Node.js Cheatsheet.

VOL - 1

@vishwajeet.js

In **browsers**, the **top-level scope** is the **global scope.**

That means that in browsers if you're in the global scope var something will define a global variable.

In **Node** this is different. The **top-level scope** is **not the global scope**; var something inside a Node module will be local to that module.

__filename; The filename of the code being executed. (absolute path)

__dirname; The name of the directory that the currently executing script resides in. (absolute path)

process; The process object is a global object and can be accessed from anywhere. It is an instance of EventEmitter.

Modules

```
var module = require('./module.js');
// Loads the module.js in same directory

module.require('./another_module.js');
// load another_module as if require() was called from
the module itself.

module.id;
// The identifier for the module. Typically this is the
fully resolved filename.

module.filename;
// The fully resolved filename to the module.

module.loaded;
// Whether or not the module is done loading, or is in
the process of loading.

module.parent;
// The module that required this one.

module.children;
// The module objects required by this one.
```

Path in Node.js

```
// Use require('path') to use this module.
// This module contains utilities for handling and transforming
file paths.
// Almost all these methods perform only string transformations.
// The file system is not consulted to check whether paths are
valid.

path.normalize(p);
// Normalize a string path, taking care of '..' and '.' parts.
path.join([path1], [path2], [...]);
// Join all arguments together and normalize the resulting path.
path.resolve([from ...], to);
// Resolves 'to' to an absolute path.
path.relative(from, to);
// Solve the relative path from 'from' to 'to'.
path.dirname(p);
// Return the directory name of a path. Similar to the Unix
dirname command.
path.basename(p, [ext]);
// Return the last portion of a path. Similar to the Unix
basename command.
path.extname(p);
// Return the extension of the path, from the last '.' to end of
string in the last portion of the path.
path.sep;
// The platform-specific file separator. '\\' or '/'.
path.delimiter;
// The platform-specific path delimiter, ';' or ':'.
```

Process in Node.js

```
process.on('exit', function(code) {});
// Emitted when the process is about to exit
process.on('uncaughtException', function(err) {});
// Emitted when an exception bubbles all the way back
to the event loop. (should not be used)

process.stdout;       // A writable stream to stdout.
process.stderr;       // A writable stream to stderr.
process.stdin;        // A readable stream for stdin.
process.argv;
// An array containing the command line arguments.
process.env;
// An object containing the user environment.
process.execPath;
// This is the absolute pathname of the executable
that started the process.
process.execArgv;
// This is the set of node-specific command line
options from the executable that started the process.
process.arch;
// What processor architecture you're running on:
'arm', 'ia32', or 'x64'.
process.config;
// An Object containing the JavaScript representation
of the configure options that were used to compile
the current node executable.
process.pid;
// The PID of the process.
process.platform;
// What platform you're running on: 'darwin',
'freebsd', 'linux', 'sunos' or 'win32'.
```

Process in Node.js

```
process.abort();
// This causes node to emit an abort. This will cause
node to exit and generate a core file.
process.chdir(dir);
// Changes the current working directory of the
process or throws an exception if that fails.
process.exit([code]);
// Ends the process with the specified code. If
omitted, exit uses the 'success' code 0.
process.getgid();
// Gets the group identity of the process.
process.setgid(id);
// Sets the group identity of the process.
process.getuid();
// Gets the user identity of the process.
process.setuid(id);
// Sets the user identity of the process.
process.getgroups();
// Returns an array with the supplementary group IDs.
process.setgroups(grps);
// Sets the supplementary group IDs.
process.kill(pid, [signal]);
// Send a signal to a process. pid is the process id
and signal is the string describing the signal to
send.
process.memoryUsage();
// Returns an object describing the memory usage of
the Node process measured in bytes.
process.uptime();
// Number of seconds Node has been running.
```

HTTP in Node.js

```
// To use the HTTP server and client one must
require('http').

http.STATUS_CODES;
// A collection of all the standard HTTP response
status codes, and the short description of each.
http.request(options, [callback]);
// This function allows one to transparently issue
requests.
http.get(options, [callback]);
// Set the method to GET and calls req.end()
automatically.

server = http.createServer([requestListener]);

// Returns a new web server object. The
requestListener is a function which is automatically
added to the 'request' event.
server.listen(port, [hostname], [backlog],
[callback]);
// Begin accepting connections on the specified port
and hostname.
server.listen(path, [callback]);
// Start a UNIX socket server listening for
connections on the given path.
server.listen(handle, [callback]);
// The handle object can be set to either a server or
socket (anything with an underlying _handle member),
or a {fd: <n>} object.
server.close([callback]);
// Stops the server from accepting new connections.
```

HTTP in Node.js

```
server.setTimeout(msecs, callback);
// Sets the timeout value for sockets, and emits a
'timeout' event on the Server object, passing the
socket as an argument, if a timeout occurs.

server.maxHeadersCount;
// Limits maximum incoming headers count, equal to
1000 by default. If set to 0 - no limit will be
applied.
server.timeout;
// The number of milliseconds of inactivity before a
socket is presumed to have timed out.

server.on('request', function (request, response) {
});
// Emitted each time there is a request.
server.on('connection', function (socket) { });
// When a new TCP stream is established.
server.on('close', function () { });
// Emitted when the server closes.
server.on('checkContinue', function (request,
response) { });
// Emitted each time a request with an http Expect:
100-continue is received.
server.on('connect', function (request, socket, head)
{ });
// Emitted each time a client requests a http CONNECT
method.
server.on('upgrade', function (request, socket, head)
{ });
// Emitted each time a client requests a http
upgrade.
```

HTTP in Node.js

```
server.on('clientError', function (exception, socket)
{ });
// If a client connection emits an 'error' event - it
will forwarded here.
request.write(chunk, [encoding]);
// Sends a chunk of the body.
request.end([data], [encoding]);
// Finishes sending the request. If any parts of the
body are unsent, it will flush them to the stream.
request.abort();  // Aborts a request.
request.setTimeout(timeout, [callback]);
// Once a socket is assigned to this request and is
connected socket.setTimeout() will be called.
request.setNoDelay([noDelay]);
// Once a socket is assigned to this request and is
connected socket.setNoDelay() will be called.
request.setSocketKeepAlive([enable], [initialDelay]);
// Once a socket is assigned to this request and is
connected socket.setKeepAlive() will be called.
request.on('response', function(response) { });
// Emitted when a response is received to this
request. This event is emitted only once.
request.on('socket', function(socket) { });
// Emitted after a socket is assigned to this
request.
request.on('connect', function(response, socket,
head) { });
// Emitted each time a server responds to a request
with a CONNECT method. If this event isn't being
listened for, clients receiving a CONNECT method will
have their connections closed.
```

HTTP in Node.js

```
request.on('upgrade', function(response, socket,
head) { });
// Emitted each time a server responds to a request
with an upgrade. If this event isn't being listened
for, clients receiving an upgrade header will have
their connections closed.
response.write(chunk, [encoding]);
// This sends a chunk of the response body. If this
merthod is called and response.writeHead() has not
been called, it will switch to implicit header mode
and flush the implicit headers.
response.writeHead(statusCode, [reasonPhrase],
[headers]);
// Sends a response header to the request.
response.setHeader(name, value);
// Sets a single header value for implicit headers.
If this header already exists in the to-be-sent
headers, its value will be replaced. Use an array of
strings here if you need to send multiple headers
with the same name.
response.getHeader(name);
// Reads out a header that's already been queued but
not sent to the client. Note that the name is case
insensitive.
response.removeHeader(name);
// Removes a header that's queued for implicit
sending.
response.end([data], [encoding]);
// This method signals to the server that all of the
response headers and body have been sent; that server
should consider this message complete. The method,
response.end(), MUST be called on each response.
```