simple-git TS

2.37.0 • Public • Published a month ago

Readme

Explore BETA

3 Dependencies

2,406 Dependents

205 Versions

Install

```
> npm i simple-git
```

Weekly Downloads

1,284,104

Version	License
2.37.0	MIT
Unpacked Size	Total Files
380 kB	262
Issues	Pull Requests
35	1

Homepage

github.com/steveukx/git-js#readme

Repository

github.com/steveukx/git-js

Last publish

a month ago

Collaborators

 

Try on RunKit

Report malware

Simple Git

 

A lightweight interface for running `git` commands in any **node.js** application.

Installation

Use your favourite package manager:

- npm: `npm install simple-git`
- yarn: `yarn add simple-git`

System Dependencies

Requires **git** to be installed and that it can be called using the command `git`.

Usage

Include into your JavaScript app using:

```
// require the library, main export is a function
const simpleGit = require('simple-git');
const git = simpleGit();
```

Include in a TypeScript app using:

```
// Import `SimpleGit` types and the default function exported from `simple
import simpleGit, {SimpleGit} from 'simple-git';
const git: SimpleGit = simpleGit();

// prior to v2.6.0 required importing from `simple-git/promise`
// this import is still available but is now deprecated
import gitP, {SimpleGit} from 'simple-git/promise';
const git: SimpleGit = gitP();
```

Configuration

Configure each `simple-git` instance with a properties object passed to the main `simpleGit` function:

```
import simpleGit, { SimpleGit, SimpleGitOptions } from 'simple-git';

const options: Partial<SimpleGitOptions> = {
   baseDir: process.cwd(),
   binary: 'git',
   maxConcurrentProcesses: 6,
};

// when setting all options in a single object
const git: SimpleGit = simpleGit(options);

// or split out the baseDir, supported for backward compatibility
const git: SimpleGit = simpleGit('/some/path', { binary: 'git' });
```

The first argument can be either a string (representing the working directory for `git` commands to run in), `SimpleGitOptions` object or `undefined`, the second parameter is an optional `SimpleGitOptions` object.

All configuration properties are optional, the default values are shown in the example above.

Per-command Configuration

To prefix the commands run by `simple-git` with custom configuration not saved in the git config (ie: using the `-c` command) supply a `config` option to the instance builder:

```
// configure the instance with a custom configuration property
const git: SimpleGit = simpleGit('/some/path', { config: ['http.proxy=some

// any command executed will be prefixed with this config
// runs: git -c http.proxy=someproxy pull
await git.pull();
```

Configuring Plugins

- Error Detection Customise the detection of errors from the underlying `git` process.
- Progress Events Receive progress events as `git` works through long-running processes.
- Timeout Automatically kill the wrapped `git` process after a rolling timeout.

Using task promises

Each task in the API returns the `simpleGit` instance for chaining together multiple tasks, and each step in the chain is also a `Promise` that can be `await` ed in an `async` function or returned in a `Promise` chain.

```
const simpleGit = require('simple-git');
```

```
const git = simpleGit();

// chain together tasks to await final result
await git.init().addRemote('origin', '...remote.git');

// or await each step individually
await git.init();
await git.addRemote('origin', '...remote.git')
```

Catching errors in async code

To catch errors in async code, either wrap the whole chain in a try/catch:

```
const git = simpleGit()
try {
    await git.init();
    await git.addRemote(name, repoUrl);
}
catch (e) { /* handle all errors here */ }
```

or catch individual steps to permit the main chain to carry on executing rather than jumping to the final `catch` on the first error:

```
const git = simpleGit()
try {
    await git.init().catch(ignoreError);
    await git.addRemote(name, repoUrl);
}
catch (e) { /* handle all errors here */ }

function ignoreError () {}
```

Using task callbacks

In addition to returning promise method can be called with a trailing callback argument

to handle the result of the task

```
const simpleGit = require('simple-git');
const git = simpleGit();
git.init(onInit).addRemote('origin', 'git@github.com:steveukx/git-js.git',

function onInit (err, initResult) { }
function onRemoteAdd (err, addRemoteResult) { }
```

If any of the steps in the chain result in an error, all pending steps will be cancelled, see the **parallel tasks** section for more information on how to run tasks in parallel rather than in series .

Task Responses

Whether using a trailing callback or a Promise, tasks either return the raw `string` or `Buffer` response from the `git` binary, or where possible a parsed interpretation of the response.

For type details of the response for each of the tasks, please see the **TypeScript definitions**.

API

API	What it does
`.add([fileA, ...], handlerFn)`	adds one or more files to be under source control
`.addAnnotatedTag(tagName, tagMessage, handlerFn)`	adds an annotated tag to the head of the current branch
`.addTag(name, handlerFn)`	adds a lightweight tag to the head of the current branch
`.catFile(options[, handlerFn])`	generate `cat-file` detail, `options` should be an array of strings as supported arguments to the cat-file command

API	What it does
`.checkIgnore([filepath, ...], handlerFn)`	checks if filepath excluded by .gitignore rules
`.clearQueue()`	immediately clears the queue of pending tasks (note: any command currently in progress will still call its completion callback)
`.commit(message, handlerFn)`	commits changes in the current working directory with the supplied message where the message can be either a single string or array of strings to be passed as separate arguments (the `git` command line interface converts these to be separated by double line breaks)
`.commit(message, [fileA, ...], options, handlerFn)`	commits changes on the named files with the supplied message, when supplied, the optional options object can contain any other parameters to pass to the commit command, setting the value of the property to be a string will add `name=value` to the command string, setting any other type of value will result in just the key from the object being passed (ie: just `name`), an example of setting the author is below
`.customBinary(gitPath)`	sets the command to use to reference git, allows for using a git binary not available on the path environment variable
`.cwd(workingDirectory)`	Sets the current working directory for all commands after this step in the chain
`.diff(options, handlerFn)`	get the diff of the current repo compared to the last commit with a set of options supplied as a string
`.diff(handlerFn)`	get the diff for all file in the current repo compared to the last commit
	gets a summary of the diff for files in the repo,

API	What it does
`.diffSummary(handlerFn)`	uses the `git diff --stat` format to calculate changes. Handler is called with a nullable error object and an instance of the DiffSummary
`.diffSummary(options, handlerFn)`	includes options in the call to `diff --stat options` and returns a DiffSummary
`.env(name, value)`	Set environment variables to be passed to the spawned child processes, see usage in detail below.
`.exec(handlerFn)`	calls a simple function in the current step
`.fetch([options, ] handlerFn)`	update the local working copy database with changes from the default remote repo and branch, when supplied the options argument can be a standard options object either an array of string commands as supported by the git fetch.
`.fetch(remote, branch, handlerFn)`	update the local working copy database with changes from a remote repo
`.fetch(handlerFn)`	update the local working copy database with changes from the default remote repo and branch
`.log([options], handlerFn)`	list commits between `options.from` and `options.to` tags or branch (if not specified will show all history). Additionally you can provide `options.file`, which is the path to a file in your repository. Then only this file will be considered. `options.symmetric` allows you to specify whether you want to use symmetric revision range (To be compatible, by default, its value is true). For any other set of options, supply `options` as an array of strings to be appended to the `git log` command. To use a custom splitter in the log format, set

API	What it does
	`options.splitter` to be the string the log should be split on. Set `options.multiLine` to true to include a multi-line body in the output format. Options can also be supplied as a standard options object for adding custom properties supported by the git log command.
`.outputHandler(handlerFn)`	attaches a handler that will be called with the name of the command being run and the `stdout` and `stderr` readable streams created by the child process running that command
`.raw(args[, handlerFn])`	Execute any arbitrary array of commands supported by the underlying git binary. When the git process returns a non-zero signal on exit and it printed something to `stderr`, the commmand will be treated as an error, otherwise treated as a success.
`.rebase([options,] handlerFn)`	Rebases the repo, `options` should be supplied as an array of string parameters supported by the git rebase command, or an object of options (see details below for option formats).
`.revert(commit [, options [, handlerFn]])`	reverts one or more commits in the working copy. The commit can be any regular commit-ish value (hash, name or offset such as `HEAD~2`) or a range of commits (eg: `master~5..master~2`). When supplied the options argument contain any options accepted by git-revert.
`.rm([fileA, ...], handlerFn)`	removes any number of files from source control
`.rmKeepLocal([fileA, ...], handlerFn)`	removes files from source control but leaves them on disk

API	What it does
`.stash([options, ][handlerFn])`	Stash the working directory, optional first argument can be an array of string arguments or options object to pass to the git stash command.
`.stashList([options, ][handlerFn])`	Retrieves the stash list, optional first argument can be an object specifying `options.splitter` to override the default value of `;;;;` , alternatively options can be a set of arguments as supported by the `git stash list` command.
`.tag(args[], handlerFn)`	Runs any supported git tag commands with arguments passed as an array of strings .
`.tags([options, ]handlerFn)`	list all tags, use the optional options object to set any options allows by the git tag command. Tags will be sorted by semantic version number by default, for git versions 2.7 and above, use the `--sort` option to set a custom sort.
`.show([options], handlerFn)`	Show various types of objects, for example the file content at a certain commit. `options` is the single value string or array of string commands you want to run

git apply

- `.applyPatch(patch, [options])` applies a single string patch (as generated by `git diff`), optionally configured with the supplied options to set any arguments supported by the apply command. Returns the unmodified string response from `stdout` of the `git` binary.
- `.applyPatch(patches, [options])` applies an array of string patches (as generated by `git diff`), optionally configured with the supplied options to set any arguments supported by the apply command. Returns the unmodified string response from `stdout` of the `git` binary.

git branch

- `.branch([options])` uses the supplied options to run any arguments supported by the branch command. Either returns a BranchSummaryResult instance when listing branches, or a BranchSingleDeleteResult type object when the options included `-d`, `-D` or `--delete` which cause it to delete a named branch rather than list existing branches.
- `.branchLocal()` gets a list of local branches as a BranchSummaryResult instance
- `.deleteLocalBranch(branchName)` deletes a local branch - treats a failed attempt as an error
- `.deleteLocalBranch(branchName, forceDelete)` deletes a local branch, optionally explicitly setting forceDelete to true - treats a failed attempt as an error
- `.deleteLocalBranches(branchNames)` deletes multiple local branches
- `.deleteLocalBranches(branchNames, forceDelete)` deletes multiple local branches, optionally explicitly setting forceDelete to true

git clean

- `.clean(mode)` clean the working tree. Mode should be "n" - dry run or "f" - force
- `.clean(cleanSwitches [,options])` set `cleanSwitches` to a string containing any number of the supported single character options, optionally with a standard options object

git checkout

- `.checkout(checkoutWhat [, options])` - checks out the supplied tag, revision or branch when supplied as a string, additional arguments supported by git checkout can be supplied as an options object/array.

- `.checkout(options)` - uses the checks out the supplied options object/array to check out.

- `.checkoutBranch(branchName, startPoint)` - checks out a new branch from the supplied start point.

- `.checkoutLocalBranch(branchName)` - checks out a new local branch

git clone

- `.clone(repoPath, [localPath, [options]])` clone a remote repo at `repoPath` to a local directory at `localPath`, optionally with a standard options object of additional arguments to include between `git clone` and the trailing `repo local` arguments

- `.clone(repoPath, [options])` clone a remote repo at `repoPath` to a directory in the current working directory with the same name as the repo
- `mirror(repoPath, [localPath, [options]])` behaves the same as the `.clone` interface with the `--mirror` flag enabled.

git config

- `.addConfig(key, value, append = false)` add a local configuration property, when `append` is set to `true` the configuration setting is appended to rather than set in the local config.
- `.listConfig()` reads the current configuration and returns a ConfigListSummary

git hash-object

- `.hashObject(filePath, write = false)` computes the object ID value for the contents of the named file (which can be outside of the work tree), optionally writing the resulting value to the object database.

git init

- `.init(bare [, options])` initialize a repository using the boolean `bare` parameter to intialise a bare repository. Any number of other arguments supported by git init can be supplied as an options object/array.
- `.init([options])` initialize a repository using any arguments supported by git init supplied as an options object/array.

git merge

- `.merge(options)` runs a merge using any configuration options supported by git merge. Conflicts during the merge result in an error response, the response is an instance of MergeSummary whether it was an error or success. When successful, the MergeSummary has all detail from a the PullSummary along with summary detail for the merge. When the merge failed, the MergeSummary contains summary detail for why the merge failed and which files prevented the merge.
- `.mergeFromTo(from, to [, options])` - merge from one branch to another, similar to `.merge` but with the `from` and `to` supplied as strings separately to any

additional the options.

git mv

- `.mv(from, to)` rename or move a single file at `from` to `to`
- `.mv(from, to)` move all files in the `from` array to the `to` directory

git pull

- `.pull([options])` pulls all updates from the default tracked remote, any arguments supported by git pull can be supplied as an options object/array.
- `.pull(remote, branch[, options])` pulls all updates from the specified remote branch (eg 'origin'/'master') along with any custom options object/array

git push

- `.push([options])` pushes to a named remote/branch using any supported options from the git push command. Note that `simple-git` enforces the use of `--verbose --porcelain` options in order to parse the response. You don't need to supply these options.
- `.push(remote, branch[, options])` pushes to a named remote/branch, supports additional options from the git push command.
- `.pushTags(remote[, options])` pushes local tags to a named remote (equivalent to using `.push([remote, '--tags'])`)

git remote

- `.addRemote(name, repo, [options])` adds a new named remote to be tracked as `name` at the path `repo`, optionally with any supported options for the git add call.
- `.getRemotes([verbose])` gets a list of the named remotes, supply the optional `verbose` option as `true` to include the URLs and purpose of each ref
- `.listRemote([options])` lists remote repositories - there are so many optional arguments in the underlying `git ls-remote` call, just supply any you want to use as the optional options eg: `git.listRemote(['--heads', '--tags'], console.log)`

- `.remote([options])` runs a `git remote` command with any number of options
- `.removeRemote(name)` removes the named remote

git reset

- `.reset(resetMode, [resetOptions])` resets the repository, sets the reset mode to one of the supported types (use a constant from the exported `ResetMode` enum, or a string equivalent: `mixed`, `soft`, `hard`, `merge`, `keep`). Any number of other arguments supported by git reset can be supplied as an options object/array.
- `.reset(resetOptions)` resets the repository with the supplied options
- `.reset()` resets the repository in `soft` mode.

git rev-parse / repo properties

- `.revparse([options])` sends the supplied options to git rev-parse and returns the string response from `git`.
- `.checkIsRepo()` gets whether the current working directory is a descendent of a git repository.
- `.checkIsRepo('bare')` gets whether the current working directory is within a bare git repo (see either git clone --bare or git init --bare).
- `.checkIsRepo('root')` gets whether the current working directory is the root directory for a repo (sub-directories will return false).

git status

- `.status([options])` gets the status of the current repo, resulting in a StatusResult. Additional arguments supported by git status can be supplied as an options object/array.

git submodule

- `.subModule(options)` Run a `git submodule` command with on or more arguments passed in as an options array or object
- `.submoduleAdd(repo, path)` Adds a new sub module
- `.submoduleInit([options]` Initialises sub modules, the optional options argument can

be used to pass extra options to the `git submodule init` command.

- `.submoduleUpdate(subModuleName, [options])` Updates sub modules, can be called with a sub module name and options, just the options or with no arguments

How to Specify Options

Where the task accepts custom options (eg: `pull` or `commit`), these can be supplied as an object, the keys of which will all be merged as trailing arguments in the command string, or as a simple array of strings.

Options as an Object

When the value of the property in the options object is a `string`, that name value pair will be included in the command string as `name=value`. For example:

```
// results in 'git pull origin master --no-rebase'
git().pull('origin', 'master', {'--no-rebase': null})

// results in 'git pull origin master --rebase=true'
git().pull('origin', 'master', {'--rebase': 'true'})
```

Options as an Array

Options can also be supplied as an array of strings to be merged into the task's commands in the same way as when an object is used:

```
//
git.pull('origin', 'master', ['--no-rebase'])
```

Release History

Major release 2.x changes the way the queue of tasks are handled to use promises internally and makes available the `.then` and `.catch` methods for integrating with promise consumers or async await.

TypeScript is used by default for all new code, allowing for auto-generated type definitions and a phased re-write of the library rather than a big-bang.

For a per-release overview of changes, see the **changelog**.

2.x Upgrade Notes

When upgrading to release 2.x from 1.x, see the **changelog** for the release 2.0.0

Recently Deprecated / Altered APIs

- ~~2.25.0 depends on Node.js version 12 or above, for use in lower versions of node.js ensure you are also importing the necessary polyfills from `core-js`, see Legacy Node Versions~~ *this change has been reverted in 2.30.0 and will be postponed until version 3.x.*
- 2.13.0 `.push` now returns a PushResult parsed representation of the response.
- 2.11.0 treats tasks chained together as atomic, where any failure in the chain prevents later tasks from executing and tasks called from the root `git` instance as the origin of a new chain, and able to be run in parallel without failures impacting one anther. Prior to this version, tasks called on the root `git` instance would be cancelled when another one failed.
- 2.7.0 deprecates use of `.silent()` in favour of using the `debug` library - see Enable Logging for further details.
- 2.6.0 introduced `.then` and `.catch` as a way to chain a promise onto the current step of the chain. Importing from `simple-git/promise` instead of just `simple-git` is no longer required and is actively discouraged.

For the full history see the **changelog**;

Concurrent / Parallel Requests

When the methods of `simple-git` are chained together, they create an execution chain that will run in series, useful for when the tasks themselves are order-dependent, eg:

```
const git = simpleGit();
git.init().addRemote('origin', 'https://some-repo.git').fetch();
```

Each task requires that the one before it has been run successfully before it is called, any errors in a step of the chain should prevent later steps from being attempted.

When the methods of `simple-git` are called on the root instance (ie: `git = simpleGit()`) rather than chained off another task, it starts a new chain and will not be affected failures in tasks already being run. Useful for when the tasks are independent of each other, eg:

```
const git = simpleGit();
const results = await Promise.all([
    git.raw('rev-parse', '--show-cdup').catch(swallow),
    git.raw('rev-parse', '--show-prefix').catch(swallow),
]);
function swallow (err) { return null }
```

Each `simple-git` instance limits the number of spawned child processes that can be run simultaneously and manages the queue of pending tasks for you. Configure this value by passing an options object to the `simpleGit` function, eg:

```
const git = simpleGit({ maxConcurrentProcesses: 10 });
```

Treating tasks called on the root instance as the start of separate chains is a change to the behaviour of `simple-git` and was added in version `2.11.0`.

Complex Requests

When no suitable wrapper exists in the interface for creating a request, it is possible to run a command directly using `git.raw([...], handler)`. The array of commands are passed directly to the `git` binary:

```
const git = require('simple-git');
const path = '/path/to/repo';
const commands = [ 'config', '--global', 'advice.pushNonFastForward', 'fal

// using an array of commands
git(path).raw(commands, (err, result) => {

  // err is null unless this command failed
  // result is the raw output of this command

});

// using a var-args of strings and awaiting rather than using the callback
const result = await git(path).raw(...commands);
```

Authentication

The easiest way to supply a username / password to the remote host is to include it in the URL, for example:

```
const USER = 'something';
const PASS = 'somewhere';
const REPO = 'github.com/username/private-repo';

const git = require('simple-git');
const remote = `https://${USER}:${PASS}@${REPO}`;

git().silent(true)
  .clone(remote)
  .then(() => console.log('finished'))
  .catch((err) => console.error('failed: ', err));
```

Be sure to enable silent mode to prevent fatal errors from being logged to stdout.

Environment Variables

Pass one or more environment variables to the child processes spawned by `simple-git` with the `.env` method which supports passing either an object of name=value pairs or setting a single variable at a time:

```
const GIT_SSH_COMMAND = "ssh -o UserKnownHostsFile=/dev/null -o StrictHost

const git = require('simple-git');

git()
  .env('GIT_SSH_COMMAND', GIT_SSH_COMMAND)
  .status((err, status) => { /*  */ })

git().env({ ...process.env, GIT_SSH_COMMAND })
  .status()
  .then(status => { })
  .catch(err => {});
```

Note - when passing environment variables into the child process, these will replace the standard `process.env` variables, the example above creates a new object based on `process.env` but with the `GIT_SSH_COMMAND` property added.

TypeScript

To import with TypeScript:

```
import simpleGit, { SimpleGit, StatusResult } from 'simple-git';

const git: SimpleGit = simpleGit();
const status: StatusResult = await git.status();
```

Promise and async compatible

For each task run, the return is the same `SimpleGit` instance for ease of building a series of tasks that all run sequentially and are treated as atomic (ie: if any step fails, the

later tasks are not attempted).

To work with promises (either directly or as part of async/await), simply call the function as before:

```
const simpleGit = require('simple-git');
const git = simpleGit();

// async / await
const status = await git.status();

// promise
git.status().then(result => {...});
```

Exception Handling

When the `git` process exits with a non-zero status (or in some cases like `merge` the git process exits with a successful zero code but there are conflicts in the merge) the task will reject with a `GitError` when there is no available parser to handle the error or a `GitResponseError` for when there is.

See the `err` property of the callback:

```
git.merge((err, mergeSummary) => {
   if (err.git) {
      mergeSummary = err.git; // the failed mergeSummary
   }
})
```

Catch errors with try/catch in async code:

```
try {
  const mergeSummary = await git.merge();
  console.log(`Merged ${ mergeSummary.merges.length } files`);
}
catch (err) {
```

```
  // err.message - the string summary of the error
  // err.stack - some stack trace detail
  // err.git - where a parser was able to run, this is the parsed content

  console.error(`Merge resulted in ${ err.git.conflicts.length } conflicts
}
```

Catch errors with a `.catch` on the promise:

```
const mergeSummary = await git.merge()
   .catch(err => {
      if (err.git) { return err.git; } // the unsuccessful mergeSummary
      throw err;                       // some other error, so throw
   });

if (mergeSummary.failed) {
   console.error(`Merge resulted in ${ mergeSummary.conflicts.length } con
}
```

With typed errors available in TypeScript

```
import simpleGit, { MergeSummary, GitResponseError } from 'simple-git';
try {
  const mergeSummary = await simpleGit().merge();
  console.log(`Merged ${ mergeSummary.merges.length } files`);
}
catch (err) {
  // err.message - the string summary of the error
  // err.stack - some stack trace detail
  // err.git - where a parser was able to run, this is the parsed content
  const mergeSummary: MergeSummary = (err as GitResponseError<MergeSummary
  const conflicts = mergeSummary?.conflicts || [];

  console.error(`Merge resulted in ${ conflicts.length } conflicts`);
}
```

Troubleshooting / FAQ

Enable logging

This library uses **debug** to handle logging, to enable logging, use either the environment variable:

```
"DEBUG=simple-git" node ./your-app.js
```

Or explicitly enable logging using the `debug` library itself:

```
require('debug').enable('simple-git');
```

Enable Verbose Logging

If the regular logs aren't sufficient to find the source of your issue, enable one or more of the following for a more complete look at what the library is doing:

- `DEBUG=simple-git:task:*` adds debug output for each task being run through the library
- `DEBUG=simple-git:task:add:*` adds debug output for specific git commands, just replace the `add` with the command you need to investigate. To output multiple just add them both to the environment variable eg: `DEBUG=simple-git:task:add:*,simple-git:task:commit:*`
- `DEBUG=simple-git:output:*` logs the raw data received from the git process on both `stdOut` and `stdErr`
- `DEBUG=simple-git,simple-git:*` logs *everything*

Every command returns ENOENT error message

There are a few potential reasons:

- `git` isn't available as a binary for the user running the main `node` process, custom paths to the binary can be used with the `.customBinary(...)` api option.
- the working directory passed in to the main `simple-git` function isn't accessible, check it is read/write accessible by the user running the `node` process. This library uses @kwsites/file-exists to validate the working directory exists, to output its logs add `@kwsites/file-exists` to your `DEBUG` environment variable. eg:

```
DEBUG=@kwsites/file-exists,simple-git node ./your-app.js
```

Log format fails

The properties of `git log` are fetched using the `--pretty=format` argument which supports different tokens depending on the version of `git` - for example the `%D` token used to show the refs was added in git `2.2.3`, for any version before that please ensure you are supplying your own format object with properties supported by the version of git you are using.

For more details of the supported tokens, please see the **official `git log` documentation**

Log response properties are out of order

The properties of `git.log` are fetched using the character sequence `ò` as a delimiter. If your commit messages use this sequence, supply a custom `splitter` in the options, for example: `git.log({ splitter: '💻' })`

Pull / Diff / Merge summary responses don't recognise any files

- Enable verbose logs with the environment variable `DEBUG=simple-git:task:*,simple-git:output:*`
- Check the output (for example: `simple-git:output:diff:1 [stdOut] 1 file changed, 1 insertion(+)`)
- Check the `stdOut` output is the same as you would expect to see when running the command directly in terminal
- Check the language used in the response is english locale

In some cases `git` will show progress messages or additional detail on error states in the output for `stdErr` that will help debug your issue, these messages are also included in the verbose log.

Legacy Node Versions

From `v3.x`, `simple-git` will drop support for `node.js` version 10 or below, to use in a lower version of node will result in errors such as:

- `Object.fromEntries is not a function`
- `Object.entries is not a function`
- `message.flatMap is not a function`

To resolve these issues, either upgrade to a newer version of node.js or ensure you are

using the necessary polyfills from `core-js` - see **Legacy Node Versions**.

Examples

using a pathspec to limit the scope of the task

If the `simple-git` api doesn't explicitly limit the scope of the task being run (ie: `git.add()` requires the files to be added, but `git.status()` will run against the entire repo), add a `pathspec` to the command using trailing options:

```
const git = simpleGit();
const wholeRepoStatus = await git.status();
const subDirStatusUsingOptArray = await git.status(['--', 'sub-dir']);
const subDirStatusUsingOptObject = await git.status({'--': null, 'sub-dir'
```

async await

```
async function status (workingDir) {
   const git = require('simple-git');

   let statusSummary = null;
   try {
      statusSummary = await git(workingDir).status();
   }
   catch (e) {
      // handle the error
   }

   return statusSummary;
}

// using the async function
status(__dirname + '/some-repo').then(status => console.log(status));
```

Initialise a git repo if necessary

```
const simpleGit = require('simple-git');
const git = simpleGit(__dirname);
```

```
git.checkIsRepo()
   .then(isRepo => !isRepo && initialiseRepo(git))
   .then(() => git.fetch());

function initialiseRepo (git) {
   return git.init()
      .then(() => git.addRemote('origin', 'https://some.git.repo'))
}
```

Update repo and get a list of tags

```
require('simple-git')(__dirname + '/some-repo')
   .pull()
   .tags((err, tags) => console.log("Latest available tag: %s", tags.lates

// update repo and when there are changes, restart the app
require('simple-git')()
   .pull((err, update) => {
      if(update && update.summary.changes) {
         require('child_process').exec('npm restart');
      }
   });
```

Starting a new repo

```
require('simple-git')()
   .init()
   .add('./*')
   .commit("first commit!")
   .addRemote('origin', 'https://github.com/user/repo.git')
   .push('origin', 'master');
```

push with `-u`

```
require('simple-git')()
   .add('./*')
   .commit("first commit!")
```

```
    .addRemote('origin', 'some-repo-url')
    .push(['-u', 'origin', 'master'], () => console.log('done'));
```

Piping to the console for long running tasks

```
require('simple-git')()
    .outputHandler((bin, stdout, stderr, args) => {
       stdout.pipe(process.stdout);
       stderr.pipe(process.stderr);

       // the name of the binary used, defaults to git, see customBinary fo
       assert.equal(bin, 'git');

       // all other arguments passsed to the binary
       assert.deepEqual(args, ['checkout', 'https://github.com/user/repo.gi
    })
    .checkout('https://github.com/user/repo.git');
```

Update repo and print messages when there are changes, restart the app

```
require('simple-git')()
    .exec(() => console.log('Starting pull...'))
    .pull((err, update) => {
      if(update && update.summary.changes) {
         require('child_process').exec('npm restart');
      }
    })
   .exec(() => console.log('pull done.'));
```

Get a full commits list, and then only between 0.11.0 and 0.12.0 tags

```
require('simple-git')()
    .log((err, log) => console.log(log))
    .log('0.11.0', '0.12.0', (err, log) => console.log(log));
```

Set the local configuration for author, then author for an individual commit

```
require('simple-git')()
   .addConfig('user.name', 'Some One')
   .addConfig('user.email', 'some@one.com')
   .commit('committed as "Some One"', 'file-one')
   .commit('committed as "Another Person"', 'file-two', { '--author': '"An
```

Get remote repositories

```
require('simple-git')()
   .listRemote(['--get-url'], (err, data) => {
      if (!err) {
         console.log('Remote url for repository at ' + __dirname + ':');
         console.log(data);
      }
   });
```

Keywords

git source control vcs

Support

Help

Company

About

Terms & Policies