Node.js in Action

by Mike Cantelon
Marc Harter
T.J. Holowaychuk
Nathan Rajlich

Chapter 4

brief contents

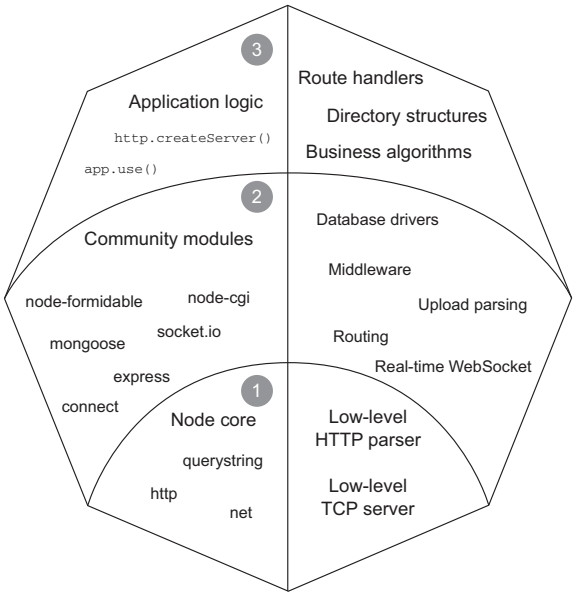
4 Building Node web applications

This chapter covers

- Handling HTTP requests with Node's API
- Building a RESTful web service
- Serving static files
- Accepting user input from forms
- Securing your application with HTTPS

In this chapter, you'll become familiar with the tools Node provides for creating HTTP servers, and you'll get acquainted with the fs (filesystem) module, which is necessary for serving static files. You'll also learn how to handle other common web application needs, such as creating low-level RESTful web services, accepting user input through HTML forms, monitoring file upload progress, and securing a web application with Node's Secure Sockets Layer (SSL).

At Node's core is a powerful streaming HTTP parser consisting of roughly 1,500 lines of optimized C, written by the author of Node, Ryan Dahl. This parser, in combination with the low-level TCP API that Node exposes to JavaScript, provides you with a very low-level, but very flexible, HTTP server.



1 Node's core APIs are always lightweight and low-level. This leaves opinions, syntactic sugar, and specific details up to the community modules.

2 Community modules are where Node thrives. Community members take the low-level core APIs and create fun and easy-to-use modules that allow you to get tasks done easily.

3 The application logic layer is where your app is implemented. The size of this layer depends on the number of community modules used and the complexity of the application.

Figure 4.1 Overview of the layers that make up a Node web application

Like most modules in Node's core, the http module favors simplicity. High-level "sugar" APIs are left for third-party frameworks, such as Connect or Express, that greatly simplify the web application building process. Figure 4.1 illustrates the anatomy of a Node web application, showing that the low-level APIs remain at the core, and that abstractions and implementations are built on top of those building blocks.

This chapter will cover some of Node's low-level APIs directly. You can safely skip this chapter if you're more interested in higher-level concepts and web frameworks, like Connect or Express, which will be covered in later chapters. But before creating rich web applications with Node, you'll need to become familiar with the fundamental HTTP API, which can be built upon to create higher-level tools and frameworks.

4.1 HTTP server fundamentals

As we've mentioned throughout this book, Node has a relatively low-level API. Node's HTTP interface is similarly low-level when compared with frameworks or languages such as PHP in order to keep it fast and flexible.

To get you started creating robust and performant web applications, this section will focus on the following topics:

- How Node presents incoming HTTP requests to developers
- How to write a basic HTTP server that responds with "Hello World"
- How to read incoming request headers and set outgoing response headers
- How to set the status code of an HTTP response

Before you can accept incoming requests, you need to create an HTTP server. Let's take a look at Node's HTTP interface.

4.1.1 How Node presents incoming HTTP requests to developers

Node provides HTTP server and client interfaces through the http module:

```
var http = require('http');
```

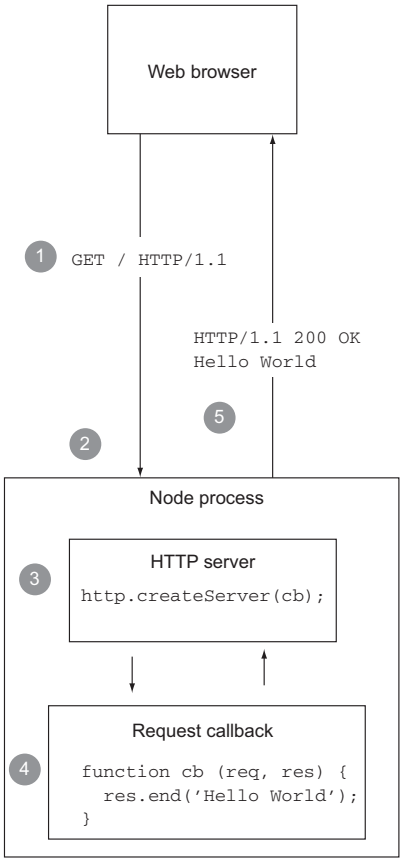
To create an HTTP server, call the `http.createServer()` function. It accepts a single argument, a callback function, that will be called on each HTTP request received by the server. This *request* callback receives, as arguments, the request and response objects, which are commonly shortened to `req` and `res`:

```
var http = require('http');
  var server = http.createServer(function(req, res){
  // handle request
  });
```

For every HTTP request received by the server, the request callback function will be invoked with new `req` and `res` objects. Prior to the callback being triggered, Node will parse the request up through the HTTP headers and provide them as part of the `req` object. But Node doesn't start parsing the body of the request until the callback has been fired. This is different from some server-side frameworks, like PHP, where both the headers and the body of the request are parsed before your application logic runs. Node provides this lower-level interface so you can handle the body data as it's being parsed, if desired.

Node will not automatically write any response back to the client. After the request callback is triggered, it's your responsibility to end the response using the `res.end()` method (see figure 4.2). This allows you to run any asynchronous logic you want during the lifetime of the request before ending the response. If you fail to end the response, the request will hang until the client times out or it will just remain open.

Node servers are long-running processes that serve many requests throughout their lifetimes.



1. An HTTP client, like a web browser, initiates an HTTP request.
2. Node accepts the connection, and incoming request data is given to the HTTP server.
3. The HTTP server parses up to the end of the HTTP headers and then hands control over to the request callback.
4. The request callback performs application logic, in this case responding immediately with the text "Hello World."
5. The request is sent back through the HTTP server, which formats a proper HTTP response for the client.

Figure 4.2 The lifecycle of an HTTP request going through a Node HTTP server

4.1.2 A basic HTTP server that responds with "Hello World"

To implement a simple Hello World HTTP server, let's flesh out the request callback function from the previous section.

First, call the `res.write()` method, which writes response data to the socket, and then use the `res.end()` method to end the response:

```
var http = require('http');
var server = http.createServer(function(req, res){
  res.write('Hello World');
  res.end();
});
```

As shorthand, `res.write()` and `res.end()` can be combined into one statement, which can be nice for small responses:

```
res.end('Hello World');
```

The last thing you need to do is bind to a port so you can listen for incoming requests. You do this by using the `server.listen()` method, which accepts a combination of

arguments, but for now the focus will be on listening for connections on a specified port. During development, it's typical to bind to an unprivileged port, such as 3000:

```
var http = require('http');
var server = http.createServer(function(req, res){
  res.end('Hello World');
});
server.listen(3000);
```

With Node now listening for connections on port 3000, you can visit http://localhost:3000 in your browser. When you do, you should receive a plain-text page consisting of the words "Hello World."

Setting up an HTTP server is just the start. You'll need to know how to set response status codes and header fields, handle exceptions appropriately, and use the APIs Node provides. First we'll take a closer look at responding to incoming requests.

4.1.3 Reading request headers and setting response headers

The Hello World example in the previous section demonstrates the bare minimum required for a proper HTTP response. It uses the default status code of 200 (indicating success) and the default response headers. Usually, though, you'll want to include any number of other HTTP headers with the response. For example, you'll have to send a `Content-Type` header with a value of `text/html` when you're sending HTML content so that the browser knows to render the result as HTML.

Node offers several methods to progressively alter the header fields of an HTTP response: the `res.setHeader(field, value)`, `res.getHeader(field)`, and `res.removeHeader(field)` methods. Here's an example of using `res.setHeader()`:

```
var body = 'Hello World';
res.setHeader('Content-Length', body.length);
res.setHeader('Content-Type', 'text/plain');
res.end(body);
```

You can add and remove headers in any order, *but* only up to the first `res.write()` or `res.end()` call. After the first part of the response body is written, Node will flush the HTTP headers that have been set.

4.1.4 Setting the status code of an HTTP response

It's common to want to send back a different HTTP status code than the default of 200. A common case would be sending back a 404 Not Found status code when a requested resource doesn't exist.

To do this, you set the `res.statusCode` property. This property can be assigned at any point during the application's response, as long as it's before the first call to `res.write()` or `res.end()`. As shown in the following example, this means `res.statusCode = 302` can be placed above the `res.setHeader()` calls, or below them:

```
var url = 'http://google.com';
var body = '<p>Redirecting to <a href="' + url + '">'
         + url + '</a></p>';

res.setHeader('Location', url);
res.setHeader('Content-Length', body.length);
res.setHeader('Content-Type', 'text/html');
res.statusCode = 302;
res.end(body);
```

Node's philosophy is to provide small but robust networking APIs, not to compete with high-level frameworks such as Rails or Django, but to serve as a tremendous platform for similar frameworks to build upon. Because of this design, neither high-level concepts like sessions nor fundamentals such as HTTP cookies are provided within Node's core. Those are left for third-party modules to provide.

Now that you've seen the basic HTTP API, it's time to put it to use. In the next section, you'll make a simple, HTTP-compliant application using this API.

4.2 Building a RESTful web service

Suppose you want to create a to-do list web service with Node, involving the typical create, read, update, and delete (CRUD) actions. These actions can be implemented in many ways, but in this section we'll focus on creating a RESTful web service—a service that utilizes the HTTP method verbs to expose a concise API.

In 2000, representational state transfer (REST) was introduced by Roy Fielding,[1] one of the prominent contributors to the HTTP 1.0 and 1.1 specifications. By convention, HTTP verbs, such as `GET`, `POST`, `PUT`, and `DELETE`, are mapped to retrieving, creating, updating, and removing the resources specified by the URL. RESTful web services have gained in popularity because they're simple to utilize and implement in comparison to protocols such as the Simple Object Access Protocol (SOAP).

Throughout this section, cURL (http://curl.haxx.se/download.html) will be used, in place of a web browser, to interact with your web service. cURL is a powerful command-line HTTP client that can be used to send requests to a target server.

To create a compliant REST server, you need to implement the four HTTP verbs. Each verb will cover a different task for the to-do list:

- `POST`—Add items to the to-do list
- `GET`—Display a listing of the current items, or display the details of a specific item
- `DELETE`—Remove items from the to-do list
- `PUT`—Should modify existing items, but for brevity's sake we'll skip `PUT` in this chapter

To illustrate the end result, here's an example of creating a new item in the to-do list using the `curl` command:

[1] Roy Thomas Fielding, "Architectural Styles and the Design of Network-based Software Architectures" (PhD diss, University of California, Irvine, 2000), www.ics.uci.edu/~fielding/pubs/dissertation/top.htm.

```
wavded@dev: ~
wavded@dev ~» curl -d 'buy node in action' http://localhost:3000
OK
```

And here's an example of viewing the items in the to-do list:

```
wavded@dev: ~
wavded@dev ~» curl http://localhost:3000
0) buy node in action
```

4.2.1 Creating resources with POST requests

In RESTful terminology, the creation of a resource is typically mapped to the `POST` verb. Therefore, `POST` will create an entry in the to-do list.

In Node, you can check which HTTP method (verb) is being used by checking the `req.method` property (as shown in listing 4.1). When you know which method the request is using, your server will know which task to perform.

When Node's HTTP parser reads in and parses request data, it makes that data available in the form of `data` events that contain chunks of parsed data ready to be handled by the program:

```
var http = require('http')
var server = http.createServer(function(req, res){
  req.on('data', function(chunk){
    console.log('parsed', chunk);
  });
  req.on('end', function(){
    console.log('done parsing');
    res.end()
  });
});
```

Data events are fired whenever a new chunk of data has been read.

A chunk, by default, is a Buffer object (a byte array).

The end event is fired when everything has been read.

By default, the `data` events provide `Buffer` objects, which are Node's version of byte arrays. In the case of textual to-do items, you don't need binary data, so setting the stream encoding to `ascii` or `utf8` is ideal; the `data` events will instead emit strings. This can be set by invoking the `req.setEncoding(encoding)` method:

```
req.setEncoding('utf8')
req.on('data', function(chunk){
  console.log(chunk);
});
```

A chunk is now a utf8 string instead of a Buffer.

In the case of a to-do list item, you need to have the entire string before it can be added to the array. One way to get the whole string is to concatenate all of the chunks of data until the `end` event is emitted, indicating that the request is complete. After the `end` event has occurred, the `item` string will be populated with the entire contents of the request body, which can then be pushed to the `items` array. When the item has been added, you can end the request with the string `OK` and Node's default status code of 200. The following listing shows this in the todo.js file.

Listing 4.1 POST request body string buffering

```
var http = require('http');
var url = require('url');
var items = [];

var server = http.createServer(function(req, res){
  switch (req.method) {
    case 'POST':
      var item = '';
      req.setEncoding('utf8');
      req.on('data', function(chunk){
        item += chunk;
      });
      req.on('end', function(){
        items.push(item);
        res.end('OK\n');
      });
      break;
  }
});
```

The data store is a regular JavaScript Array in memory.

req.method is the HTTP method requested.

Set up string buffer for the incoming item.

Encode incoming data events as UTF-8 strings.

Concatenate data chunk onto the buffer.

Push complete new item onto the items array.

Figure 4.3 illustrates the HTTP server handling an incoming HTTP request and buffering the input before acting on the request at the end.

The application can now add items, but before you try it out using cURL, you should complete the next task so you can get a listing of the items as well.

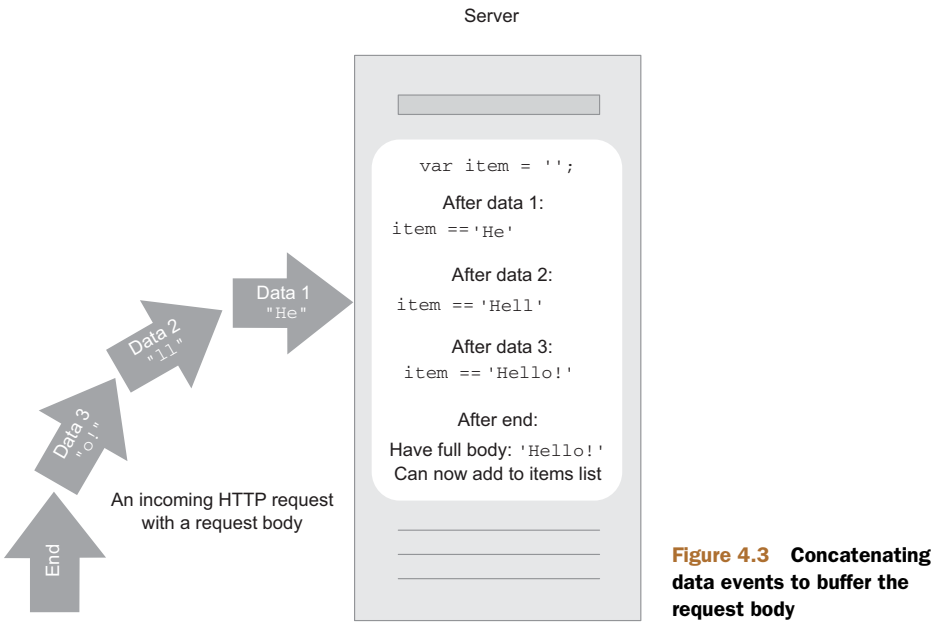


Figure 4.3 Concatenating data events to buffer the request body

4.2.2 Fetching resources with GET requests

To handle the `GET` verb, add it to the same `switch` statement as before, followed by the logic for listing the to-do items. In the following example, the first call to `res.write()` will write the header with the default fields, as well as the data passed to it:

```
...
case 'GET':
items.forEach(function(item, i){
  res.write(i + ') ' + item + '\n');
});
res.end();
break;
...
```

Now that the app can display the items, it's time to give it a try! Fire up a terminal, start the server, and `POST` some items using `curl`. The `-d` flag automatically sets the request method to `POST` and passes in the value as `POST` data:

```
$ curl -d 'buy groceries' http://localhost:3000
OK
$ curl -d 'buy node in action' http://localhost:3000
OK
```

Next, to `GET` the list of to-do list items, you can execute `curl` without any flags, as `GET` is the default verb:

```
$ curl http://localhost:3000
0) buy groceries
1) buy node in action
```

SETTING THE CONTENT-LENGTH HEADER

To speed up responses, the `Content-Length` field should be sent with your response when possible. In the case of the item list, the body can easily be constructed ahead of time in memory, allowing you to access the string length and flush the entire list in one shot. Setting the `Content-Length` header implicitly disables Node's chunked encoding, providing a performance boost because less data needs to be transferred.

An optimized version of the `GET` handler could look something like this:

```
var body = items.map(function(item, i){
  return i + ') ' + item;
}).join('\n');
res.setHeader('Content-Length', Buffer.byteLength(body));
res.setHeader('Content-Type', 'text/plain; charset="utf-8"');
res.end(body);
```

You may be tempted to use the `body.length` value for the `Content-Length`, but the `Content-Length` value should represent the byte length, not character length, and the two will be different if the string contains multibyte characters. To avoid this problem, Node provides the `Buffer.byteLength()` method.

The following Node REPL session illustrates the difference by using the string `length` directly, as the five-character string is comprised of seven bytes:

```
$ node
> 'etc …'.length
5
> Buffer.byteLength('etc …')
7
```

The Node REPL

Node, like many other languages, provides a REPL (read-eval-print-loop) interface, available by running `node` from the command line without any arguments. A REPL allows you to write snippets of code and to get immediate results as each statement is written and executed. It can be great for learning a programming language, running simple tests, or even debugging.

4.2.3 Removing resources with DELETE requests

Finally, the `DELETE` verb will be used to remove an item. To accomplish this, the app will need to check the requested URL, which is how the HTTP client will specify which item to remove. In this case, the identifier will be the array index in the `items` array; for example, `DELETE /1` or `DELETE /5`.

The requested URL can be accessed with the `req.url` property, which may contain several components depending on the request. For example, if the request was `DELETE /1?api-key=foobar`, this property would contain both the pathname and query string `/1?api-key=foobar`.

To parse these sections, Node provides the url module, and specifically the `.parse()` function. The following node REPL session illustrates the use of this function, parsing the URL into an object, including the `pathname` property you'll use in the `DELETE` handler:

```
$ node
> require('url').parse('http://localhost:3000/1?api-key=foobar')
{ protocol: 'http:',
  slashes: true,
  host: 'localhost:3000',
  port: '3000',
  hostname: 'localhost',
  href: 'http://localhost:3000/1?api-key=foobar',
  search: '?api-key=foobar',
  query: 'api-key=foobar',
  pathname: '/1',
  path: '/1?api-key=foobar' }
```

`url.parse()` parses out only the pathname for you, but the item ID is still a string. In order to work with the ID within the application, it should be converted to a number. A simple solution is to use the `String#slice()` method, which returns a portion of the string between two indexes. In this case, it can be used to skip the first character, giving you just the number portion, still as a string. To convert this string to a number, it can be passed to the JavaScript global function `parseInt()`, which returns a `Number`.

Listing 4.2 first does a couple of checks on the input value, because you can never trust user input to be valid, and then it responds to the request. If the number is "not a number" (the JavaScript value `NaN`), the status code is set to 400 indicating a Bad Request. Following that, the code checks if the item exists, responding with a 404 Not Found error if it doesn't. After the input has been validated, the item can be removed from the `items` array, and then the app will respond with 200, OK.

Listing 4.2 `DELETE` request handler

```
...
case 'DELETE':
  var path = url.parse(req.url).pathname;
  var i = parseInt(path.slice(1), 10);

  if (isNaN(i)) {
    res.statusCode = 400;
    res.end('Invalid item id');
  } else if (!items[i]) {
    res.statusCode = 404;
    res.end('Item not found');
  } else {
    items.splice(i, 1);
    res.end('OK\n');
  }
  break;
...
```

Add DELETE case to the switch statement

Check that number is valid

Ensure requested index exists

Delete requested item

You might be thinking that 15 lines of code to remove an item from an array is a bit much, but we promise that this is much easier to write with higher-level frameworks providing additional sugar APIs. Learning these fundamentals of Node is crucial for understanding and debugging, and it enables you to create more powerful applications and frameworks.

A complete RESTful service would also implement the `PUT` HTTP verb, which should modify an existing item in the to-do list. We encourage you to try implementing this final handler yourself, using the techniques used in this REST server so far, before you move on to the next section, in which you'll learn how to serve static files from your web application.

4.3 Serving static files

Many web applications share similar, if not identical, needs, and serving static files (CSS, JavaScript, images) is certainly one of these. Although writing a robust and efficient static file server is nontrivial, and robust implementations already exist within Node's community, implementing your own static file server in this section will illustrate Node's low-level filesystem API.

In this section you'll learn how to

- Create a simple static file server
- Optimize the data transfer with `pipe()`
- Handle user and filesystem errors by setting the status code

Let's start by creating a basic HTTP server for serving static assets.

4.3.1 Creating a static file server

Traditional HTTP servers like Apache and IIS are first and foremost file servers. You might currently have one of these file servers running on an old website, and moving it over to Node, replicating this basic functionality, is an excellent exercise to help you better understand the HTTP servers you've probably used in the past.

Each static file server has a root directory, which is the base directory files are served from. In the server you'll create, you'll define a `root` variable, which will act as the static file server's root directory:

```
var http = require('http');
var parse = require('url').parse;
var join = require('path').join;
var fs = require('fs');

var root = __dirname;

...
```

`__dirname` is a *magic* variable provided by Node that's assigned the directory path to the file. It's magic because it could be assigned different values in the same program if you have files spread about in different directories. In this case, the server will be serving static files relative to the same directory as this script, but you could configure `root` to specify any directory path.

The next step is accessing the `pathname` of the URL in order to determine the requested file's path. If a URL's `pathname` is /index.html, and your `root` file directory is /var/www/example.com/public, you can simply join these using the path module's `.join()` method to form the absolute path /var/www/example.com/public/index.html. The following code shows how this could be done:

```
var http = require('http');
var parse = require('url').parse;
var join = require('path').join;
var fs = require('fs');

var root = __dirname;

var server = http.createServer(function(req, res){
  var url = parse(req.url);
  var path = join(root, url.pathname);
});

server.listen(3000);
```

Directory traversal attack

The file server built in this section is a simplified one. If you want to run this in production, you should validate the input more thoroughly to prevent users from getting access to parts of the filesystem you don't intend them to via a directory traversal attack. Wikipedia has an explanation of how this type of attack works (http://en.wikipedia.org/wiki/Directory_traversal_attack).

Now that you have the path, the contents of the file need to be transferred. This can be done using high-level streaming disk access with `fs.ReadStream`, one of Node's `Stream` classes. This class emits data events as it incrementally reads the file from disk. The next listing implements a simple but fully functional file server.

Listing 4.3 Bare-bones `ReadStream` static file server

```
var http = require('http');
var parse = require('url').parse;
var join = require('path').join;
var fs = require('fs');

var root = __dirname;

var server = http.createServer(function(req, res){
  var url = parse(req.url);
  var path = join(root, url.pathname);
  var stream = fs.createReadStream(path);
  stream.on('data', function(chunk){
    res.write(chunk);
  });
  stream.on('end', function(){
    res.end();
  });
});

server.listen(3000);
```

Construct absolute path

Create fs.ReadStream

Write file data to response

End response when file is complete

This file server would work in most cases, but there are many more details you'll need to consider. Next up, you'll learn how to optimize the data transfer while making the code for the server even shorter.

Optimizing data transfer with Stream#pipe()

Although it's important to know how the `fs.ReadStream` works and what flexibility its events provide, Node also provides a higher-level mechanism for performing the same task: `Stream#pipe()`. This method allows you to greatly simplify your server code.

Pipes and plumbing

A helpful way to think about pipes in Node is to think about plumbing. If you have water coming from a source (such as a water heater) and you want to direct it to a destination (like a kitchen faucet), you can route that water from its source to its destination by adding a pipe to connect the two. Water can then flow from the source *through the pipe* to the destination.

The same concept is true for pipes in Node, but instead of water you're dealing with data coming from a source (called a `ReadableStream`) that you can then "pipe" to some destination (called a `WritableStream`). You hook up the plumbing with the `pipe` method:

```
ReadableStream#pipe(WritableStream);
```

(continued)

An example of using pipes is reading a file (`ReadableStream`) and writing its contents to another file (`WritableStream`):

```
var readStream = fs.createReadStream('./original.txt')
var writeStream = fs.createWriteStream('./copy.txt')
readStream#pipe(writeStream);
```

Any `ReadableStream` can be piped into any `WritableStream`. For example, an HTTP request (`req`) object is a `ReadableStream`, and you can stream its contents to a file:

```
req.pipe(fs.createWriteStream('./req-body.txt'))
```

For an in-depth look at streams in Node, including a list of available built-in streams, check out the stream handbook on GitHub: https://github.com/substack/stream-handbook.

```
var server = http.createServer(function(req, res){
  var url = parse(req.url);
  var path = join(root, url.pathname);
  var stream = fs.createReadStream(path);
  stream.pipe(res);
});
```

res.end() called internally by stream.pipe()

Figure 4.4 shows an HTTP server in the act of reading a static file from the filesystem and then piping the result to the HTTP client using `pipe()`.

At this point, you can test to confirm that the static file server is functioning by executing the following `curl` command. The `-i`, or `--include` flag, instructs cURL to output the response header:

```
$ curl http://localhost:3000/static.js -i
HTTP/1.1 200 OK
Connection: keep-alive
Transfer-Encoding: chunked

var http = require('http');
var parse = require('url').parse;
var join = require('path').join;
...
```

As previously mentioned, the root directory used is the directory that the static file server script is in, so the preceding `curl` command requests the server's script itself, which is sent back as the response body.

This static file server isn't complete yet, though—it's still prone to errors. A single unhandled exception, such as a user requesting a file that doesn't exist, will bring down your entire server. In the next section, you'll add error handling to the file server.

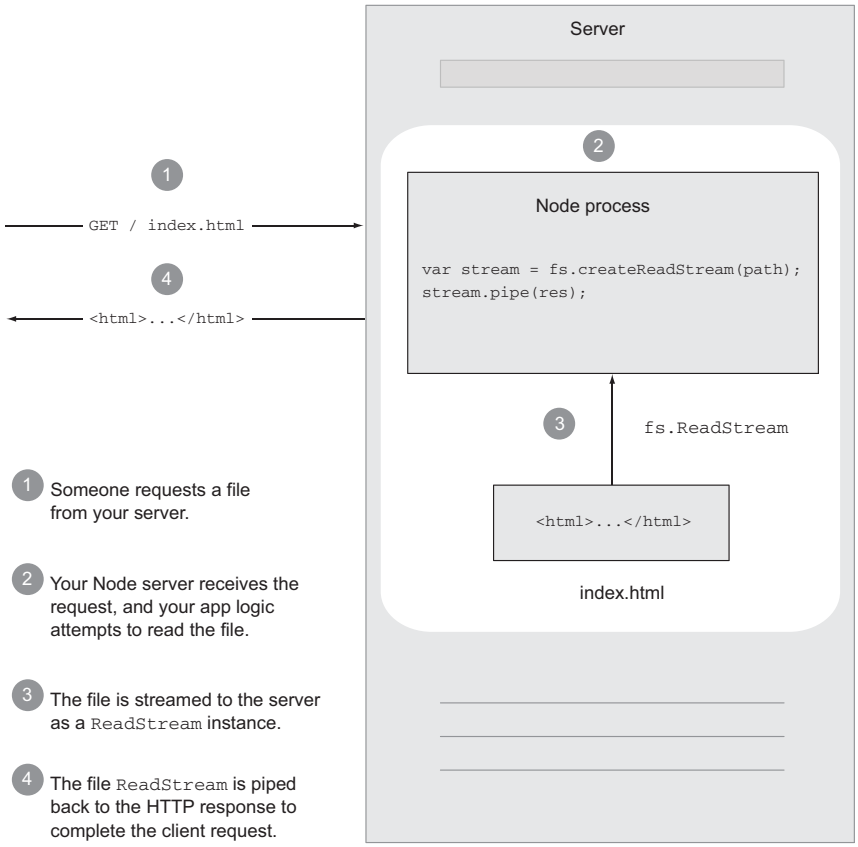


Figure 4.4 A Node HTTP server serving a static file from the filesystem using `fs.ReadStream`

4.3.2 Handling server errors

Our static file server is not yet handling errors that could occur as a result of using `fs.ReadStream`. Errors will be thrown in the current server if you access a file that doesn't exist, access a forbidden file, or run into any other file I/O–related problem. In this section, we'll touch on how you can make the file server, or any Node server, more robust.

In Node, anything that inherits from `EventEmitter` has the potential of emitting an `error` event. A stream, like `fs.ReadStream`, is simply a specialized `EventEmitter` that contains predefined events such as `data` and `end`, which we've already looked at. By default, `error` events will be thrown when no listeners are present. This means that if you don't listen for these errors, they'll crash your server.

To illustrate this, try requesting a file that doesn't exist, such as /notfound.js. In the terminal session running your server, you'll see the stack trace of an exception printed to stderr, similar to the following:

```
stream.js:99
throw arguments[1]; // Unhandled 'error' event.
^
Error: ENOENT, No such file or directory
  ➥'/Users/tj/projects/node-in-action/source/notfound.js'
```

To prevent errors from killing the server, you need to listen for errors by registering an `error` event handler on the `fs.ReadStream` (something like the following snippet), which responds with the 500 response status indicating an internal server error:

```
...
stream.pipe(res);
stream.on('error', function(err){
  res.statusCode = 500;
  res.end('Internal Server Error');
});
...
```

Registering an `error` event helps you catch any foreseen or unforeseen errors and enables you to respond more gracefully to the client.

4.3.3 Preemptive error handling with fs.stat

The files transferred are static, so the `stat()` system call can be utilized to request information about the files, such as the modification time, byte size, and more. This information is especially important when providing conditional `GET` support, where a browser may issue a request to check if its cache is stale.

The refactored file server shown in listing 4.4 makes a call to `fs.stat()` and retrieves information about a file, such as its size, or an error code. If the named file doesn't exist, `fs.stat()` will respond with a value of `ENOENT` in the `err.code` field, and you can return the error code 404, indicating that the file is not found. If you receive other errors from `fs.stat()`, you can return a generic 500 error code.

Listing 4.4 Checking for a file's existence and responding with `Content-Length`

```
var server = http.createServer(function(req, res){
  var url = parse(req.url);
  var path = join(root, url.pathname);
  fs.stat(path, function(err, stat){
    if (err) {
      if ('ENOENT' == err.code) {
        res.statusCode = 404;
        res.end('Not Found');
      } else {
        res.statusCode = 500;
        res.end('Internal Server Error');
      }
    } else {
      res.setHeader('Content-Length', stat.size);
      var stream = fs.createReadStream(path);
      stream.pipe(res);
      stream.on('error', function(err){
```

Construct absolute path

File doesn't exist

Parse URL to obtain path name

Check for file's existence

Some other error

Set Content-Length using stat object

```
        res.statusCode = 500;
        res.end('Internal Server Error');
      });
    }
  });
});
```

Now that we've taken a low-level look at file serving with Node, let's take a look at an equally common, and perhaps more important, feature of web application development: getting user input from HTML forms.

4.4 Accepting user input from forms

Web applications commonly gather user input through form submissions. Node doesn't handle the workload (like validation or file uploads) for you—Node just provides you with the body data. Although this may seem inconvenient, it leaves opinions to third-party frameworks in order to provide a simple and efficient low-level API.

In this section, we'll take a look at how you can do the following:

- Handle submitted form fields
- Handle uploaded files using formidable
- Calculate upload progress in real time

Let's dive into how you process incoming form data using Node.

4.4.1 Handling submitted form fields

Typically two `Content-Type` values are associated with form submission requests:

- `application/x-www-form-urlencoded`—The default for HTML forms
- `multipart/form-data`—Used when the form contains files, or non-ASCII or binary data

In this section, you'll rewrite the to-do list application from the previous section to utilize a form and a web browser. When you're done, you'll have a web-based to-do list that looks like the one in figure 4.5.

In this to-do list application, a `switch` is used on the request method, `req.method`, to form simple request routing. This is shown in listing 4.5. Any URL that's not *exactly* "/" is considered a 404 Not Found response. Any HTTP verb that is not `GET` or `POST` is

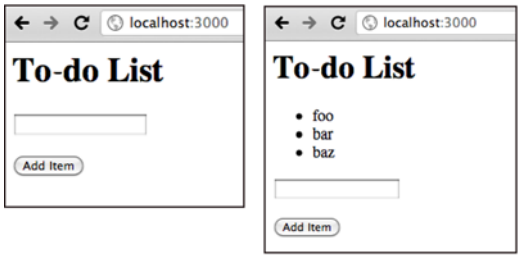


Figure 4.5 A to-do-list application utilizing an HTML form and a web browser. The left screenshot shows the state of the application when it's first loaded and the right shows what the applications looks like after some items have been added.

a 400 Bad Request response. The handler functions `show()`, `add()`, `badRequest()`, and `notFound()` will be implemented throughout the rest of this section.

Listing 4.5 HTTP server supporting GET and POST

```
var http = require('http');
var items = [];

var server = http.createServer(function(req, res){
  if ('/' == req.url) {
    switch (req.method) {
      case 'GET':
        show(res);
        break;
      case 'POST':
        add(req, res);
        break;
      default:
        badRequest(res);
    }
  } else {
    notFound(res);
  }
});

server.listen(3000);
```

Although markup is typically generated using template engines, the example in the following listing uses string concatenation for simplicity. There's no need to assign `res.statusCode` because it defaults to 200 OK. The resulting HTML page in a browser is shown in figure 4.5.

Listing 4.6 To-do list form and item list

```
function show(res) {
  var html = '<html><head><title>Todo List</title></head><body>'
           + '<h1>Todo List</h1>'
           + '<ul>'
           + items.map(function(item){
               return '<li>' + item + '</li>'
             }).join('')
           + '</ul>'
           + '<form method="post" action="/">'
           + '<p><input type="text" name="item" /></p>'
           + '<p><input type="submit" value="Add Item" /></p>'
           + '</form></body></html>';
  res.setHeader('Content-Type', 'text/html');
  res.setHeader('Content-Length', Buffer.byteLength(html));
  res.end(html);
}
```

For simple apps, inlining the HTML instead of using a template engine works well.

The `notFound()` function accepts the response object, setting the status code to 404 and response body to Not Found:

```
function notFound(res) {
  res.statusCode = 404;
  res.setHeader('Content-Type', 'text/plain');
  res.end('Not Found');
}
```

The implementation of the 400 Bad Request response is nearly identical to `notFound()`, indicating to the client that the request was invalid:

```
function badRequest(res) {
  res.statusCode = 400;
  res.setHeader('Content-Type', 'text/plain');
  res.end('Bad Request');
}
```

Finally, the application needs to implement the `add()` function, which will accept both the `req` and `res` objects. This is shown in the following code:

```
var qs = require('querystring');

function add(req, res) {
  var body = '';
  req.setEncoding('utf8');
  req.on('data', function(chunk){ body += chunk });
  req.on('end', function(){
    var obj = qs.parse(body);
    items.push(obj.item);
    show(res);
  });
}
```

For simplicity, this example assumes that the `Content-Type` is `application/x-www-form-urlencoded`, which is the default for HTML forms. To parse this data, you simply concatenate the `data` event chunks to form a complete body string. Because you're not dealing with binary data, you can set the request encoding type to `utf8` with `res.setEncoding()`. When the request emits the `end` event, all `data` events have completed, and the `body` variable contains the entire body as a string.

Buffering too much data

Buffering works well for small request bodies containing a bit of JSON, XML, and the like, but the buffering of this data can be problematic. It can create an application availability vulnerability if the buffer isn't properly limited to a maximum size, which we'll discuss further in chapter 7. Because of this, it's often beneficial to implement a streaming parser, lowering the memory requirements and helping prevent resource starvation. This process incrementally parses the data chunks as they're emitted, though this is more difficult to use and implement.

THE QUERYSTRING MODULE

In the server's `add()` function implementation, you utilized Node's querystring module to parse the body. Let's take a look at a quick REPL session demonstrating how Node's `querystring.parse()` function works—this is the function used in the server.

Imagine the user submitted an HTML form to your to-do list with the text "take ferrets to the vet":

```
$ node
> var qs = require('querystring');
> var body = 'item=take+ferrets+to+the+vet';
> qs.parse(body);
{ item: 'take ferrets to the vet' }
```

After adding the item, the server returns the user back to the original form by calling the same `show()` function previously implemented. This is only the route taken for this example; other approaches could potentially display a message such as "Added to-do list item" or could redirect the user back to `/`.

Try it out. Add a few items and you'll see the to-do items output in the unordered list. You can also implement the delete functionality that we did in the REST API previously.

4.4.2 Handling uploaded files using formidable

Handling uploads is another very common, and important, aspect of web development. Imagine you're trying to create an application where you upload your photo collection and share it with others using a link on the web. You can do this using a web browser through HTML form file uploads.

The following example shows a form that uploads a file with an associated `name` field:

```
<form method="post" action="/" enctype="multipart/form-data">
<p><input type="text" name="name" /></p>
<p><input type="file" name="file" /></p>
<p><input type="submit" value="Upload" /></p>
</form>
```

To handle file uploads properly and accept the file's content, you need to set the `enctype` attribute to `multipart/form-data`, a MIME type suited for BLOBs (binary large objects).

Parsing multipart requests in a performant and streaming fashion is a nontrivial task, and we won't cover the details in this book, but Node's community has provided several modules to perform this function. One such module, formidable, was created by Felix Geisendörfer for his media upload and transformation startup, Transloadit, where performance and reliability are key.

What makes formidable a great choice for handling file uploads is that it's a streaming parser, meaning it can accept chunks of data as they arrive, parse them, and emit specific parts, such as the part headers and bodies previously mentioned. Not

only is this approach fast, but the lack of buffering prevents memory bloat, even for very large files such as videos, which otherwise could overwhelm a process.

Now, back to our photo-sharing example. The HTTP server in the following listing implements the beginnings of the file upload server. It responds to `GET` with an HTML form, and it has an empty function for `POST`, in which formidable will be integrated to handle file uploading.

Listing 4.7 HTTP server setup prepared to accept file uploads

```
var http = require('http');
var server = http.createServer(function(req, res){
  switch (req.method) {
    case 'GET':
      show(req, res);
      break;
    case 'POST':
      upload(req, res);
      break;
  }
});

function show(req, res) {
  var html = ''
    + '<form method="post" action="/" enctype="multipart/form-data">'
    + '<p><input type="text" name="name" /></p>'
    + '<p><input type="file" name="file" /></p>'
    + '<p><input type="submit" value="Upload" /></p>'
    + '</form>';
  res.setHeader('Content-Type', 'text/html');
  res.setHeader('Content-Length', Buffer.byteLength(html));
  res.end(html);
}

function upload(req, res) {
  // upload logic
}
```

Serve HTML form with file input

Now that the `GET` request is taken care of, it's time to implement the `upload()` function, which is invoked by the request callback when a `POST` request comes in. The `upload()` function needs to accept the incoming upload data, which is where formidable comes in. In the rest of this section, you'll learn what's needed in order to integrate formidable into your web application:

1. Install formidable through npm.
2. Create an `IncomingForm` instance.
3. Call `form.parse()` with the HTTP request object.
4. Listen for form events `field`, `file`, and `end`.
5. Use formidable's high-level API.

The first step to utilizing formidable in the project is to install it. This can be done by executing the following command, which installs the module locally into the ./node_modules directory:

```
$ npm install formidable
```

To access the API, you need to `require()` it, along with the initial http module:

```
var http = require('http');
var formidable = require('formidable');
```

The first step to implementing the `upload()` function is to respond with 400 Bad Request when the request doesn't appear to contain the appropriate type of content:

```
function upload(req, res) {
  if (!isFormData(req)) {
    res.statusCode = 400;
    res.end('Bad Request: expecting multipart/form-data');
    return;
  }
}

function isFormData(req) {
  var type = req.headers['content-type'] || '';
  return 0 == type.indexOf('multipart/form-data');
}
```

The helper function `isFormData()` checks the `Content-Type` header field for `multipart/form-data` by using the JavaScript `String.indexOf()` method to assert that `multipart/form-data` is at the beginning of the field's value.

Now that you know that it's a multipart request, you need to initialize a new `formidable.IncomingForm` form and then issue the `form.parse(req)` method call, where `req` is the request object. This allows formidable to access the request's `data` events for parsing:

```
function upload(req, res) {
  if (!isFormData(req)) {
    res.statusCode = 400;
    res.end('Bad Request');
    return;
  }

  var form = new formidable.IncomingForm();
  form.parse(req);
}
```

The `IncomingForm` object emits many events itself, and by default it streams file uploads to the /tmp directory. As shown in the following listing, formidable issues events when form elements have been processed. For example, a `file` event is issued when a file has been received and processed, and `field` is issued on the complete receipt of a field.

Listing 4.8 Using formidable's API

```
...
var form = new formidable.IncomingForm();

form.on('field', function(field, value){
  console.log(field);
  console.log(value);
});

form.on('file', function(name, file){
  console.log(name);
  console.log(file);
});

form.on('end', function(){
  res.end('upload complete!');
});

form.parse(req);
...
```

By examining the first two `console.log()` calls in the `field` event handler, you can see that "my clock" was entered in the `name` text field:

```
name
my clock
```

The `file` event is emitted when a file upload is complete. The `file` object provides you with the file size, the path in the form.uploadDir directory (/tmp by default), the original basename, and the MIME type. The `file` object looks like the following when it's passed to `console.log()`:

```
{ size: 28638,
  path: '/tmp/d870ede4d01507a68427a3364204cdf3',
  name: 'clock.png',
  type: 'image/png',
  lastModifiedDate: Sun, 05 Jun 2011 02:32:10 GMT,
  length: [Getter],
  filename: [Getter],
  mime: [Getter],
  ...
}
```

Formidable also provides a higher-level API, essentially wrapping the API we've already looked at into a single callback. When a function is passed to `form.parse()`, an `error` is passed as the first argument if something goes wrong. Otherwise, two objects are passed: `fields` and `files`.

The `fields` object may look something like the following `console.log()` output:

```
{ name: 'my clock' }
```

The `files` object provides the same `File` instances that the `file` event emits, keyed by name like `fields`.

It's important to note that you can listen for these events even while using the callback, so functions like progress reporting aren't hindered. The following code shows how this more concise API can be used to produce the same results that we've already discussed:

```
var form = new formidable.IncomingForm();
form.parse(req, function(err, fields, files){
  console.log(fields);
  console.log(files);
  res.end('upload complete!');
});
```

Now that you have the basics, we'll look at calculating upload progress, a process that comes quite naturally to Node and its event loop.

4.4.3 Calculating upload progress

Formidable's `progress` event emits the number of bytes received and bytes expected. This allows you to implement a progress bar. In the following example, the percentage is computed and logged by invoking `console.log()` each time the `progress` event is fired:

```
form.on('progress', function(bytesReceived, bytesExpected){
  var percent = Math.floor(bytesReceived / bytesExpected * 100);
  console.log(percent);
});
```

This script will yield output similar to the following:

```
1
2
4
5
6
8
...
99
100
```

Now that you understand this concept, the next obvious step would be to relay that progress back to the user's browser. This is a fantastic feature for any application expecting large uploads, and it's a task that Node is well suited for. By using the WebSocket protocol, for instance, or a real-time module like Socket.IO, it would be possible in just a few lines of code. We'll leave that as an exercise for you to figure out.

We have one final, and very important, topic to cover: securing your application.

4.5 Securing your application with HTTPS

A frequent requirement for e-commerce sites, and sites dealing with sensitive data, is to keep traffic to and from the server private. Standard HTTP sessions involve the client and server exchanging information using unencrypted text. This makes HTTP traffic fairly trivial to eavesdrop on.

The Hypertext Transfer Protocol Secure (HTTPS) protocol provides a way to keep web sessions private. HTTPS combines HTTP with the TLS/SSL transport layer. Data sent using HTTPS is encrypted and is therefore harder to eavesdrop on. In this section, we'll cover some basics on securing your application using HTTPS.

If you'd like to take advantage of HTTPS in your Node application, the first step is getting a private key and a certificate. The private key is, essentially, a "secret" needed to decrypt data sent between the server and client. The private key is kept in a file on the server in a place where it can't be easily accessed by untrusted users. In this section, you'll generate what's called a *self-signed certificate*. These kinds of SSL certificates can't be used in production websites because browsers will display a warning message when a page is accessed with an untrusted certificate, but it's useful for development and testing encrypted traffic.

To generate a private key, you'll need OpenSSL, which will already be installed on your system if you installed Node. To generate a private key, which we'll call key.pem, open up a command-line prompt and enter the following:

```
openssl genrsa 1024 > key.pem
```

In addition to a private key, you'll need a certificate. Unlike a private key, a certificate can be shared with the world; it contains a public key and information about the certificate holder. The public key is used to encrypt traffic sent from the client to the server.

The private key is used to create the certificate. Enter the following to generate a certificate called key-cert.pem:

```
openssl req -x509 -new -key key.pem > key-cert.pem
```

Now that you've generated your keys, put them in a safe place. In the HTTPS server in the following listing we reference keys stored in the same directory as our server script, but keys are more often kept elsewhere, typically ~/.ssh. The following code will create a simple HTTPS server using your keys.

Listing 4.9 HTTPS server options

```
var https = require('https');
var fs = require('fs');

var options = {
  key:  fs.readFileSync('./key.pem'),
  cert: fs.readFileSync('./key-cert.pem')
};

https.createServer(options, function (req, res) {
  res.writeHead(200);
  res.end("hello world\n");
}).listen(3000);
```

SSL key and cert given as options

options object is passed in first

https and http modules have almost identical APIs

Once the HTTPS server code is running, you can connect to it securely using a web browser. To do so, navigate to https://localhost:3000/ in your web browser. Because

the certificate used in our example isn't backed by a Certificate Authority, a warning will be displayed. You can ignore this warning here, but if you're deploying a public site, you should always properly register with a Certificate Authority (CA) and get a real, trusted certificate for use with your server.

4.6 Summary

In this chapter, we've introduced the fundamentals of Node's HTTP server, showing you how to respond to incoming requests and how to handle asynchronous exceptions to keep your application reliable. You've learned how to create a RESTful web application, serve static files, and even create an upload progress calculator.

You may also have seen that starting with Node from a web application developer's point of view can seem daunting. As seasoned web developers, we promise that it's worth the effort. This knowledge will aid in your understanding of Node for debugging, authoring open source frameworks, or contributing to existing frameworks.

This chapter's fundamental knowledge will prepare you for diving into Connect, a higher-level framework that provides a fantastic set of bundled functionality that every web application framework can take advantage of. Then there's Express—the icing on the cake! Together, these tools will make everything you've learned in this chapter easier, more secure, and more enjoyable.

Before we get there, though, you'll need somewhere to store your application data. In the next chapter, we'll look at the rich selection of database clients created by the Node community, which will help power the applications you create throughout the rest of the book.