Node.js v8.1.1 Documentation

Index | View on single page | View as JSON

Table of Contents

REPL

Stability: 2 - Stable

The `repl` module provides a Read-Eval-Print-Loop (REPL) implementation that is available both as a standalone program or includible in other applications. It can be accessed using:

```
const repl = require('repl');
```

Design and Features

The `repl` module exports the `repl.REPLServer` class. While running, instances of `repl.REPLServer` will accept individual lines of user input, evaluate those according to a user-defined evaluation function, then output the result. Input and output may be from `stdin` and `stdout`, respectively, or may be connected to any Node.js stream.

Instances of `repl.REPLServer` support automatic completion of inputs, simplistic Emacs-style line editing, multi-line inputs, ANSI-styled output, saving and restoring current REPL session state, error recovery, and customizable evaluation functions.

Commands and Special Keys

The following special commands are supported by all REPL instances:

- `.break` - When in the process of inputting a multi-line expression, entering the `.break` command (or pressing the `<ctrl>-C` key combination) will abort further input or processing of that expression.
- `.clear` - Resets the REPL `context` to an empty object and clears any multi-line expression currently being input.
- `.exit` - Close the I/O stream, causing the REPL to exit.
- `.help` - Show this list of special commands.
- `.save` - Save the current REPL session to a file: `> .save ./file/to/save.js`
- `.load` - Load a file into the current REPL session. `> .load ./file/to/load.js`
- `.editor` - Enter editor mode (`<ctrl>-D` to finish, `<ctrl>-C` to cancel)

```
> .editor
// Entering editor mode (^D to finish, ^C to cancel)
```

```
function welcome(name) {
  return `Hello ${name}!`;
}

welcome('Node.js User');

// ^D
'Hello Node.js User!'
>
```

The following key combinations in the REPL have these special effects:

- `<ctrl>-C` - When pressed once, has the same effect as the `.break` command. When pressed twice on a blank line, has the same effect as the `.exit` command.
- `<ctrl>-D` - Has the same effect as the `.exit` command.
- `<tab>` - When pressed on a blank line, displays global and local(scope) variables. When pressed while entering other input, displays relevant autocompletion options.

Default Evaluation

By default, all instances of `repl.REPLServer` use an evaluation function that evaluates JavaScript expressions and provides access to Node.js' built-in modules. This default behavior can be overridden by passing in an alternative evaluation function when the `repl.REPLServer` instance is created.

JavaScript Expressions

The default evaluator supports direct evaluation of JavaScript expressions:

```
> 1 + 1
2
> const m = 2
undefined
> m + 1
3
```

Unless otherwise scoped within blocks or functions, variables declared either implicitly or using the `const`, `let`, or `var` keywords are declared at the global scope.

Global and Local Scope

The default evaluator provides access to any variables that exist in the global scope. It is possible to expose a variable to the REPL explicitly by assigning it to the `context` object associated with each `REPLServer`. For example:

```
const repl = require('repl');
const msg = 'message';

repl.start('> ').context.m = msg;
```

Properties in the `context` object appear as local within the REPL:

```
$ node repl_test.js
> m
'message'
```

It is important to note that context properties are *not* read-only by default. To specify read-only globals, context properties must be defined using `Object.defineProperty()`:

```
const repl = require('repl');
const msg = 'message';

const r = repl.start('> ');
Object.defineProperty(r.context, 'm', {
  configurable: false,
  enumerable: true,
  value: msg
});
```

Accessing Core Node.js Modules

The default evaluator will automatically load Node.js core modules into the REPL environment when used. For instance, unless otherwise declared as a global or scoped variable, the input `fs` will be evaluated on-demand as `global.fs = require('fs')`.

```
> fs.createReadStream('./some/file');
```

Assignment of the `_` (underscore) variable

The default evaluator will, by default, assign the result of the most recently evaluated expression to the special variable `_` (underscore). Explicitly setting `_` to a value will disable this behavior.

```
> [ 'a', 'b', 'c' ]
[ 'a', 'b', 'c' ]
> _.length
3
> _ += 1
Expression assignment to _ now disabled.
4
> 1 + 1
2
> _
4
```

Custom Evaluation Functions

When a new `repl.REPLServer` is created, a custom evaluation function may be provided. This can be used, for instance, to implement fully customized REPL applications.

The following illustrates a hypothetical example of a REPL that performs translation of text from one language to another:

```
const repl = require('repl');
const { Translator } = require('translator');

const myTranslator = new Translator('en', 'fr');

function myEval(cmd, context, filename, callback) {
  callback(null, myTranslator.translate(cmd));
}

repl.start({ prompt: '> ', eval: myEval });
```

Recoverable Errors

As a user is typing input into the REPL prompt, pressing the `<enter>` key will send the current line of input to the `eval` function. In order to support multi-line input, the eval function can return an instance of `repl.Recoverable` to the provided callback function:

```
function myEval(cmd, context, filename, callback) {
  let result;
  try {
    result = vm.runInThisContext(cmd);
  } catch (e) {
    if (isRecoverableError(e)) {
      return callback(new repl.Recoverable(e));
    }
  }
  callback(null, result);
}

function isRecoverableError(error) {
  if (error.name === 'SyntaxError') {
    return /^(Unexpected end of input|Unexpected token)/.test(error.mess
  }
  return false;
}
```

Customizing REPL Output

By default, `repl.REPLServer` instances format output using the `util.inspect()` method before writing the output to the provided Writable stream (`process.stdout` by default). The `useColors` boolean option can be specified at construction to instruct the default writer to use ANSI style codes to colorize the output from the `util.inspect()` method.

It is possible to fully customize the output of a `repl.REPLServer` instance by passing a new function in using the `writer` option on construction. The following example, for instance, simply converts any input text to upper case:

```
const repl = require('repl');

const r = repl.start({ prompt: '> ', eval: myEval, writer: myWriter });

function myEval(cmd, context, filename, callback) {
  callback(null, cmd);
}

function myWriter(output) {
  return output.toUpperCase();
}
```

Class: REPLServer

Added in: v0.1.91

The `repl.REPLServer` class inherits from the `readline.Interface` class. Instances of `repl.REPLServer` are created using the `repl.start()` method and *should not* be created directly using the JavaScript `new` keyword.

Event: 'exit'

Added in: v0.7.7

The `'exit'` event is emitted when the REPL is exited either by receiving the `.exit` command as input, the user pressing `<ctrl>-C` twice to signal `SIGINT`, or by pressing `<ctrl>-D` to signal `'end'` on the input stream. The listener callback is invoked without any arguments.

```
replServer.on('exit', () => {
  console.log('Received "exit" event from repl!');
  process.exit();
});
```

Event: 'reset'

Added in: v0.11.0

The `'reset'` event is emitted when the REPL's context is reset. This occurs whenever the `.clear` command is received as input *unless* the REPL is using the default evaluator and the `repl.REPLServer` instance was created with the `useGlobal` option set to `true`. The listener callback will be called with a reference to the `context` object as the only argument.

This can be used primarily to re-initialize REPL context to some pre-defined state as illustrated in the following simple example:

```
const repl = require('repl');

function initializeContext(context) {
  context.m = 'test';
}
```

```
const r = repl.start({ prompt: '> ' });
initializeContext(r.context);

r.on('reset', initializeContext);
```

When this code is executed, the global `'m'` variable can be modified but then reset to its initial value using the `.clear` command:

```
$ ./node example.js
> m
'test'
> m = 1
1
> m
1
> .clear
Clearing context...
> m
'test'
>
```

replServer.defineCommand(keyword, cmd)

Added in: v0.3.0

- `keyword` <string> The command keyword (*without* a leading `.` character).
- `cmd` <Object> | <Function> The function to invoke when the command is processed.

The `replServer.defineCommand()` method is used to add new `.`-prefixed commands to the REPL instance. Such commands are invoked by typing a `.` followed by the `keyword`. The `cmd` is either a Function or an object with the following properties:

- `help` <string> Help text to be displayed when `.help` is entered (Optional).
- `action` <Function> The function to execute, optionally accepting a single string argument.

The following example shows two new commands added to the REPL instance:

```
const repl = require('repl');

const replServer = repl.start({ prompt: '> ' });
replServer.defineCommand('sayhello', {
  help: 'Say hello',
  action(name) {
    this.bufferedCommand = '';
    console.log(`Hello, ${name}!`);
    this.displayPrompt();
  }
});
replServer.defineCommand('saybye', function saybye() {
  console.log('Goodbye!');
  this.close();
});
```

The new commands can then be used from within the REPL instance:

```
> .sayhello Node.js User
Hello, Node.js User!
> .saybye
Goodbye!
```

replServer.displayPrompt([preserveCursor])

Added in: v0.1.91

- `preserveCursor` <boolean>

The `replServer.displayPrompt()` method readies the REPL instance for input from the user, printing the configured `prompt` to a new line in the `output` and resuming the `input` to accept new input.

When multi-line input is being entered, an ellipsis is printed rather than the 'prompt'.

When `preserveCursor` is `true`, the cursor placement will not be reset to `0`.

The `replServer.displayPrompt` method is primarily intended to be called from within the action function for commands registered using the `replServer.defineCommand()` method.

repl.start([options])

▶ History

- `options` <Object> | <string>
  - `prompt` <string> The input prompt to display. Defaults to `>` (with a trailing space).
  - `input` <Readable> The Readable stream from which REPL input will be read. Defaults to `process.stdin`.
  - `output` <Writable> The Writable stream to which REPL output will be written. Defaults to `process.stdout`.
  - `terminal` <boolean> If `true`, specifies that the `output` should be treated as a a TTY terminal, and have ANSI/VT100 escape codes written to it. Defaults to checking the value of the `isTTY` property on the `output` stream upon instantiation.
  - `eval` <Function> The function to be used when evaluating each given line of input. Defaults to an async wrapper for the JavaScript `eval()` function. An `eval` function can error with `repl.Recoverable` to indicate the input was incomplete and prompt for additional lines.
  - `useColors` <boolean> If `true`, specifies that the default `writer` function should include ANSI color styling to REPL output. If a custom `writer` function is provided then this has no effect. Defaults to the REPL instances `terminal` value.
  - `useGlobal` <boolean> If `true`, specifies that the default evaluation function will use the JavaScript `global` as the context as opposed to creating

a new separate context for the REPL instance. Defaults to `false`.

  - `ignoreUndefined` <boolean> If `true`, specifies that the default writer will not output the return value of a command if it evaluates to `undefined`. Defaults to `false`.
  - `writer` <Function> The function to invoke to format the output of each command before writing to `output`. Defaults to `util.inspect()`.
  - `completer` <Function> An optional function used for custom Tab auto completion. See `readline.InterfaceCompleter` for an example.
  - `replMode` <symbol> A flag that specifies whether the default evaluator executes all JavaScript commands in strict mode or default (sloppy) mode. Acceptable values are:
    - `repl.REPL_MODE_SLOPPY` - evaluates expressions in sloppy mode.
    - `repl.REPL_MODE_STRICT` - evaluates expressions in strict mode. This is equivalent to prefacing every repl statement with `'use strict'`.
    - `repl.REPL_MODE_MAGIC` - This value is **deprecated**, since enhanced spec compliance in V8 has rendered magic mode unnecessary. It is now equivalent to `repl.REPL_MODE_SLOPPY` (documented above).
  - `breakEvalOnSigint` - Stop evaluating the current piece of code when `SIGINT` is received, i.e. `Ctrl+C` is pressed. This cannot be used together with a custom `eval` function. Defaults to `false`.

The `repl.start()` method creates and starts a `repl.REPLServer` instance.

If `options` is a string, then it specifies the input prompt:

```
const repl = require('repl');

// a Unix style prompt
repl.start('$ ');
```

The Node.js REPL

Node.js itself uses the `repl` module to provide its own interactive interface for executing JavaScript. This can be used by executing the Node.js binary without passing any arguments (or by passing the `-i` argument):

```
$ node
> const a = [1, 2, 3];
undefined
> a
[ 1, 2, 3 ]
> a.forEach((v) => {
...   console.log(v);
...   });
1
2
3
```

Environment Variable Options

Various behaviors of the Node.js REPL can be customized using the following environment variables:

- `NODE_REPL_HISTORY` - When a valid path is given, persistent REPL history will be saved to the specified file rather than `.node_repl_history` in the user's home directory. Setting this value to `""` will disable persistent REPL history. Whitespace will be trimmed from the value.
- `NODE_REPL_HISTORY_SIZE` - Defaults to `1000`. Controls how many lines of history will be persisted if history is available. Must be a positive number.
- `NODE_REPL_MODE` - May be any of `sloppy`, `strict`, or `magic`. Defaults to `sloppy`, which will allow non-strict mode code to be run. `magic` is **deprecated** and treated as an alias of `sloppy`.

Persistent History

By default, the Node.js REPL will persist history between `node` REPL sessions by saving inputs to a `.node_repl_history` file located in the user's home directory. This can be disabled by setting the environment variable `NODE_REPL_HISTORY=""`.

NODE_REPL_HISTORY_FILE

Added in: v2.0.0 Deprecated since: v3.0.0

Stability: 0 - Deprecated: Use `NODE_REPL_HISTORY` instead.

Previously in Node.js/io.js v2.x, REPL history was controlled by using a `NODE_REPL_HISTORY_FILE` environment variable, and the history was saved in JSON format. This variable has now been deprecated, and the old JSON REPL history file will be automatically converted to a simplified plain text format. This new file will be saved to either the user's home directory, or a directory defined by the `NODE_REPL_HISTORY` variable, as documented in the Environment Variable Options.

Using the Node.js REPL with advanced line-editors

For advanced line-editors, start Node.js with the environmental variable `NODE_NO_READLINE=1`. This will start the main and debugger REPL in canonical terminal settings, which will allow use with `rlwrap`.

For example, the following can be added to a `.bashrc` file:

```
alias node="env NODE_NO_READLINE=1 rlwrap node"
```

Starting multiple REPL instances against a single running instance

It is possible to create and run multiple REPL instances against a single running instance of Node.js that share a single `global` object but have separate I/O interfaces.

The following example, for instance, provides separate REPLs on `stdin`, a Unix socket, and a TCP socket:

```
const net = require('net');
const repl = require('repl');
let connections = 0;

repl.start({
  prompt: 'Node.js via stdin> ',
  input: process.stdin,
  output: process.stdout
});

net.createServer((socket) => {
  connections += 1;
  repl.start({
    prompt: 'Node.js via Unix socket> ',
    input: socket,
    output: socket
  }).on('exit', () => {
    socket.end();
  });
}).listen('/tmp/node-repl-sock');

net.createServer((socket) => {
  connections += 1;
  repl.start({
    prompt: 'Node.js via TCP socket> ',
    input: socket,
    output: socket
  }).on('exit', () => {
    socket.end();
  });
}).listen(5001);
```

Running this application from the command line will start a REPL on stdin. Other REPL clients may connect through the Unix socket or TCP socket. `telnet` , for instance, is useful for connecting to TCP sockets, while `socat` can be used to connect to both Unix and TCP sockets.

By starting a REPL from a Unix socket-based server instead of stdin, it is possible to connect to a long-running Node.js process without restarting it.

For an example of running a "full-featured" (`terminal`) REPL over a `net.Server` and `net.Socket` instance, see: https://gist.github.com/2209310

For an example of running a REPL instance over curl(1), see: https://gist.github.com/2053342