

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 a 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.

Revision #3

Created 10 December 2024 23:41:11 by joepie91

Updated 24 December 2024 00:27:56 by joepie91