20

NODE.JS

So far, we've used the JavaScript language in a single environment: the browser. This chapter and the next one will briefly introduce Node.js, a program that allows you to apply your JavaScript skills outside of the browser. With it, you can build anything from small command line tools to HTTP servers that power dynamic websites.

These chapters aim to teach you the main concepts that Node.js uses and to give you enough information to write useful programs for it. They do not try to be a complete, or even a thorough, treatment of the platform.

If you want to follow along and run the code in this chapter, you'll need to install Node.js version 18 or higher. To do so, go to *https://nodejs.org* and follow the installation instructions for your operating system. You can also find further documentation for Node.js there.

Background

When building systems that communicate over the network, the way you manage input and output—that is, the reading and writing of data to and from the network and hard drive—can make a big difference in how quickly a system responds to the user or to network requests.

In such programs, asynchronous programming is often helpful. It allows the program to send and receive data from and to multiple devices at the same time without complicated thread management and synchronization.

Node was initially conceived for the purpose of making asynchronous programming easy and convenient. JavaScript lends itself well to a system like Node. It is one of the few programming languages that does not have a built-in way to do input and output. Thus, JavaScript could be fit onto Node's rather eccentric approach to network and filesystem programming without

ending up with two inconsistent interfaces. In 2009, when Node was being designed, people were already doing callback-based programming in the browser, so the community around the language was used to an asynchronous programming style.

The node Command

When Node.js is installed on a system, it provides a program called `node`, which is used to run JavaScript files. Say you have a file *hello.js*, containing this code:

```
let message = "Hello world";
console.log(message);
```

You can then run `node` from the command line like this to execute the program:

```
$ node hello.js
Hello world
```

The `console.log` method in Node does something similar to what it does in the browser. It prints out a piece of text. But in Node, the text will go to the process's standard output stream rather than to a browser's JavaScript console. When running `node` from the command line, that means you see the logged values in your terminal.

If you run `node` without giving it a file, it provides you with a prompt at which you can type JavaScript code and immediately see the result.

```
$ node
> 1 + 1
2
> [-1, -2, -3].map(Math.abs)
[1, 2, 3]
> process.exit(0)
$
```

The `process` binding, just like the `console` binding, is available globally in Node. It provides various ways to inspect and manipulate the current program.

The `exit` method ends the process and can be given an exit status code, which tells the program that started `node` (in this case, the command line shell) whether the program completed successfully (code zero) or encountered an error (any other code).

To find the command line arguments given to your script, you can read `process.argv`, which is an array of strings. Note that it also includes the name of the `node` command and your script name, so the actual arguments start at index 2. If *showargv.js* contains the statement `console.log(process.argv)`, you could run it like this:

```
$ node showargv.js one --and two
["node", "/tmp/showargv.js", "one", "--and", "two"]
```

All the standard JavaScript global bindings, such as `Array`, `Math`, and `JSON`, are also present in Node's environment. Browser-related functionality, such as `document` or `prompt`, is not.

Modules

Beyond the bindings I mentioned, such as `console` and `process`, Node puts few additional bindings in the global scope. If you want to access built-in functionality, you have to ask the module system for it.

Node started out using the CommonJS module system, based on the `require` function, which we saw in Chapter 10. It will still use this system by default when you load a *.js* file.

But today, Node also supports the more modern ES module system. When a script's filename ends in *.mjs*, it is considered to be such a module, and you can use `import` and `export` in it (but not `require`). We will use ES modules in this chapter.

When importing a module—whether with `require` or `import`—Node has to resolve the given string to an actual file that it can load. Names that start with `/`, `./`, or `../` are resolved as files, relative to the current module's path. Here, `.` stands for the current directory, `../` for one directory up, and `/` for the root of the filesystem. If you ask for "`./graph.mjs`" from the file */tmp/ robot/robot.mjs*, Node will try to load the file */tmp/robot/graph.mjs*.

When a string that does not look like a relative or absolute path is imported, it is assumed to refer to either a built-in module or a module installed in a `node_modules` directory. For example, importing from "`node:fs`" will give you Node's built-in filesystem module. Importing "`robot`" might try to load the library found in *node_modules/robot/*. It's common to install such libraries using NPM, which we'll return to in a moment.

Let's set up a small project consisting of two files. The first one, called *main.mjs*, defines a script that can be called from the command line to reverse a string.

```
import {reverse} from "./reverse.mjs";

// Index 2 holds the first actual command line argument
let argument = process.argv[2];

console.log(reverse(argument));
```

The file *reverse.mjs* defines a library for reversing strings, which can be used both by this command line tool and by other scripts that need direct access to a string-reversing function.

```
export function reverse(string) {
  return Array.from(string).reverse().join("");
}
```

Remember that `export` is used to declare that a binding is part of the module's interface. That allows *main.mjs* to import and use the function.

We can now call our tool like this:

```
$ node main.mjs JavaScript
tpircSavaJ
```

Installing with NPM

NPM, introduced in Chapter 10, is an online repository of JavaScript modules, many of which are specifically written for Node. When you install Node on

your computer, you also get the `npm` command, which you can use to interact with this repository.

NPM's main use is downloading packages. We saw the `ini` package in Chapter 10. We can use NPM to fetch and install that package on our computer.

```
$ npm install ini
added 1 package in 723ms

$ node
> const {parse} = require("ini");
> parse("x = 1\ny = 2");
{ x: '1', y: '2' }
```

After running `npm install`, NPM will have created a directory called *node_modules*. Inside that directory will be an *ini* directory that contains the library. You can open it and look at the code. When we import "`ini`", this library is loaded, and we can call its `parse` property to parse a configuration file.

By default, NPM installs packages under the current directory rather than in a central place. If you are used to other package managers, this may seem unusual, but it has advantages—it puts each application in full control of the packages it installs and makes it easier to manage versions and clean up when removing an application.

Package Files

After running `npm install` to install some package, you will find not only a *node_modules* directory but also a file called *package.json* in your current directory. It is recommended to have such a file for each project. You can create it manually or run `npm init`. This file contains information about the project, such as its name and version, and lists its dependencies.

The robot simulation from Chapter 7, as modularized in the exercise in Chapter 10, might have a *package.json* file like this:

```
{
  "author": "Marijn Haverbeke",
  "name": "eloquent-javascript-robot",
```

```
  "description": "Simulation of a package-delivery robc
  "version": "1.0.0",
  "main": "run.mjs",
  "dependencies": {
    "dijkstrajs": "^1.0.1",
    "random-item": "^1.0.0"
  },
  "license": "ISC"
}
```

When you run `npm install` without naming a package to install, NPM will install the dependencies listed in *package.json*. When you install a specific package that is not already listed as a dependency, NPM will add it to *package.json*.

Versions

A *package.json* file lists both the program's own version and versions for its dependencies. Versions are a way to deal with the fact that packages evolve separately, and code written to work with a package as it existed at one point may not work with a later, modified version of the package.

NPM demands that its packages follow a schema called *semantic versioning*, which encodes some information about which versions are *compatible* (don't break the old interface) in the version number. A semantic version consists of three numbers separated by periods, such as `2.3.0`. Every time new functionality is added, the middle number has to be incremented. Every time compatibility is broken, so that existing code that uses the package might not work with the new version, the first number has to be incremented.

A caret character (`^`) in front of the version number for a dependency in *package.json* indicates that any version compatible with the given number may be installed. For example, "`^2.3.0`" would mean that any version greater than or equal to 2.3.0 and less than 3.0.0 is allowed.

The `npm` command is also used to publish new packages or new versions of packages. If you run `npm publish` in a directory that has a *package.json* file, it will publish a package with the name and version listed in the JSON file to the registry. Anyone can publish packages to NPM—though only under a package name that isn't in use yet, since it wouldn't be good if random people could update existing packages.

This book won't delve further into the details of NPM usage. Refer to *https://www.npmjs.com* for further documentation and a way to search for packages.

The Filesystem Module

One of the most commonly used built-in modules in Node is the `node:fs` module, which stands for "filesystem." It exports functions for working with files and directories.

For example, the function called `readFile` reads a file and then calls a callback with the file's contents.

```
import {readFile} from "node:fs";
readFile("file.txt", "utf8", (error, text) => {
  if (error) throw error;
  console.log("The file contains:", text);
});
```

The second argument to `readFile` indicates the *character encoding* used to decode the file into a string. There are several ways in which text can be encoded to binary data, but most modern systems use UTF-8. Unless you have reasons to believe another encoding is used, pass "`utf8`" when reading a text file. If you do not pass an encoding, Node will assume you are interested in the binary data and will give you a `Buffer` object instead of a string. This is an array-like object that contains numbers representing the bytes (8-bit chunks of data) in the files.

```
import {readFile} from "node:fs";
readFile("file.txt", (error, buffer) => {
  if (error) throw error;
  console.log("The file contained", buffer.length, "byt
              "The first byte is:", buffer[0]);
});
```

A similar function, `writeFile`, is used to write a file to disk.

```
import {writeFile} from "node:fs";
writeFile("graffiti.txt", "Node was here", err => {
  if (err) console.log(`Failed to write file: ${err}`);
  else console.log("File written.");
});
```

Here it was not necessary to specify the encoding—`writeFile` will assume that when it is given a string to write, rather than a `Buffer` object, it should write it out as text using its default character encoding, which is UTF-8.

The `node:fs` module contains many other useful functions: `readdir` will give you the files in a directory as an array of strings, `stat` will retrieve information about a file, `rename` will rename a file, `unlink` will remove one, and so on. See the documentation at *https://nodejs.org for specifics*.

Most of these take a callback function as the last parameter, which they call either with an error (the first argument) or with a successful result (the second). As we saw in Chapter 11, there are downsides to this style of programming—the biggest one being that error handling becomes verbose and error prone.

The `node:fs/promises` module exports most of the same functions as the old `node:fs` module but uses promises rather than callback functions.

```
import {readFile} from "node:fs/promises";
readFile("file.txt", "utf8")
  .then(text => console.log("The file contains:", text)
```

Sometimes you don't need asynchronicity and it just gets in the way. Many of the functions in `node:fs` also have a synchronous variant, which has the same name with `Sync` added to the end. For example, the synchronous version of `readFile` is called `readFileSync`.

```
import {readFileSync} from "node:fs";
console.log("The file contains:",
            readFileSync("file.txt", "utf8"));
```

Note that while such a synchronous operation is being performed, your program is stopped entirely. If it should be responding to the user or to other machines on the network, being stuck on a synchronous action might produce annoying delays.

The HTTP Module

Another central module is called `node:http`. It provides functionality for running an HTTP server.

This is all it takes to start an HTTP server:

```
import {createServer} from "node:http";
let server = createServer((request, response) => {
  response.writeHead(200, {"Content-Type": "text/html"});
  response.write(`
    <h1>Hello!</h1>
    <p>You asked for <code>${request.url}</code></p>`);
  response.end();
});
server.listen(8000);
console.log("Listening! (port 8000)");
```

If you run this script on your own machine, you can point your web browser at *http://localhost:8000/hello* to make a request to your server. It will respond with a small HTML page.

The function passed as the argument to `createServer` is called every time a client connects to the server. The `request` and `response` bindings are objects representing the incoming and outgoing data. The first contains information about the request, such as its `url` property, which tells us to what URL the request was made.

When you open that page in your browser, it sends a request to your own computer. This causes the server function to run and send back a response, which you can then see in the browser.

To send something to the client, you call methods on the `response` object. The first, `writeHead`, will write out the response headers (see Chapter 18). You give

it the status code (200 for “OK” in this case) and an object that contains header values. The example sets the `Content-Type` header to inform the client that we’ll be sending back an HTML document.

Next, the actual response body (the document itself) is sent with `response.write`. You’re allowed to call this method multiple times if you want to send the response piece by piece—for example, to stream data to the client as it becomes available. Finally, `response.end` signals the end of the response.

The call to `server.listen` causes the server to start waiting for connections on port 8000. This is why you have to connect to *localhost:8000* to speak to this server, rather than just *localhost*, which would use the default port 80.

When you run this script, the process just sits there and waits. When a script is listening for events—in this case, network connections—`node` will not automatically exit when it reaches the end of the script. To close it, press CTRL-C.

A real web server usually does more than the one in the example—it looks at the request’s method (the `method` property) to see what action the client is trying to perform and looks at the request’s URL to find out on which resource this action is being performed. We’ll see a more advanced server later in this chapter.

The `node:http` module also provides a `request` function that can be used to make HTTP requests. However, it is a lot more cumbersome to use than `fetch`, which we saw in Chapter 18. Fortunately, `fetch` is also available in Node as a global binding. Unless you want to do something very specific, such as processing the response document piece by piece as the data comes in over the network, I recommend sticking to `fetch`.

Streams

The response object that the HTTP server could write to is an example of a *writable stream* object, which is a widely used concept in Node. Such objects have a `write` method that can be passed a string or a `Buffer` object to write something to the stream. Their `end` method closes the stream and optionally takes a value to write to the stream before closing. Both of these methods can

also be given a callback as an additional argument, which they will call when the writing or closing has finished.

It is possible to create a writable stream that points at a file with the `createWriteStream` function from the `node:fs` module. You can then use the `write` method on the resulting object to write the file one piece at a time rather than in one shot, as with `writeFile`.

Readable streams are a little more involved. The `request` argument to the HTTP server's callback is a readable stream. Reading from a stream is done using event handlers rather than methods.

Objects that emit events in Node have a method called `on` that is similar to the `addEventListener` method in the browser. You give it an event name and then a function, and it will register that function to be called whenever the given event occurs.

Readable streams have "`data`" and "`end`" events. The first is fired every time data comes in, and the second is called whenever the stream is at its end. This model is most suited for *streaming* data that can be immediately processed, even when the whole document isn't available yet. A file can be read as a readable stream by using the `createReadStream` function from `node:fs`.

This code creates a server that reads request bodies and streams them back to the client as all-uppercase text:

```
import {createServer} from "node:http";
createServer((request, response) => {
  response.writeHead(200, {"Content-Type": "text/plain"
  request.on("data", chunk =>
    response.write(chunk.toString().toUpperCase()));
  request.on("end", () => response.end());
}).listen(8000);
```

The `chunk` value passed to the data handler will be a binary `Buffer`. We can convert this to a string by decoding it as UTF-8 encoded characters with its `toString` method.

The following piece of code, when run with the uppercasing server active, will send a request to that server and write out the response it gets:

```
fetch("http://localhost:8000/", {
  method: "POST",
  body: "Hello server"
}).then(resp => resp.text()).then(console.log);
// → HELLO SERVER
```

A File Server

Let's combine our newfound knowledge about HTTP servers and working with the filesystem to create a bridge between the two: an HTTP server that allows remote access to a filesystem. Such a server has all kinds of uses—it allows web applications to store and share data, or it can give a group of people shared access to a bunch of files.

When we treat files as HTTP resources, the HTTP methods `GET`, `PUT`, and `DELETE` can be used to read, write, and delete the files, respectively. We will interpret the path in the request as the path of the file that the request refers to.

We probably don't want to share our whole filesystem, so we'll interpret these paths as starting in the server's working directory, which is the directory in which it was started. If I ran the server from */tmp/public/* (or *C:\tmp\public* on Windows), then a request for */file.txt* should refer to */tmp/ public/file.txt* (or *C:\tmp\public\file.txt*).

We'll build the program piece by piece, using an object called `methods` to store the functions that handle the various HTTP methods. Method handlers are `async` functions that get the request object as their argument and return a promise that resolves to an object that describes the response.

```
import {createServer} from "node:http";

const methods = Object.create(null);

createServer((request, response) => {
  let handler = methods[request.method] || notAllowed;
  handler(request).catch(error => {
    if (error.status != null) return error;
```

```
      return {body: String(error), status: 500};
    }).then(({body, status = 200, type = "text/plain"}) =
      response.writeHead(status, {"Content-Type": type});
      if (body?.pipe) body.pipe(response);
      else response.end(body);
    });
  }).listen(8000);

  async function notAllowed(request) {
    return {
      status: 405,
      body: `Method ${request.method} not allowed.`
    };
  }
```

This starts a server that just returns 405 error responses, which is the code used to indicate that the server refuses to handle a given method.

When a request handler's promise is rejected, the `catch` call translates the error into a response object, if it isn't one already, so that the server can send back an error response to inform the client that it failed to handle the request.

The `status` field of the response description may be omitted, in which case it defaults to 200 (OK). The content type, in the `type` property, can also be left off, in which case the response is assumed to be plaintext.

When the value of `body` is a readable stream, it will have a `pipe` method that we can use to forward all content from a readable stream to a writable stream. If not, it is assumed to be either `null` (no body), a string, or a buffer, and it is passed directly to the response's `end` method.

To figure out which file path corresponds to a request URL, the `urlPath` function uses the built-in `URL` class (which also exists in the browser) to parse the URL. This constructor expects a full URL, not just the part starting with the slash that we get from `request.url`, so we give it a dummy domain name to fill in. It extracts its pathname, which will be something like "`/file.txt`", decodes that to get rid of the `%20`-style escape codes, and resolves it relative to the program's working directory.

```
  import {resolve, sep} from "node:path";
```

```
const baseDirectory = process.cwd();

function urlPath(url) {
  let {pathname} = new URL(url, "http://d");
  let path = resolve(decodeURIComponent(pathname).slice
  if (path != baseDirectory &&
      !path.startsWith(baseDirectory + sep)) {
    throw {status: 403, body: "Forbidden"};
  }
  return path;
}
```

As soon as you set up a program to accept network requests, you have to start worrying about security. In this case, if we aren't careful, it is likely that we'll accidentally expose our whole filesystem to the network.

File paths are strings in Node. To map such a string to an actual file, there's a nontrivial amount of interpretation going on. Paths may, for example, include `../` to refer to a parent directory. One obvious source of problems would be requests for paths like */../secret_file*.

To avoid such problems, `urlPath` uses the `resolve` function from the `node:path` module, which resolves relative paths. It then verifies that the result is *below* the working directory. The `process.cwd` function (where `cwd` stands for "current working directory") can be used to find this working directory. The `sep` binding from the `node:path` package is the system's path separator—a backslash on Windows and a forward slash on most other systems. When the path doesn't start with the base directory, the function throws an error response object, using the HTTP status code indicating that access to the resource is forbidden.

We'll set up the `GET` method to return a list of files when reading a directory and to return the file's content when reading a regular file.

One tricky question is what kind of `Content-Type` header we should set when returning a file's content. Since these files could be anything, our server can't simply return the same content type for all of them. NPM can help us again here. The `mime-types` package (content type indicators like `text/plain` are also called *MIME types*) knows the correct type for a large number of file extensions.

The following `npm` command, in the directory where the server script lives, installs a specific version of `mime`:

```
$ npm install mime-types@2.1.0
```

When a requested file does not exist, the correct HTTP status code to return is 404. We'll use the `stat` function, which looks up information about a file, to find out both whether the file exists and whether it is a directory.

```
import {createReadStream} from "node:fs";
import {stat, readdir} from "node:fs/promises";
import {lookup} from "mime-types";

methods.GET = async function(request) {
  let path = urlPath(request.url);
  let stats;
  try {
    stats = await stat(path);
  } catch (error) {
    if (error.code != "ENOENT") throw error;
    else return {status: 404, body: "File not found"};
  }
  if (stats.isDirectory()) {
    return {body: (await readdir(path)).join("\n")};
  } else {
    return {body: createReadStream(path),
            type: lookup(path)};
  }
};
```

Because it has to touch the disk and thus might take a while, `stat` is asynchronous. Since we're using promises rather than callback style, it has to be imported from `node:fs/promises` instead of directly from `node:fs`.

When the file does not exist, `stat` will throw an error object with a `code` property of "`ENOENT`". These somewhat obscure, Unix-inspired codes are how you recognize error types in Node.

The `stats` object returned by `stat` tells us a number of things about a file, such as its size (`size` property) and its modification date (`mtime` property). Here we

are interested in the question of whether it is a directory or a regular file, which the `isDirectory` method tells us.

We use `readdir` to read the array of files in a directory and return it to the client. For normal files, we create a readable stream with `createReadStream` and return that as the body, along with the content type that the `mime` package gives us for the file's name.

The code to handle `DELETE` requests is slightly simpler.

```
import {rmdir, unlink} from "node:fs/promises";

methods.DELETE = async function(request) {
  let path = urlPath(request.url);
  let stats;
  try {
    stats = await stat(path);
  } catch (error) {
    if (error.code != "ENOENT") throw error;
    else return {status: 204};
  }
  if (stats.isDirectory()) await rmdir(path);
  else await unlink(path);
  return {status: 204};
};
```

When an HTTP response does not contain any data, the status code 204 ("no content") can be used to indicate this. Since the response to deletion doesn't need to transmit any information beyond whether the operation succeeded, that is a sensible thing to return here.

You may be wondering why trying to delete a nonexistent file returns a success status code rather than an error. When the file being deleted is not there, you could say that the request's objective is already fulfilled. The HTTP standard encourages us to make requests *idempotent*, which means that making the same request multiple times produces the same result as making it once. In a way, if you try to delete something that's already gone, the effect you were trying to create has been achieved—the thing is no longer there.

This is the handler for `PUT` requests:

```
import {createWriteStream} from "node:fs";

function pipeStream(from, to) {
  return new Promise((resolve, reject) => {
    from.on("error", reject);
    to.on("error", reject);
    to.on("finish", resolve);
    from.pipe(to);
  });
}

methods.PUT = async function(request) {
  let path = urlPath(request.url);
  await pipeStream(request, createWriteStream(path));
  return {status: 204};
};
```

We don't need to check whether the file exists this time—if it does, we'll just overwrite it. We again use `pipe` to move data from a readable stream to a writable one, in this case from the request to the file. But since `pipe` isn't written to return a promise, we have to write a wrapper, `pipeStream`, that creates a promise around the outcome of calling `pipe`.

When something goes wrong when opening the file, `createWriteStream` will still return a stream, but that stream will fire an "`error`" event. The stream from the request may also fail—for example, if the network goes down. So we wire up both streams' "`error`" events to reject the promise. When `pipe` is done, it will close the output stream, which causes it to fire a "`finish`" event. That's the point at which we can successfully resolve the promise (returning nothing).

The full script for the server is available at *https://eloquentjavascript.net/code/file_server.mjs*. You can download that and, after installing its dependencies, run it with Node to start your own file server. And, of course, you can modify and extend it to solve this chapter's exercises or to experiment.

The command line tool `curl`, widely available on Unix-like systems (such as macOS and Linux), can be used to make HTTP requests. The following session briefly tests our server. The `-X` option is used to set the request's method, and `-d` is used to include a request body.

```
$ curl http://localhost:8000/file.txt
File not found
$ curl -X PUT -d CONTENT http://localhost:8000/file.txt
$ curl http://localhost:8000/file.txt
CONTENT
$ curl -X DELETE http://localhost:8000/file.txt
$ curl http://localhost:8000/file.txt
File not found
```

The first request for `file.txt` fails, since the file does not exist yet. The `PUT` request creates the file, and behold, the next request successfully retrieves it. After deleting it with a `DELETE` request, the file is again missing.

Summary

Node is a nice, small system that lets us run JavaScript in a nonbrowser context. It was originally designed for network tasks to play the role of a node in a network, but it lends itself to all kinds of scripting tasks. If writing JavaScript is something you enjoy, automating tasks with Node may work well for you.

NPM provides packages for everything you can think of (and quite a few things you'd probably never think of), and it allows you to fetch and install those packages with the `npm` program. Node comes with a number of built-in modules, including the `node:fs` module for working with the filesystem and the `node:http` module for running HTTP servers.

All input and output in Node is done asynchronously, unless you explicitly use a synchronous variant of a function, such as `readFileSync`. Node originally used callbacks for asynchronous functionality, but the `node:fs/promises` package provides a promise-based interface to the filesystem.

Exercises

Search Tool

On Unix systems, there is a command line tool called `grep` that can be used to quickly search files for a regular expression.

Write a Node script that can be run from the command line and acts somewhat like `grep`. It treats its first command line argument as a regular expression and treats any further arguments as files to search. It outputs the names of any file whose content matches the regular expression.

When that works, extend it so that when one of the arguments is a directory, it searches through all files in that directory and its subdirectories.

Use asynchronous or synchronous filesystem functions as you see fit. Setting things up so that multiple asynchronous actions are requested at the same time might speed things up a little, but not a huge amount, since most filesystems can read only one thing at a time.

Directory Creation

Though the `DELETE` method in our file server is able to delete directories (using `rmdir`), the server currently does not provide any way to *create* a directory.

Add support for the `MKCOL` method ("make collection"), which should create a directory by calling `mkdir` from the `node:fs` module. `MKCOL` is not a widely used HTTP method, but it does exist for this same purpose in the WebDAV standard, which specifies a set of conventions on top of HTTP that make it suitable for creating documents.

A Public Space on the Web

Since the file server serves up any kind of file and even includes the right `Content-Type` header, you can use it to serve a website. Given that this server allows everybody to delete and replace files, this would make for an interesting kind of website: one that can be modified, improved, and vandalized by everybody who takes the time to make the right HTTP request.

Write a basic HTML page that includes a simple JavaScript file. Put the files in a directory served by the file server and open them in your browser.

Next, as an advanced exercise or even a weekend project, combine all the knowledge you gained from this book to build a more user-friendly interface for modifying the website—from *inside* the website.

Use an HTML form to edit the content of the files that make up the website, allowing the user to update them on the server by using HTTP requests, as described in Chapter 18.

Start by making only a single file editable. Then make it so that the user can select which file to edit. Use the fact that our file server returns lists of files when reading a directory.

Don’t work directly in the code exposed by the file server, since if you make a mistake, you are likely to damage the files there. Instead, keep your work outside of the publicly accessible directory and copy it there when testing.