-->  SOURCE STREAM  --value-->  TRANSFORM STREAM(S) --value-->  SINK STREAM  --value-->  USER CODE
```

Definitions

This is an overview of the terms used in this specification, as a reading aid; note that their descriptions here are limited, and **may not be sufficient to implement their concepts**. Further technical definition may exist elsewhere in the specification.

- **Stream**: An object which defines (at least) a function that reads a value from another stream and/or yields a value upon being read.
- **Pipeline**: A sequence of streams, which reads and their resulting values propagate through, potentially being transformed in the process.
- **Source stream**: The first stream in a pipeline. This is where values originally originate from. They may be generated on-the-fly, or originate from an external source of some kind.
- **Sink stream**: The last stream in a pipeline. This is where the read is originally initiated.
- **Transform stream**: A stream which exists in a pipeline somewhere inbetween a source and a sink stream.

- **Pipeline completion**: The point where an entire pipeline has been successfully read to its end; as defined by its source stream. A pipeline has only fully completed once all internal buffers on all streams in the pipeline have run out.
- **Pipeline termination:** The point where a pipeline has been terminated or aborted prematurely. Again, it has only fully terminated once all internal buffers on all streams in the pipeline have run out.
- **Buffer:** A mechanism that any stream may have, where it internally stores some value(s) that are already known, but that it does not intend to yield or process until a next read. Frequently used to handle values that result in multiple transformed values (or vice versa), as well as format parsing.
- **Forking stream:** A stream which yields multiple other 'forks' (which are also streams), across which values are distributed or divided in some manner specific to the forking stream (mirroring, round-robin, etc.).
- **Converging stream:** The inverse of a forking stream; it takes multiple 'forks' and combines them back into one coherent stream, in some manner specific to the converging stream (combining, interspersing, etc.).
- **Upstream:** The stream(s) that come before the stream in question, ie. towards the direction of the source stream.
- **Downstream:** The stream(s) that come after the stream in question, ie. towards the direction of the sink stream.
- **EndOfStream marker:** A special error type that signals that the end of the (source) stream has been reached successfully. There will be no more data to read.
- **Aborted marker:** A special error type that signals that the pipeline has been terminated (ie. aborted) prematurely; either due to expected conditions, or due to an unexpected error.
- **Side-effects:** Any kind of externally observable change or interaction that a stream makes outside of its locally-defined state. This includes things like database queries, filesystem operations, but also eg. changing global variables or some other value that is 'owned' by something other than the stream.
- **Error:** Any object that inherits from the `Error` prototype/class, either directly or indirectly (eg. through a custom error type). Note the capitalization!

Backpressure

TODO

Stream API

A stream is any object which has the following properties:

`promistreamVersion`: Set to the value `0`, as of this version of the specification.

`description`: A short string containing a brief textual description (a few words at most) of the stream; this will typically be something like a package name, or a description of the purpose. This

string is for debugging purposes only; it is not guaranteed or expected to be unique or in any particular format, but should be concise enough to be usable in a visualization of a pipeline.

`abort`: A function which, upon invocation, causes the stream to run its stream termination logic. Further defined below.

`peek`: A function which, upon invocation, determines the availability of more values upstream. Further defined below.

`read`: A function which, upon invocation, produces a value. Further defined below.

Whether something constitutes a valid stream, is determined solely by the object shape; the object does *not* need to inherit from any particular prototype, nor does it need to have any particular name.

Below, the behaviours of these functions will be detailed further; note that there will often be special technical requirements for different types of streams, that are detailed after the general description in a subsection.

The `read` function

Signature: `read(source)`

This function should, upon invocation, produce a Promise which eventually resolves (to a read value) or rejects (with an error or marker). The stream MAY internally delay the settling of this Promise to control backpressure. The stream MAY (and typically will) consult the stream directly upstream from it (`source`), by calling its read function, in the process of producing its own value.

When the upstream read was successful, and a value was obtained, the produced Promise will resolve with that value. When the upstream read fails, the produced Promise will reject with an error or marker. Markers are simply Errors of a specific type, that signify the state of the source stream; the possible markers are detailed below.

When you read from upstream, you should be prepared to handle these markers if your stream is managing any resources that it needs to dispose of upon the pipeline completing or terminating (but it SHOULD only do so upon the first observation of a marker). Errors that are not markers can typically be left to propagate downstream; unless specified otherwise below. All markers and errors are rejections rather than resolutions, and so can be handled with a `catch` block, or will automatically propagate if left uncaught.

Note that the `read` function MAY be called while a previous `read` operation is still in flight (see "Ordering considerations" below), and streams should be prepared to handle this.

Types of stream ending

There are three different ways in which a pipeline can end, signified by different markers:

- **`EndOfStream`**: The source stream reached its end successfully and all streams have drained their buffers. This is the success condition, and is initiated by the source stream.
- **`Aborted` (with `true` as reason)**: The source stream was prematurely terminated, but under expected conditions. This is typically initiated by code that is external to the pipeline, signifying that it does not need any further data from the pipeline, and the streams within it can dispose of any resources they are holding onto. This is arguably also a success condition.
- **`Aborted` (with an `Error` as reason)**: The source stream was prematurely terminated, due to an unexpected error. This may have been initiated by external code, but will typically happen from within a stream in the pipeline, when it encounters an unexpected failure. This is the failure condition.

Dealing with uncertain reads

A thing to remember is that a `read` call MUST always produce a Promise which resolves to a value or rejects to an error or marker. It is not allowed (or possible) to respond with a "try again later"; in that case, your stream SHOULD return a Promise that will only be settled at that later moment.

This is especially important for eg. transform streams that do not have a 1:1 correspondence between the values it reads from upstream, and the values that it yields itself, such as filtering streams. In that case, the stream should typically implemented such that it continues reading from upstream until an acceptable value has been obtained, and only then yield that value.

Ordering considerations

By default, streams SHOULD always process reads 'in order'; that is, if there is some kind of correspondence between the yielded values and the upstream reads, the order of these yielded values must also correspond, even in the face of multiple concurrent in-flight read operations. This is also true for markers and errors. Streams MAY produce values out of order, but if they do so, this MUST be clearly documented and serve a specific documented purpose.

If the stream's internal logic is incapable of processing concurrent requests, the stream MUST enforce sequential processing of inbound read requests through some sort of queueing mechanism.

-Non-normative- A stream may be composed with the off-the-shelf `sequentialize` stream to meet this requirement without implementing any custom queueing logic.

Source streams

A source stream will have its `read` function invoked without a `source` argument, as there is no stream upstream of it, and source streams are expected to either generate their values or obtain them from some source external to the pipeline.

A source stream is responsible for producing (and rejecting with) markers when the pipeline has been completed or terminated. It may either reuse a previously-generated marker, or generate a new one upon each `read` invocation.

Sink streams

A sink stream MUST watch for non-marker rejections of upstream reads (ie. rejection with an Error), and if one is observed, call its upstream's `abort` function with that Error as an argument. This is critical to the correct propagation of automatic aborts for unexpected errors.

A sink stream MUST, upon the first post-buffer-drain encounter with an Aborted marker where the `reason` is an Error, reject with that Error (ie. *not* the marker). It MUST propagate the Aborted marker itself on subsequent reads. This ensures that calling code receives the original Error, in the same way that they would if streams had not been used.

A sink stream SHOULD, upon any encounter with an Aborted marker where the `reason` is `true`, reject with that marker to propagate it. It MAY choose to resolve with a value instead.

A sink stream MUST, upon the first read invocation, eventually either resolve with a value (even if it is `undefined`) or reject with an Error. It MAY respond to subsequent `read` invocations by propagating an `EndOfStream` or `Aborted` marker.

Streams that buffer

If you are implementing a stream that has some sort of internal buffering, then there is a special consideration that you need to make for ended streams; when you either generate or receive an `EndOfStream`/`Aborted` marker, you should make sure to process your internal buffers prior to propagating it downstream. Often this will mean buffering up the marker internally, and only returning it once enough reads have occurred to exhaust the internal buffer (although you MAY implement any buffer-draining behaviour that is appropriate to the purpose of your stream).

Since `EndOfStream` and `Aborted` markers are final and reusable, you MAY buffer these up and continue yielding them on future reads in perpetuity, without invoking any further upstream reads, if this makes implementation easier for you.

The `peek` function

Signature: `peek(source)`

This function should, upon invocation, produce a Promise which eventually resolves to a boolean, indicating whether more data is available at the source stream that has not been read yet. While this query MAY be answered by any stream in the pipeline if it has specific reason to need to do so, it SHOULD typically be propagated upstream (to `source`) as-is such that the source stream can answer it (unless specified otherwise below).

Note that any stream which answers `true` to a `peek` request MUST *reserve* or otherwise keep track of the 'peeked' items, to ensure that many subsequent `peek` requests will only result in as many confirmations as there are actual items to be read. In the simplest implementation, this may simply be a counter of how far the pipeline has 'peeked ahead' (decremented by actual reads happening), but for more complex buffering situations it may be necessary to maintain buffers of actual values. A stream responding to a `peek` call MAY therefore invoke the upstream `read` function if necessary to implement its behaviour. however, it MAY NOT initiate any processing of the resulting value if doing so would cause side-effects.

The Promise that is produced for a `peek` call MAY be delayed by the stream, if this is necessary for determining an accurate answer to the query.

-Non-normative, rationale- The purpose of the `peek` function is to support safe parallelization, especially when unbounded; this ensures that even when a parallel reading implementation is allowed to have `Infinity` simultaneous reads, there will only ever be approximately as many in-flight reads as there are items readily available at the source stream. This prevents resource exhaustion. This cannot be implemented simply by doing many `read` calls, as a `read` may take a very long time to be processed and cause side-effects, thereby defeating the point - the `peek` call is instead meant to be propagated more or less directly to the source stream, bypassing any processing delays.

The `abort` function

Signature: `abort(reason, source)`

This function should, upon invocation, do any teardown that is immediately needed when a pipeline is aborted, and then propagate the `abort` call to the stream upstream from it (`source`), passing on the `reason` as well. Note that this is not the only opportunity to do stream teardown; for teardown that is not immediately required, it will often be easier to handle this in the `read` implementation alongside `EndOfStream` markers, as an `Aborted` marker will be passed down upon the next read after the `abort` call has arrived at the source stream.

The `abort` function may be called with a `reason` of either `true` (to indicate a termination under expected conditions, eg. no further data is required) or an Error (to indicate a termination due to an unexpected error somewhere in the pipeline).

Source streams

Source streams must additionally, upon invocation, set an internal 'aborted' flag. Subsequent reads (after any buffer draining) should produce `Aborted` markers.

Sink streams

A sink stream is a stream which is expected to be placed at the end of a pipeline. Upon reading from it, it SHOULD start driving reads above it (eg. in an asynchronous loop), and eventually yield a value that represents the outcome of that read process. It MAY choose to support being read from multiple times, if necessary to make it work ergonomically. The value it yields MAY be something that represents an in-progress read loop (such as a generator), rather than a conclusive value.

-Non-normative- Some examples of the values that a sink stream may yield:

- An array containing all of the values read from upstream, yielded when the pipeline completes. (collect)
- A stream in some other format, that produces values as they are read from upstream (to-node-stream)
- The last observed value, yielded when the pipeline completes (default behaviour of simple-sink)

What to expect from the Promistreams beta phase

Promistreams are currently in their beta testing phase. What this means in practice:

- You can start using Promistreams in your projects today. But be prepared to sometimes debug issues.
- Linear pipelines will generally work perfectly. More complex pipelines may still run into small issues in edgecases.
- The stream interface has not been stabilized yet, but it is also not likely to meaningfully change anymore.
- At any given time, the latest versions of `@promistream` packages should work correctly together.
- A pretty broad collection of streams is already available. Some of the most commonly useful ones are documented on this wiki, and there are more packages on npm.
- However, the stream collection is still incomplete; in particular, several splitting and merging streams are still in active development, as well adapter streams from and to different stream and sequence types.
- Documentation may still be missing for a (shrinking) subset of Promistream packages. However, a Promistream package will typically include an `example.js` that demonstrates its use.
- Bugs should, for now, be reported privately, either on Matrix (@joepie91@pixie.town) or by e-mail (admin@cryto.net). I'll walk through the issue and debug it with you personally. If you're not sure whether it's a bug, then it's a bug, just potentially a documentation bug instead of an implementation bug. Please let me know!

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`!