
streamline.js

`streamline.js` is a language tool to simplify asynchronous Javascript programming.

tldr; See Cheat Sheet

Instead of writing hairy code like:

```
1 function archiveOrders(date, cb) {
2   db.connect(function(err, conn) {
3     if (err) return cb(err);
4     conn.query("select * from orders where date < ?", [date], function(
5       err, orders) {
6       if (err) return cb(err);
7       helper.each(orders, function(order, next) {
8         conn.execute("insert into archivedOrders ...", [order.id, ...],
9           function(err) {
10            if (err) return cb(err);
11            conn.execute("delete from orders where id=?", [order.id],
12              function(err) {
13                if (err) return cb(err);
14                next();
15              });
16            });
17          }, function() {
18            console.log("orders have been archived");
19            cb();
20          });
21        });
22      });
23    });
24  }
```

you write:

```
1 function archiveOrders(date, _) {
2   var conn = db.connect(_);
3   conn.query("select * from orders where date < ?", [date], _).forEach_(
4     (_, function(_, order) {
5       conn.execute("insert into archivedOrders ...", [order.id, ...], _);
6       conn.execute("delete from orders where id=?", [order.id], _);
7     });
8   console.log("orders have been archived");
9 }
```

and streamline transforms the code and takes care of the callbacks!

No control flow APIs to learn! You just have to follow a simple rule:

Replace all callbacks by an underscore and write your code as if all functions were synchronous.

Streamline is not limited to a subset of Javascript. You can use all the features of Javascript in your asynchronous code: conditionals, loops, **try/catch/finally** blocks, anonymous functions, chaining, **this**, etc.

Streamline also provides *futures*, and asynchronous variants of the EcmaScript 5 array functions (**forEach**, **map**, etc.).

News

The latest cool feature is **TypeScript** support. See <https://github.com/Sage/streamlinejs/wiki/TypeScript-support> for details.

Streamline 1.0 was a major revamp as a Babel Plugin. Streamline 2.0 was a smaller step from Babel 5 to Babel 6. See <https://github.com/Sage/streamlinejs/wiki/Babel-upgrade> for details.

Installation

NPM, of course:

```
1 npm install streamline -g
```

The **-g** option installs streamline *globally*.

You can also install it *locally* (without **-g**) but then the **_node** and **_coffee** commands will not be in your default PATH.

Note: If you encounter a permission error when installing on UNIX systems, you should retry with **sudo**.

Warning: you may get errors during install because fibers is now installed as an optional package and it may fail to build. But this package is optional and **streamline itself should install fine**.

Hello World

Streamline modules have **._js** or **._coffee** extensions and you run them with **_node** or **_coffee**.

Example:

```
1 $ cat > hello._js
2 console.log('hello ...');
3 setTimeout(_, 1000);
4 console.log('... world');
```

```
5 ^D
6 $ _node hello
```

You can also create standalone shell utilities. See this example.

Compiling and writing loaders

You can also set up your code so that it can be run directly with `node` or `coffee`. You have two options here:

The first one is to compile your source. The recommended way is with babel's CLI (see babel-plugin-streamline). But you can still use streamline's CLI (`_node -c myfile._js` or `_coffee -c myfile._coffee`)

The second one is to create a loader which will register `require` hooks for the `._js` and `._coffee` extensions. See this example.

Compiling will give you the fastest startup time because node will directly load the compiled `*.js` files but the registration API has a `cache` option which comes close.

The recommendation is to use the loader during development but deploy precompiled files.

Runtime dependencies

The runtime library is provided as a separate `streamline-runtime` package.

If you deploy precompiled files you only need `streamline-runtime`.

If your application/library uses a loader you will need to deploy both `streamline-runtime` and `streamline` with it.

Browser-side use

You have two options to use streamline in the browser:

- You can transform and bundle your files with browserify. See how the build.js script builds the `'test/browser/*-test.js` files for an example.
- You can also transform the code in the browser with the `transform` API. All the necessary JS code is available as a single `lib/browser/transform.js` file. See the streamlineMe example.

Generation options

Streamline can transform the code for several target runtimes:

- *callbacks*. The transformed code will be pure ES5 code. It should be compatible with all JavaScript engines.
- *fibers*. The transformed code will take advantage of the fibers library. This option is only available server-side.
- *generators*. The transformed code will take advantage of JavaScript generators. It will run in node.js 0.12 (with the `--harmony` flag), in node.js 4.0 (without any special flag) and in latest browsers.
- *await*. The transformed code will take advantage of ES7 async/await.

The choice of a target runtime should be driven by benchmarks:

- The *fibers* mode gives superior development experience (because it uses real stacks for each fiber so you can step over async calls). It is also very efficient in production if your code traverses many layers of asynchronous calls.
- The *callbacks* transform is obtained by chaining the *generators* transform and the regenerator transform. It is less efficient than the *generators* transform and we recommend that you use *generators* if generators are supported by your target JavaScript engine and that you only use *callbacks* if you target a legacy JavaScript engine.
- The *await* mode is experimental at this stage. It relies on an emulation as async/await is not yet available natively in JavaScript engines.

You can control the target runtime with the `--runtime` (`callbacks` | `fibers` | `generators` | `await`) CLI option, or with the `runtime` API option.

Interoperability with standard node.js code

You can call standard node functions from streamline code. For example the `fs.readFile` function:

```
1 function lineCount(path, _) {
2   return fs.readFile(path, "utf8", _).split('\n').length;
3 }
```

You can also call streamline functions as if they were standard node functions. For example, the `lineCount` function that we just defined above can be called as follows in standard node.js style:

```
1 lineCount("README.md", function(err, result) {
2   if (err) return console.error("ERROR: " + err.message);
```

```
3 console.log("README has " + result + " lines.");
4 });
```

You can mix streamline functions, classical callback based code and synchronous functions in the same file.

Streamline only transforms the functions that have the special `_` parameter.

Note: this works with all transformation options. Even if you use the *fibers* option, you can seamlessly call standard callback based node APIs and the asynchronous functions that you create with streamline have the standard node callback signature.

Interoperability with promises

Streamline also provides seamless interoperability with Promise libraries, in both directions.

First, you can consume promises from streamline code, by passing two underscores to their `then` method:

```
1 function myStreamlineFunction(p1, p2, _) {
2   var result = functionReturningAPromise(p1, p2).then(_, _);
3   // do something with result
4 }
```

Note: if the promise fails the error will be propagated as an exception and you can catch it with `try/catch`.

In the other direction you can get a promise from any callback-based asynchronous function by passing `void _` instead of `_`. For example:

```
1 function readFileWithPromise(path) {
2   var p = fs.readFile(path, 'utf8', void _);
3   // p is a promise.
4   p.then(function(result) {
5     // do something with result
6   }, function(err) {
7     // handle error
8   });
9 }
```

Futures

Streamline also provides *futures*. Futures are like promises, without all the bells and whistles. They let you parallelize I/O operations in a very simple manner.

If you pass `!_` instead of `_` when calling a streamline function, the function returns a *future*. The *future* is just a regular node.js asynchronous function that you can call later to obtain the result. Here is an example:

```
1 function countLines(path, _) {
2   return fs.readFile(path, "utf8", _).split('\n').length;
3 }
4
5 function compareLineCounts(path1, path2, _) {
6   // parallelize the two countLines operations
7   var n1 = countLines(path1, !_);
8   var n2 = countLines(path2, !_);
9   // get the results and diff them
10  return n1(_) - n2(_);
11 }
```

In this example, `countLines` is called twice with `!_`. These calls start the `fs.readFile` asynchronous operations and return immediately two *futures* (`n1` and `n2`). The `return` statement retrieves the results with `n1(_)` and `n2(_)` calls and computes their difference.

See the futures wiki page for details.

The flows module contains utilities to deal with futures. For example `flows.collect` to wait on an array of futures and `flows.funnel` to limit the number of concurrent operations.

Asynchronous Array functions

Streamline extends the Array prototype with asynchronous variants of the EcmaScript 5 `forEach`, `map`, `filter`, `reduce`, ... functions. These asynchronous variants are postfixed with an underscore and they take an extra `_` argument (their callback too), but they are otherwise similar to the standard ES5 functions. Here is an example with the `map_` function:

```
1 function dirLines(dir, _) {
2   return fs.readdir(dir, _).map_(_, function(_, file) {
3     return fs.readFile(dir + '/' + file, 'utf8', _).split('\n').length;
4   });
5 }
```

Parallelizing loops is easy: just pass the number of parallel operations as second argument to the call:

```
1 function dirLines(dir, _) {
2   // process 8 files in parallel
3   return fs.readdir(dir, _).map_(_, 8, function(_, file) {
4     return fs.readFile(dir + '/' + file, 'utf8', _).split('\n').length;
5   });
6 }
```

If you don't want to limit the level of parallelism, just pass `-1`.

See the documentation of the `builtins` module for details.

Exception Handling

Streamline lets you do your exception handling with the usual `try/catch` construct. The `finally` clause is also fully supported.

Streamline overrides the `ex.stack` getter to give you complete comprehensive stacktrace information. In *callbacks* and *generators* modes you get two stack traces:

- the *raw* stack trace of the last callback.
- the *async* stack trace of the asynchronous calls that caused the exception.

In *fibers* mode there is a single stack trace.

Exception handling also works with futures and promises. If a future throws an exception before you try to read its result, the exception is memorized by the future and you get it at the point where your try to read the future's result. For example:

```
1 try {
2   var n1 = countLines(badPath, !_);
3   var n2 = countLines(goodPath, !_);
4   setTimeout(_, 1000); // n1 fails, exception is memorized
5   return n1(_) - n2(_); // exception is thrown by n1(_) expression.
6 } catch (ex) {
7   console.error(ex.stack); // exception caught here
8 }
```

Special callbacks

multiple results

Some APIs return several results through their callback. For example:

```
1 request(options, function(err, response, body) {
2   // ...
3 });
```

You can get all the results by passing `[_]` instead of `_`:

```
1 var results = request(options, [_]);
2 // will be better with destructuring assignment.
3 var response = results[0];
4 var body = results[1];
```

Note: if you only need the first result you can pass `_`:

```
1 var response = request(options, _);
```

callback + errback

Some APIs don't follow the standard *error first* callback convention of node.js. Instead, they accept a pair of callback and errback arguments. Streamline lets you call them by passing two `_` arguments. For example:

```
1 function nodeStyleFn(arg, _) {
2   return callbackErrbackStyleFn(arg, _, _);
3 }
```

As seen above, this feature is used in the promise interop: `result = promise.then(_, _)` is just a special case.

It can also be used to handle the special *error-less* callback of `fs.exists`:

```
1 function fileExists(path, _) {
2   // the second _ is ignored by fs.exists!
3   return fs.exists(path, _, _);
4 }
```

CoffeeScript support

CoffeeScript is fully supported.

Debugging with source maps

You can seamlessly debug streamline code thanks to JavaScript source maps. See this video for a quick demo.

To activate this feature, pass the `--source-map` options to `_node` or `_coffee`, or set the `sourceMap` option if you register via a loader.

Examples

The tutorial shows streamline.js in action on a simple *search aggregator* application.

The diskUsage examples show an asynchronous directory traversal that computes disk usage.

The loader examples demonstrate how you can enable the `._js` and `._coffee` require hooks.

Online demo

You can see how streamline transforms the code by playing with the online demo.

Troubleshooting

Read the FAQ.

If you don't find your answer in the FAQ, post to the mailing list, or file an issue in GitHub's issue tracking.

Related Packages

The following packages are installed together with streamline:

- babel-plugin-streamline: babel plug-in which transforms streamline code.
- streamline-runtime: runtime that you should distribute with compiled modules.

The following packages extend the power of streamline:

- express-streamline: interop with express
- streamline-express: interop with express
- ez-streams: streams and transforms for streamline.
- streamline-flamegraph: flamegraph monitoring.

Resources

The tutorial and FAQ are must-reads for starters.

The API is documented here.

For support and discussion, please join the streamline.js mailing list.

Credits

See the AUTHORS file.

Special thanks to Marcel Laverdet who contributed the *fibers* implementation and to Geoffry Song who contributed source map support (in 0.x versions).

License

MIT