

Basic Webserver example

```
var http = require('http');
http.createServer(function (request, response) {
    response.writeHead(
    200, {'Content-Type':
    'text/plain'});
    response.end('Hello
    World\n');
}).listen(8124);
console.log('Server running
    at http://127.0.0.1:8124/');
```

Timers

To schedule execution of a one-time callback after delay milliseconds. Optionally you can also pass arguments to the callback.

```
setTimeout(callback, delay,
    [arg], [...]);
```

Stop a timer that was previously created with `setTimeout()`.

```
clearTimeout(t);
```

To schedule the repeated execution of callback every delay milliseconds. Optionally you can also pass arguments to the callback.

```
setInterval(callback, delay,
    [arg], [...]);
```

Stop a timer that was previously created with `setInterval()`.

```
clearInterval(t);
```

To schedule the "immediate" execution of callback after I/O events callbacks and before `setTimeout` and `setInterval`.

```
setImmediate(callback,
    [arg], [...]);
```

Stop a timer that was previously created with `setImmediate()`.

Timers (cont)

> **clearImmediate(immediateObject);**

Allow you to create a timer that is active but if it is the only item left in the event loop, node won't keep the program running.

```
unref();
```

If you had previously `unref()`d a timer you can call `ref()` to explicitly request the timer hold the program open.

```
ref();
```

Events

Adds a listener to the end of the listeners array for the specified event.

```
emit(event, ...args, listener);
```

Same as `emit(event, ...args, listener)`.

```
emit(event, listener);
```

Adds a one time listener for the event. This listener is invoked only the next time the event is fired, after which it is removed.

```
emit(event, ...args, listener);
```

Remove a listener from the listener array for the specified event.

```
emit(event, ...args, listener);
```

Removes all listeners, or those of the specified event.

```
emit(event, ...args, listener);
```

By default `EventEmitters` will print a warning if more than 10 listeners are added for a particular event.

```
emit(event, ...args, listener);
```

Events (cont)

> Returns an array of listeners for the specified event.

```
emitter.listeners(event);
```

Execute each of the listeners in order with the supplied arguments. Returns true if event had listeners, false otherwise.

```
emitter.emit(event, [arg1], [arg2], [...]);
```

Return the number of listeners for a given event.

```
EventEmitter.listenerCount(emitter, event);
```

File system

Write buffer to the file specified by `fd`.

```
fs.write(fd, buffer, offset,
    length, position, callback);
```

Synchronous version of `fs.write()`. Returns the number of bytes written.

```
fs.writeSync(fd, buffer,
    offset, length, position);
```

Read data from the file specified by `fd`.

```
fs.read(fd, buffer, offset,
    length, position, callback);
```

Synchronous version of `fs.read`. Returns the number of bytes read.

```
fs.readSync(fd, buffer,
    offset, length, position);
```

Asynchronously reads the entire contents of a file.

```
fs.readFile(filename,
    [options], callback);
```



By **camilo.herbert**

cheatography.com/camilo-herbert/

Not published yet.

Last updated 3rd August, 2018.

Page 1 of 7.

Sponsored by **Readable.com**

Measure your website readability!

<https://readable.com>

File system (cont)

> Synchronous version of `fs.readFile`.
Returns the contents of the filename. If the encoding option is specified then this function returns a string. Otherwise it returns a buffer.

`fs.readFileSync(filename, [options]);`
Asynchronously writes data to a file, replacing the file if it already exists. data can be a string or a buffer.

`fs.writeFile(filename, data, [options], callback);`
The synchronous version of `fs.writeFile`.

`fs.writeFileSync(filename, data, [options]);`
Asynchronously append data to a file, creating the file if it not yet exists. data can be a string or a buffer.

`fs.appendFile(filename, data, [options], callback);`
The synchronous version of `fs.appendFile`.

`fs.appendFileSync(filename, data, [options]);`
Watch for changes on filename, where filename is either a file or a directory. The returned object is a `fs.FSWatcher`. The listener callback gets two arguments (event, filename). event is either 'rename' or 'change', and filename is the name of the file which triggered the event.

`fs.watch(filename, [options], [listener]);`
Test whether or not the given path exists by checking with the file system. Then call the callback argument with either true or false. (should not be used)

File system (cont)

> **`fs.exists(path, callback);`**
Synchronous version of `fs.exists`. (should not be used)

`fs.existsSync(path);`
`fs.Stats`: objects returned from `fs.stat()`, `fs.lstat()` and `fs.fstat()` and their synchronous counterparts are of this type.

`stats.isFile();`
`stats.isDirectory();`
`stats.isBlockDevice();`
`stats.isCharacterDevice();`
`stats.isSymbolicLink()` // (only valid with `fs.lstat()`)
`stats.isFIFO();`
`stats.isSocket();`
Returns a new `ReadStream` object.

`fs.createReadStream(path, [options]);`

HTTP - Requests

Sends a chunk of the body.

`request.write(chunk, [encoding]);`
Finishes sending the request. If any parts of the body are unsent, it will flush them to the stream.

`request.end([data], [encoding]);`
Aborts a request.

`request.abort();`
Once a socket is assigned to this request and is connected `socket.setTimeout()` will be called.

`request.setTimeout(timeout, [callback]);`

> Once a socket is assigned to this request and is connected `socket.setNoDelay()` will be called.

`request.setNoDelay([noDelay]);`
Once a socket is assigned to this request and is connected `socket.setKeepAlive()` will be called.

`request.setSocketKeepAlive([enable], [initialDelay]);`
Emitted when a response is received to this request. This event is emitted only once.

`request.on('response', function(response) { });`
Emitted after a socket is assigned to this request.

`request.on('socket', function(socket) { });`
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.

`request.on('connect', 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.

`request.on('upgrade', function(response, socket, head) { });`
Emitted when the server sends a '100 Continue' HTTP response, usually because the request contained 'Expect: 100-continue'. This is an instruction that the client should send the request body.

HTTP - Requests (cont)

> Once a socket is assigned to this request and is connected `socket.setNoDelay()` will be called.

`request.setNoDelay([noDelay]);`
Once a socket is assigned to this request and is connected `socket.setKeepAlive()` will be called.

`request.setSocketKeepAlive([enable], [initialDelay]);`
Emitted when a response is received to this request. This event is emitted only once.

`request.on('response', function(response) { });`
Emitted after a socket is assigned to this request.

`request.on('socket', function(socket) { });`
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.

`request.on('connect', 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.

`request.on('upgrade', function(response, socket, head) { });`
Emitted when the server sends a '100 Continue' HTTP response, usually because the request contained 'Expect: 100-continue'. This is an instruction that the client should send the request body.



By **camilo.herbert**
cheatography.com/camilo-herbert/

Not published yet.
Last updated 3rd August, 2018.
Page 2 of 7.

Sponsored by **Readable.com**
Measure your website readability!
<https://readable.com>

HTTP - Requests (cont)

```
> request.on('continue', function() {});
```

Global Objects

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

__file name;

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

__dirname;

A reference to the current module. In particular module.exports is used for defining what a module exports and makes available through require().

module;

A reference to the module.exports that is shorter to type.

exports;

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

process;

The Buffer class is a global type for dealing with binary data directly.

Buffer;

Modules

Loads the module module.js in the same directory.

```
var module = require('./module.js');
```

load another module as if require() was called from the module itself.

```
module.require('./another-module.js');
```

Modules (cont)

> The identifier for the module. Typically this is the fully resolved filename.

module.id;

The fully resolved filename to the module.

module.filename;

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

module.loaded;

The module that required this one.

module.parent;

The module objects required by this one.

module.children;

```
exports.area = function(r) {
  return Math.PI * r;
};
```

If you want the root of your module's export to be a function (such as a constructor) or if you want to export a complete object in one assignment instead of building it one property at a time, assign it to module.exports instead of exports.

```
module.exports = function(width) {
  return {
    area: function() {
      return width * width;
    }
  };
};
```

Stream - Writable

```
var writer =
```

```
getWritableStreamSomehow();
```

This method writes some data to the underlying system, and calls the supplied callback once the data has been fully handled.

```
writable.write(chunk,
[encoding], [callback]);
```

If a writable.write(chunk) call returns false, then the drain event will indicate when it is appropriate to begin writing more data to the stream.

```
writer.once('drain', write);
```

Call this method when no more data will be written to the stream.

```
writable.end([chunk],
[encoding], [callback]);
```

When the end() method has been called, and all data has been flushed to the underlying system, this event is emitted.

```
writer.on('finish', function()
{});
```

This is emitted whenever the pipe() method is called on a readable stream, adding this writable to its set of destinations.

```
writer.on('pipe', function(
on(src) {});
```

This is emitted whenever the unpipe() method is called on a readable stream, removing this writable from its set of destinations.

```
writer.on('unpipe', function(
on(src) {});
```

Emitted if there was an error when writing or piping data.

```
writer.on('error', function(
on(src) {});
```

Path

Normalize a string path, taking care of '..' and '.' parts.

```
path.normalize(p);
```

Join all arguments together and normalize the resulting path.

```
path.join([path1], [path2], [...]);
```

Resolves 'to' to an absolute path.

```
path.resolve([from ...], to);
```

Solve the relative path from 'from' to 'to'.

```
path.relative(from, to);
```

Return the directory name of a path. Similar to the Unix dirname command.

```
path.dirname(p);
```

Return the last portion of a path. Similar to the Unix basename command.

```
path.basename(p, [ext]);
```

Return the extension of the path, from the last '.' to end of string in the last portion of the path.

```
path.extname(p);
```

The platform-specific file separator. '\\' or '/'.

```
path.sep;
```

The platform-specific path delimiter, ';' or ':'.

```
path.delimiter;
```

HTTP - Server Events

Emitted each time there is a request.

```
server.on('request', function (request, response) { });
```

When a new TCP stream is established.

```
server.on('connection', function (socket) { });
```

Emitted when the server closes.

```
server.on('close', function () { });
```

Emitted each time a request with an http Expect: 100-continue is received.

```
server.on('checkContinue', function (request, response) { });
```

Emitted each time a client requests a http CONNECT method.

```
server.on('connect', function (request, socket, head) { });
```

Emitted each time a client requests a http upgrade.

```
server.on('upgrade', function (request, socket, head) { });
```

If a client connection emits an 'error' event - it will be forwarded here.

```
server.on('clientError', function (exception, socket) { });
```

HTTP - Responses (cont)

> Sends a HTTP/1.1 100 Continue message to the client, indicating that the request body should be sent.

```
response.writeContinue();
```

Sends a response header to the request.

```
response.writeHead(statusCode, [reasonPhrase], [headers]);
```

Sets the Socket's timeout value to msec. If a callback is provided, then it is added as a listener on the 'timeout' event on the response object.

```
response.setTimeout(msecs, callback);
```

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.setHeader(name, value);
```

Reads out a header that's already been queued but not sent to the client. Note that the name is case insensitive.

```
response.getHeader(name);
```

Removes a header that's queued for implicit sending.

```
response.removeHeader(name);
```

This method adds HTTP trailing headers (a header but at the end of the message) to the response.

```
response.addTrailers(headers);
```

HTTP - Responses

This sends a chunk of the response body. If this method is called and response.writeHead() has not been called, it will switch to implicit header mode and flush the implicit headers.

```
response.write(chunk, [encoding]);
```

HTTP - Responses (cont)

> 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.

`response.end([data], [encoding]);`

When using implicit headers (not calling `response.writeHead()` explicitly), this property controls the status code that will be sent to the client when the headers get flushed.

`response.statusCode;`

Boolean (read-only). True if headers were sent, false otherwise.

`response.headersSent;`

When true, the Date header will be automatically generated and sent in the response if it is not already present in the headers.

Defaults to true.

`response.sendDate;`

Indicates that the underlying connection was terminated before `response.end()` was called or able to flush.

`response.on('close', function () {});`

Emitted when the response has been sent.

`response.on('finish', function() {});`

Console

Prints to stdout with newline.

`console.log([data], [...]);`

Same as `console.log`.

`console.info([data], [...]);`

Same as `console.log` but prints to stderr.

`console.error([data], [...]);`

Same as `console.error`.

`console.warn([data], [...]);`

Uses `util.inspect` on obj and prints resulting string to stdout.

`console.dir(obj);`

Mark a time.

`console.time(label);`

Finish timer, record output.

`console.timeEnd(label);`

Print a stack trace to stderr of the current position.

`console.assert(expression);`

Same as `assert.ok()` where if the expression evaluates as false throw an `AssertionError` with message.

`console.assert(expression, [message]);`

Process

Emitted when the process is about to exit

`process.on('exit', function (code) {});`

Emitted when an exception bubbles all the way back to the event loop. (should not be used)

`process.on('uncaughtException', function(err) {});`

A writable stream to stdout.

`process.stdout;`

Process (cont)

> A writable stream to stderr.

`process.stderr;`

A readable stream for stdin.

`process.stdin;`

An array containing the command line arguments.

`process.argv;`

An object containing the user environment.

`process.env;`

This is the absolute pathname of the executable that started the process.

`process.execPath;`

This is the set of node-specific command line options from the executable that started the process.

`process.execArgv;`

Stream - Readable

`var readable =`

`getReadableStreamSomehow();`

When a chunk of data can be read from the stream, it will emit a 'readable' event.

`readable.on('readable', function() {});`

If you attach a data event listener, then it will switch the stream into flowing mode, and data will be passed to your handler as soon as it is available.

`readable.on('data', function (chunk) {});`

This event fires when there will be no more data to read.

`readable.on('end', function() {});`

Stream - Readable (cont)

> Emitted when the underlying resource (for example, the backing file descriptor) has been closed. Not all streams will emit this.

```
readable.on('close', function() {});
```

Emitted if there was an error receiving data.

```
readable.on('error', function() {});
```

The `read()` method pulls some data out of the internal buffer and returns it. If there is no data available, then it will return null. This method should only be called in non-flowing mode. In flowing-mode, this method is called automatically until the internal buffer is drained.

```
readable.read([size]);
```

Call this function to cause the stream to return strings of the specified encoding instead of Buffer objects.

```
readable.setEncoding(encoding);
```

This method will cause the readable stream to resume emitting data events.

```
readable.resume();
```

This method will cause a stream in flowing-mode to stop emitting data events.

```
readable.pause();
```

This method pulls all the data out of a readable stream, and writes it to the supplied destination, automatically managing the flow so that the destination is not overwhelmed by a fast readable stream.

```
readable.pipe(destination, [options]);
```

Stream - Readable (cont)

> This method will remove the hooks set up for a previous `pipe()` call. If the destination is not specified, then all pipes are removed.

```
readable.unpipe([destination]);
```

This is useful in certain cases where a stream is being consumed by a parser, which needs to "un-consume" some data that it has optimistically pulled out of the source, so that the stream can be passed on to some other party.

```
readable.unshift(chunk);
```

HTTP

A collection of all the standard HTTP response status codes, and the short description of each.

```
http.STATUS_CODES;
```

This function allows one to transparently issue requests.

```
http.request(options, [callback]);
```

Set the method to GET and calls `req.end()` automatically.

```
http.get(options, [callback]);
```

Returns a new web server object. The `requestListener` is a function which is automatically added to the 'request' event.

```
server = http.createServer([requestListener]);
```

Begin accepting connections on the specified port and hostname.

```
server.listen(port, [hostname], [backlog], [callback]);
```

HTTP (cont)

> Start a UNIX socket server listening for connections on the given path.

```
server.listen(path, [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.listen(handle, [callback]);
```

Stops the server from accepting new connections.

```
server.close([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.setTimeout(msecs, callback);
```

Limits maximum incoming headers count, equal to 1000 by default. If set to 0 - no limit will be applied.

```
server.maxHeadersCount;
```

The number of milliseconds of inactivity before a socket is presumed to have timed out.

```
server.timeout;
```

HTTP - Messages

In case of server request, the HTTP version sent by the client.

In the case of client response, the HTTP version of the connected-to server.

```
message.httpVersion;
```

The request/response headers object.

```
message.headers;
```

The request/response trailers object. Only populated after the 'end' event.



By **camilo.herbert**
cheatography.com/camilo-herbert/

Not published yet.
 Last updated 3rd August, 2018.
 Page 6 of 7.

Sponsored by **Readable.com**
 Measure your website readability!
<https://readable.com>

HTTP - Messages (cont)

> message.trailers;

The request method as a string. Read only.

Example: 'GET', 'DELETE'.

message.method;

Request URL string. This contains only the URL that is present in the actual HTTP request.

message.url;

The 3-digit HTTP response status code.

E.G. 404.

message.statusCode;

The net.Socket object associated with the connection.

message.socket;

Calls message.connection.setTimeout(-msecs, callback).

message.setTimeout(msecs, callback);



By **camilo.herbert**

cheatography.com/camilo-herbert/

Not published yet.

Last updated 3rd August, 2018.

Page 7 of 7.

Sponsored by **Readable.com**

Measure your website readability!

<https://readable.com>