ShellJS

ShellJS - Unix shell commands for Node.js

Fork me on GitHub

unix passing

windows passing

coverage 97%

npm v0.8.4

downloads 27M/month

ShellJS is a portable **(Windows/Linux/macOS)** implementation of Unix shell commands on top of the Node.js API. You can use it to eliminate your shell script's dependency on Unix while still keeping its familiar and powerful commands. You can also install it globally so you can run it from outside Node projects - say goodbye to those gnarly Bash scripts!

ShellJS is proudly tested on every node release since <!-- start minVersion --> `v6` <!-- stop minVersion -->!

The project is unit-tested and battle-tested in projects like:

- Firebug - Firefox's infamous debugger
- JSHint & ESLint - popular JavaScript linters
- Zepto - jQuery-compatible JavaScript library for modern browsers
- Yeoman - Web application stack and development tool
- Deployd.com - Open source PaaS for quick API backend generation
- And many more.

If you have feedback, suggestions, or need help, feel free to post in our issue tracker.

Think ShellJS is cool? Check out some related projects in our Wiki page!

Upgrading from an older version? Check out our breaking changes page to see
what changes to watch out for while upgrading.

Command line use

If you just want cross platform UNIX commands, checkout our new project shelljs/shx, a utility to expose `shelljs` to
the command line.

For example:

```
$ shx mkdir -p foo
$ shx touch foo/bar.txt
$ shx rm -rf foo
```

Plugin API

ShellJS now supports third-party plugins! You can learn more about using plugins and writing your own ShellJS commands in the wiki.

A quick note about the docs

For documentation on all the latest features, check out our README. To read docs that are consistent with the latest release, check out the npm page or shelljs.org.

Installing

Via npm:

```
$ npm install [-g] shelljs
```

Examples

```
var shell = require('shelljs');

if (!shell.which('git')) {
  shell.echo('Sorry, this script requires git');
  shell.exit(1);
}

// Copy files to release dir
shell.rm('-rf', 'out/Release');
shell.cp('-R', 'stuff/', 'out/Release');

// Replace macros in each .js file
shell.cd('lib');
shell.ls('*.js').forEach(function (file) {
  shell.sed('-i', 'BUILD_VERSION', 'v0.1.2', file);
  shell.sed('-i', /^.*REMOVE_THIS_LINE.*$/, '', file);
  shell.sed('-i', /.*REPLACE_LINE_WITH_MACRO.*\n/, shell.cat('macro.js'), fi
});
shell.cd('..');

// Run external tool synchronously
if (shell.exec('git commit -am "Auto-commit"').code !== 0) {
  shell.echo('Error: Git commit failed');
  shell.exit(1);
}
```

Exclude options

If you need to pass a parameter that looks like an option, you can do so like:

```
shell.grep('--', '-v', 'path/to/file'); // Search for "-v", no grep options

shell.cp('-R', '-dir', 'outdir'); // If already using an option, you're done
```

Global vs. Local

We no longer recommend using a global-import for ShellJS (i.e. `require('shelljs/global')`). While still supported for convenience, this pollutes the global namespace, and should therefore only be used with caution.

Instead, we recommend a local import (standard for npm packages):

```
var shell = require('shelljs');
shell.echo('hello world');
```

Command reference

All commands run synchronously, unless otherwise stated.
All commands accept standard bash globbing characters (`*`, `?`, etc.), compatible with the node `glob` module.

For less-commonly used commands and features, please check out our wiki page.

cat([options,] file [, file ...])

cat([options,] file_array)

Available options:

- `-n`: number all output lines

Examples:

```
var str = cat('file*.txt');
var str = cat('file1', 'file2');
var str = cat(['file1', 'file2']); // same as above
```

Returns a ShellString containing the given file, or a concatenated string containing the files if more than one file is given (a new line character is introduced between each file).

cd([dir])

Changes to directory `dir` for the duration of the script. Changes to home directory if no argument is supplied. Returns a ShellString to indicate success or failure.

chmod([options,] octal_mode || octal_string, file)

chmod([options,] symbolic_mode, file)

Available options:

- `-v`: output a diagnostic for every file processed
- `-c`: like verbose, but report only when a change is made
- `-R`: change files and directories recursively

Examples:

```
chmod(755, '/Users/brandon');
chmod('755', '/Users/brandon'); // same as above
chmod('u+x', '/Users/brandon');
chmod('-R', 'a-w', '/Users/brandon');
```

Alters the permissions of a file or directory by either specifying the absolute permissions in octal form or expressing the changes in symbols. This command tries to mimic the POSIX behavior as much as possible. Notable exceptions:

- In symbolic modes, `a-r` and `-r` are identical. No consideration is given to the `umask`.
- There is no "quiet" option, since default behavior is to run silent.

Returns a ShellString indicating success or failure.

cp([options,] source [, source ...], dest)

cp([options,] source_array, dest)

Available options:

- `-f`: force (default behavior)
- `-n`: no-clobber
- `-u`: only copy if `source` is newer than `dest`
- `-r`, `-R`: recursive
- `-L`: follow symlinks
- `-P`: don't follow symlinks

Examples:

```
cp('file1', 'dir1');
cp('-R', 'path/to/dir/', '~/newCopy/');
cp('-Rf', '/tmp/*', '/usr/local/*', '/home/tmp');
cp('-Rf', ['/tmp/*', '/usr/local/*'], '/home/tmp'); // same as above
```

Copies files. Returns a ShellString indicating success
or failure.

pushd([options,] [dir | '-N' | '+N'])

Available options:

- `-n`: Suppresses the normal change of directory when adding directories to the stack, so that only the stack is manipulated.
- `-q`: Suppresses output to the console.

Arguments:

- `dir`: Sets the current working directory to the top of the stack, then executes the equivalent of `cd dir`.
- `+N`: Brings the Nth directory (counting from the left of the list printed by dirs, starting with zero) to the top of the list by rotating the stack.
- `-N`: Brings the Nth directory (counting from the right of the list printed by dirs, starting with zero) to the top of the list by rotating the stack.

Examples:

```
// process.cwd() === '/usr'
pushd('/etc'); // Returns /etc /usr
pushd('+1');   // Returns /usr /etc
```

Save the current directory on the top of the directory stack and then `cd` to `dir`. With no arguments, `pushd` exchanges the top two directories. Returns an array of paths in the stack.

popd([options,] ['-N' | '+N'])

Available options:

- `-n`: Suppress the normal directory change when removing directories from the stack, so that only the stack is manipulated.
- `-q`: Supresses output to the console.

Arguments:

- `+N`: Removes the Nth directory (counting from the left of the list printed by dirs), starting with zero.
- `-N`: Removes the Nth directory (counting from the right of the list printed by dirs), starting with zero.

Examples:

```
echo(process.cwd()); // '/usr'
pushd('/etc');       // '/etc /usr'
echo(process.cwd()); // '/etc'
popd();              // '/usr'
echo(process.cwd()); // '/usr'
```

When no arguments are given, `popd` removes the top directory from the stack and performs a `cd` to the new top directory. The elements are numbered from 0, starting at the first directory listed with dirs (i.e., `popd` is equivalent to `popd +0`). Returns an array of paths in the stack.

dirs([options | '+N' | '-N'])

Available options:

- `-c`: Clears the directory stack by deleting all of the elements.
- `-q`: Supresses output to the console.

Arguments:

- `+N`: Displays the Nth directory (counting from the left of the list printed by dirs when invoked without options), starting with zero.
- `-N`: Displays the Nth directory (counting from the right of the list printed by dirs when invoked without options), starting with zero.

Display the list of currently remembered directories. Returns an array of paths in the stack, or a single path if `+N` or `-N` was specified.

See also: `pushd`, `popd`

echo([options,] string [, string ...])

Available options:

- `-e`: interpret backslash escapes (default)
- `-n`: remove trailing newline from output

Examples:

```
echo('hello world');
var str = echo('hello world');
echo('-n', 'no newline at end');
```

Prints `string` to stdout, and returns a ShellString.

exec(command [, options] [, callback])

Available options:

- `async`: Asynchronous execution. If a callback is provided, it will be set to `true`, regardless of the passed value (default: `false`).
- `fatal`: Exit upon error (default: `false`).

- `silent`: Do not echo program output to console (default: `false`).
- `encoding`: Character encoding to use. Affects the values returned to stdout and stderr, and what is written to stdout and stderr when not in silent mode (default: `'utf8'`).
- and any option available to Node.js's `child_process.exec()`

Examples:

```
var version = exec('node --version', {silent:true}).stdout;

var child = exec('some_long_running_process', {async:true});
child.stdout.on('data', function(data) {
  /* ... do something with data ... */
});

exec('some_long_running_process', function(code, stdout, stderr) {
  console.log('Exit code:', code);
  console.log('Program output:', stdout);
  console.log('Program stderr:', stderr);
});
```

Executes the given `command` *synchronously*, unless otherwise specified.
When in synchronous mode, this returns a ShellString.
Otherwise, this returns the child process object, and the `callback`
receives the arguments `(code, stdout, stderr)`.

Not seeing the behavior you want? `exec()` runs everything through `sh`
by default (or `cmd.exe` on Windows), which differs from `bash`. If you
need bash-specific behavior, try out the `{shell: 'path/to/bash'}` option.

Security note: as `shell.exec()` executes an arbitrary string in the
system shell, it is **critical** to properly sanitize user input to avoid
command injection. For more context, consult the Security
Guidelines.

find(path [, path ...])

find(path_array)

Examples:

```
find('src', 'lib');
find(['src', 'lib']); // same as above
find('.').filter(function(file) { return file.match(/\.js$/); });
```

Returns a ShellString (with array-like properties) of all
files (however deep) in the given paths.

The main difference from `ls('-R', path)` is that the resulting file names
include the base directories (e.g., `lib/resources/file1` instead of just `file1`).

grep([options,] regex_filter, file [, file ...])

grep([options,] regex_filter, file_array)

Available options:

- `-v`: Invert `regex_filter` (only print non-matching lines).
- `-l`: Print only filenames of matching files.
- `-i`: Ignore case.

Examples:

```
grep('-v', 'GLOBAL_VARIABLE', '*.js');
grep('GLOBAL_VARIABLE', '*.js');
```

Reads input string from given files and returns a ShellString containing all lines of the @ file that match the given `regex_filter`.

head([{'-n': <num>},] file [, file ...])

head([{'-n': <num>},] file_array)

Available options:

- `-n <num>`: Show the first `<num>` lines of the files

Examples:

```
var str = head({'-n': 1}, 'file*.txt');
var str = head('file1', 'file2');
var str = head(['file1', 'file2']); // same as above
```

Read the start of a `file`. Returns a ShellString.

ln([options,] source, dest)

Available options:

- `-s`: symlink
- `-f`: force

Examples:

```
ln('file', 'newlink');
ln('-sf', 'file', 'existing');
```

Links `source` to `dest`. Use `-f` to force the link, should `dest` already exist. Returns a ShellString indicating success or failure.

ls([options,] [path, ...])

ls([options,] path_array)

Available options:

- `-R`: recursive
- `-A`: all files (include files beginning with `.`, except for `.` and `..`)
- `-L`: follow symlinks
- `-d`: list directories themselves, not their contents
- `-l`: list objects representing each file, each with fields containing `ls -l` output fields. See `fs.Stats` for more info

Examples:

```
ls('projs/*.js');
ls('projs/**/*.js'); // Find all js files recursively in projs
ls('-R', '/users/me', '/tmp');
ls('-R', ['/users/me', '/tmp']); // same as above
ls('-l', 'file.txt'); // { name: 'file.txt', mode: 33188, nlink: 1, ...}
```

Returns a ShellString (with array-like properties) of all
the files in the given `path`, or files in the current directory if no
`path` is provided.

mkdir([options,] dir [, dir ...])

mkdir([options,] dir_array)

Available options:

- `-p`: full path (and create intermediate directories, if necessary)

Examples:

```
mkdir('-p', '/tmp/a/b/c/d', '/tmp/e/f/g');
mkdir('-p', ['/tmp/a/b/c/d', '/tmp/e/f/g']); // same as above
```

Creates directories. Returns a ShellString indicating
success or failure.

mv([options ,] source [, source ...], dest')

mv([options ,] source_array, dest')

Available options:

- `-f`: force (default behavior)
- `-n`: no-clobber

Examples:

```
mv('-n', 'file', 'dir/');
mv('file1', 'file2', 'dir/');
mv(['file1', 'file2'], 'dir/'); // same as above
```

Moves `source` file(s) to `dest`. Returns a ShellString
indicating success or failure.

pwd()

Returns the current directory as a ShellString.

rm([options,] file [, file ...])

rm([options,] file_array)

Available options:

- `-f`: force
- `-r, -R`: recursive

Examples:

```
rm('-rf', '/tmp/*');
rm('some_file.txt', 'another_file.txt');
rm(['some_file.txt', 'another_file.txt']); // same as above
```

Removes files. Returns a ShellString indicating success
or failure.

sed([options,] search_regex, replacement, file [, file ...])

sed([options,] search_regex, replacement, file_array)

Available options:

- `-i`: Replace contents of `file` in-place. *Note that no backups will be created!*

Examples:

```
sed('-i', 'PROGRAM_VERSION', 'v0.1.3', 'source.js');
```

Reads an input string from `file`s, line by line, and performs a JavaScript `replace()` on each of the lines from the input string using the given `search_regex` and `replacement` string or
function. Returns the new ShellString after replacement.

Note:

Like unix `sed`, ShellJS `sed` supports capture groups. Capture groups are specified using the `$n` syntax:

```
sed(/(\w+)\s(\w+)/, '$2, $1', 'file.txt');
```

Also, like unix `sed`, ShellJS `sed` runs replacements on each line from the input file (split by '\n') separately, so `search_regex`es that span more than one line (or inlclude '\n') will not match anything and nothing will be replaced.

set(options)

Available options:

- `+/-e`: exit upon error (`config.fatal`)
- `+/-v`: verbose: show all commands (`config.verbose`)
- `+/-f`: disable filename expansion (globbing)

Examples:

```
set('-e'); // exit upon first error
set('+e'); // this undoes a "set('-e')"
```

Sets global configuration variables.

sort([options,] file [, file ...])

sort([options,] file_array)

Available options:

- `-r`: Reverse the results
- `-n`: Compare according to numerical value

Examples:

```
sort('foo.txt', 'bar.txt');
sort('-r', 'foo.txt');
```

Return the contents of the `file`s, sorted line-by-line as a ShellString. Sorting multiple files mixes their content (just as unix `sort` does).

tail([{'-n': <num>},] file [, file ...])

tail([{'-n': <num>},] file_array)

Available options:

- `-n <num>`: Show the last `<num>` lines of `file`s

Examples:

```
var str = tail({'-n': 1}, 'file*.txt');
var str = tail('file1', 'file2');
var str = tail(['file1', 'file2']); // same as above
```

Read the end of a `file`. Returns a ShellString.

tempdir()

Examples:

```
var tmp = tempdir(); // "/tmp" for most *nix platforms
```

Searches and returns string containing a writeable, platform-dependent temporary directory.
Follows Python's tempfile algorithm.

test(expression)

Available expression primaries:

- `'-b', 'path'`: true if path is a block device
- `'-c', 'path'`: true if path is a character device
- `'-d', 'path'`: true if path is a directory
- `'-e', 'path'`: true if path exists
- `'-f', 'path'`: true if path is a regular file
- `'-L', 'path'`: true if path is a symbolic link
- `'-p', 'path'`: true if path is a pipe (FIFO)
- `'-S', 'path'`: true if path is a socket

Examples:

```
if (test('-d', path)) { /* do something with dir */ };
if (!test('-f', path)) continue; // skip if it's a regular file
```

Evaluates `expression` using the available primaries and returns corresponding boolean value.

ShellString.prototype.to(file)

Examples:

```
cat('input.txt').to('output.txt');
```

Analogous to the redirection operator `>` in Unix, but works with `ShellStrings` (such as those returned by `cat`, `grep`, etc.). *Like Unix redirections,* `to()` *will overwrite any existing file!* Returns the same ShellString this operated on, to support chaining.

ShellString.prototype.toEnd(file)

Examples:

```
cat('input.txt').toEnd('output.txt');
```

Analogous to the redirect-and-append operator `>>` in Unix, but works with `ShellStrings` (such as those returned by `cat`, `grep`, etc.). Returns the same ShellString this operated on, to support chaining.

touch([options,] file [, file ...])

touch([options,] file_array)

Available options:

- `-a`: Change only the access time
- `-c`: Do not create any files
- `-m`: Change only the modification time
- `{'-d': someDate}`, `{date: someDate}`: Use `someDate` (instance of `Date`) instead of current time
- `{'-r': file}`, `{reference: file}`: Use `file`'s times instead of current time

Examples:

```
touch('source.js');
touch('-c', 'path/to/file.js');
touch({ '-r': 'referenceFile.txt' }, 'path/to/file.js');
touch({ date: new Date('December 17, 1995 03:24:00') }, 'path/to/file.js');
```

Update the access and modification times of each file to the current time. A file argument that does not exist is created empty, unless `-c` is supplied. This is a partial implementation of `touch(1)`. Returns a ShellString indicating success or failure.

uniq([options,] [input, [output]])

Available options:

- `-i`: Ignore case while comparing
- `-c`: Prefix lines by the number of occurrences
- `-d`: Only print duplicate lines, one for each group of identical lines

Examples:

```
uniq('foo.txt');
uniq('-i', 'foo.txt');
uniq('-cd', 'foo.txt', 'bar.txt');
```

Filter adjacent matching lines from `input`. Returns a ShellString.

which(command)

Examples:

```
var nodeExec = which('node');
```

Searches for `command` in the system's `PATH`. On Windows, this uses the `PATHEXT` variable to append the extension if it's not already executable. Returns a ShellString containing the absolute path to `command`.

exit(code)

Exits the current process with the given exit `code`.

error()

Tests if error occurred in the last command. Returns a truthy value if an error returned, or a falsy value otherwise.

Note: do not rely on the return value to be an error message. If you need the last error message, use the `.stderr` attribute from the last command's return value instead.

ShellString(str)

Examples:

```
var foo = new ShellString('hello world');
```

This is a dedicated type returned by most ShellJS methods, which wraps a string (or array) value. This has all the string (or array) methods, but also exposes extra methods: `.to()`, `.toEnd()`, and all the pipe-able methods (ex. `.cat()`, `.grep()`, etc.). This can be easily converted into a string by calling `.toString()`.

This type also exposes the corresponding command's stdout, stderr, and return status code via the `.stdout` (string), `.stderr` (string), and `.code` (number) properties respectively.

env['VAR_NAME']

Object containing environment variables (both getter and setter). Shortcut to `process.env`.

Pipes

Examples:

```
grep('foo', 'file1.txt', 'file2.txt').sed(/o/g, 'a').to('output.txt');
echo('files with o\'s in the name:\n' + ls().grep('o'));
cat('test.js').exec('node'); // pipe to exec() call
```

Commands can send their output to another command in a pipe-like fashion. `sed`, `grep`, `cat`, `exec`, `to`, and `toEnd` can appear on the right-hand side of a pipe. Pipes can be chained.

Configuration

config.silent

Example:

```
var sh = require('shelljs');
var silentState = sh.config.silent; // save old silent state
sh.config.silent = true;
/* ... */
sh.config.silent = silentState; // restore old silent state
```

Suppresses all command output if `true`, except for `echo()` calls. Default is `false`.

config.fatal

Example:

```
require('shelljs/global');
config.fatal = true; // or set('-e');
cp('this_file_does_not_exist', '/dev/null'); // throws Error here
/* more commands... */
```

If `true`, the script will throw a Javascript error when any shell.js command encounters an error. Default is `false`. This is analogous to Bash's `set -e`.

config.verbose

Example:

```
config.verbose = true; // or set('-v');
cd('dir/');
rm('-rf', 'foo.txt', 'bar.txt');
exec('echo hello');
```

Will print each command as follows:

```
cd dir/
rm -rf foo.txt bar.txt
exec echo hello
```

config.globOptions

Example:

```
config.globOptions = {nodir: true};
```

Use this value for calls to `glob.sync()` instead of the default options.

config.reset()

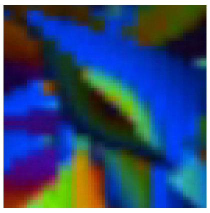
Example:

```
var shell = require('shelljs');
// Make changes to shell.config, and do stuff...
/* ... */
shell.config.reset(); // reset to original state
// Do more stuff, but with original settings
/* ... */
```

Reset `shell.config` to the defaults:

```
{
  fatal: false,
  globOptions: {},
  maxdepth: 255,
  noglob: false,
  silent: false,
  verbose: false,
}
```

Team

	
Nate Fischer	Brandon Freitag