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.

Revision #3
Created 2024-12-10 23:46:13 UTC by joepie91
Updated 2024-12-24 20:56:55 UTC by joepie91