O'REILLY®

Third Edition

JavaScript Cookbook

Programming the Web

Free Chapters

compliments of

Couchbase

Adam D. Scott,
Matthew MacDonald
& Shelley Powers

THIRD EDITION

JavaScript Cookbook

This excerpt contains Chapters 17, 18, and 21. The complete book is available on the O'Reilly Online Learning Platform and through other retailers.

Adam D. Scott, Matthew MacDonald, and Shelley Powers

Beijing • Boston • Farnham • Sebastopol • Tokyo

O'REILLY®

JavaScript Cookbook, Third Edition
by Adam D. Scott, Matthew MacDonald, and Shelley Powers

Printed in the United States of America.

Published by O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472.

O'Reilly books may be purchased for educational, business, or sales promotional use. Online editions are also available for most titles (*http://oreilly.com*). For more information, contact our corporate/institutional sales department: 800-998-9938 or *corporate@oreilly.com*.

Acquisitions Editor: Jennifer Pollock
Development Editor: Angela Rufino
Production Editor: Katherine Tozer
Copyeditor: Sonia Saruba
Proofreader: James Fraleigh

Indexer: Potomac Indexing, LLC
Interior Designer: David Futato
Cover Designer: Karen Montgomery
Illustrator: Kate Dullea

July 2021: Third Edition

Revision History for the Third Edition
2021-07-16: First Release

See *http://oreilly.com/catalog/errata.csp?isbn=9781492055754* for release details.

978-1-492-05575-4

[LSI]

Table of Contents

Foreword vii

17. Node Basics 1
17.1 Managing Node Versions with Node Version Manager 1
17.2 Responding to a Simple Browser Request 4
17.3 Interactively Trying Out Node Code Snippets with REPL 6
17.4 Reading and Writing File Data 9
17.5 Getting Input from the Terminal 14
17.6 Getting the Path to the Current Script 16
17.7 Working with Node Timers and Understanding the Node Event Loop 17

18. Node Modules 23
18.1 Searching for a Specific Node Module via npm 24
18.2 Converting Your Library into a Node Module 25
18.3 Taking Your Code Across Module Environments 26
18.4 Creating an Installable Node Module 29
18.5 Writing Multiplatform Libraries 35
18.6 Unit Testing Your Modules 39

21. Building Web Applications with Express 43
21.1 Using Express to Respond to Requests 43
21.2 Using the Express-Generator 47
21.3 Routing 52
21.4 Working with OAuth 54
21.5 OAuth 2 User Authentication with Passport.js 64
21.6 Serving Up Formatted Data 69
21.7 Building a RESTful API 70
21.8 Building a GraphQL API 74

Foreword

JavaScript Cookbook (O'Reilly) is an invaluable resource for any full stack or backend developer working with JavaScript. It provides practical solutions to common programming problems, organized into easy-to-follow recipes. Couchbase is highlighting three chapters that cover the essential topics of Node basics, Node modules, and building web applications with Express. These topics will help any developer who has some familiarity with JavaScript and wants to build a web API or a web application that interacts with a JSON NoSQL database like Couchbase.

The first chapter, "Node Basics," covers the basics of Node.js development, including using NVM to switch between versions of Node, creating a web server response, using REPL to test code snippets, utilizing the filesystem support, and the importance of the `__dirname` and `__filename` variables. It also highlights that Node runs on a single thread but interacts with threaded or asynchronous I/O operations through the "event loop." These are fundamental concepts for understanding how Node interacts with external processes, as in the Couchbase Node.js SDK and other database/backend services.

The second chapter, "Node Modules," covers Node's built-in modularity, which allows developers to download and use Node modules by including a single require() statement. For example, Ottoman.js is a Node.js library that provides developers with high-level abstractions that closely mirror the structure of their data. To include it in your application, first use `npm install ottoman`, and then use `require('ottoman')` within your code.

Modularity and the ability to include modules is a highly important concept to understand about any ecosystem. This cookbook includes straightforward solutions to the following fundamental topics:

- Searching the npm website to find modules
- Node's default module system, based on CommonJS
- Three key constructs: `exports`, `require()`, and `module`
- The *package.json* file for packaging and uploading modules to npm (with additional coverage of a README file)

And perhaps most importantly, this chapter touches on the importance of unit testing, which is stressed as important for building modules, but is important for testing all code.

Finally, the third chapter, "Building Web Applications With Express," covers the Express web framework for building web applications in Node. Express.js is appealing to full stack and backend developers because it provides a lightweight, highly customizable framework for both building web applications and/or APIs.

Topics covered include:

- Express support for multiple templating engines and CSS preprocessors
- Using routes to respond to HTTP requests based on the request path and parameters
- OAuth and OAuth 2 with Passport.js for authentication
- Serving formatted data that can be processed in webpages before display (particularly noteworthy to full stack developers)

Finally, it covers building a REST API or a GraphQL. Perhaps most refreshing about this section of the book, or at least most appropriate for a "cookbook," is a lack of discussion about which is better, REST or GraphQL. It simply presents these as two separate problems to solve, and explains how Express can solve them. It also leaves the actual backend data source out of the recipe, which makes the code more general purpose, but does require you to fill in things like CRUD operations with the Couchbase Node.js SDK or the use of higher level abstractions with a tool like Ottoman.js.

Overall, these excerpts from *JavaScript Cookbook* can help both backend and full stack developers. They cover essential topics such as Node.js basics, module development, and building web applications with Express. The practical examples and code snippets provided in each chapter make it easy for developers to apply the concepts to their own projects. Whether you're a seasoned developer or just starting with Node.js, this excerpt is a valuable reference that should not be missed.

I hope you enjoy this excerpt and find it useful. And if you're a backend or full stack developer who is evaluating databases, I hope you'll consider checking out Couchbase Capella™, the DBaaS from Couchbase. There's a free Capella trial, no credit card needed, that allows you to experiment with the Couchbase Node.js SDK and/or the Ottoman.js library. The Capella trial comes with sample JSON data, which is a great fit for a full stack approach that includes JavaScript and JavaScript standards all the way from the database to the API to the front end.

—Matthew Groves

CHAPTER 17

Node Basics

The dividing line between "old" and "new" JavaScript occurred when Node.js (referred to primarily as just Node) was released to the world. Yes, the ability to dynamically modify page elements was an essential milestone, as was the emphasis on establishing a path forward to new versions of ECMAScript, but it was Node that really made us look at JavaScript in a whole new way. And it's a way I like—I'm a big fan of Node and server-side JavaScript development.

In this chapter, we'll explore the basics of Node. At a minimum, you will need to have Node installed, as covered in Chapter 1 or Recipe 17.1.

17.1 Managing Node Versions with Node Version Manager

Problem

You need to install and manage multiple versions of Node on your development machine.

Solution

Use Node Version Manager (NVM), which allows you to install and use any distributed version of Node on a per-shell basis. NVM is compatible with Linux, macOS, and Windows Subsystem for Linux.

To install NVM, run the install script using either `curl` or `wget` in your system's terminal application:

```
## using curl:
curl -o- https://raw.githubusercontent.com/nvm-sh/nvm/v0.37.2/install.sh | bash
```

```
## using wget:
wget -qO- https://raw.githubusercontent.com/nvm-sh/nvm/v0.37.2/install.sh | bash
```

If you are developing on Windows, we recommend using nvm-windows, which is unaffiliated with the NVM project, but provides similar functionality for the Windows operating system. For instructions on how to use `nvm-windows`, consult the project's documentation.

Once you have installed NVM, you will need to install a version of Node. To install the latest version of Node, run:

```
$ nvm install node
```

You can also install a specific version of Node:

```
# install the latest path release of a major version
$ nvm install 15

# install a specific major/minor/patch version
$ nvm install 15.6.0
```

Once you've installed Node, you'll need to set a default version for new shell sessions. This can either be the latest version of Node that has been installed or a specific version number:

```
# default new shell sessions to the latest version of node
nvm alias default node
# default new shell sessions to a specific version
nvm alias default 14
```

To switch the version being used in a shell session, use the `nvm use` command followed by a specific installed version:

```
$ nvm use 15
```

Discussion

Using NVM allows you to easily download and switch between multiple versions of Node on your operating system. This can be incredibly useful when working with libraries that support multiple versions and legacy codebases. It also simplifies the management of Node within your development environment. You can view the list of releases and support timelines for each release.

When using NVM, it's possible to list out all of the versions installed on your machine using the `nvm ls` command. This will show all of the installed versions, the default version for new shell sessions, and any LTS versions that you do not have installed:

```
$ nvm ls
         v8.1.2
        v8.11.3
       v10.13.0
->     v10.23.1
        v12.8.0
       v12.20.0
       v12.20.1
        v13.5.0
       v14.14.0
       v14.15.1
       v14.15.4
        v15.6.0
         system
default -> 14 (-> v14.15.4)
node -> stable (-> v15.6.0) (default)
stable -> 15.6 (-> v15.6.0) (default)
iojs -> N/A (default)
unstable -> N/A (default)
lts/* -> lts/fermium (-> v14.15.4)
lts/argon -> v4.9.1 (-> N/A)
lts/boron -> v6.17.1 (-> N/A)
lts/carbon -> v8.17.0 (-> N/A)
lts/dubnium -> v10.23.1
lts/erbium -> v12.20.1
lts/fermium -> v14.15.4
```

As you can see, I have several redundant patch versions of major releases installed on my machine. To uninstall and remove a specific version, you can use the `nvm unin stall` command:

```
nvm uninstall 14.14
```

Keeping track of which version of Node a project is designed to use can be a challenge. To make this easier, you can add an `.nvmrc` file to your project's root directory. The contents of the file is the version of Node that the project is designed to use. For example:

```
# default to the latest LTS version
$ lts/*

# to use a specific version
$ 14.15.4
```

To use the version specified in a project's `.nvmrc` file, run `nvm use` command from the root of the director.

For large projects, using a container technology, such as Docker, is an incredibly useful way to ensure version matching across environments, including deployment. The Node documentation has a helpful guide on Dockerizing a Node.js web app.

17.2 Responding to a Simple Browser Request

Problem

You want to create a Node application that can respond to a very basic browser request.

Solution

Use the built-in Node HTTP server to respond to requests:

```
// load http module
const http = require('http');

// create http server
http
  .createServer((req, res) => {
    // content header
    res.writeHead(200, { 'content-type': 'text/plain' });

    // write message and signal communication is complete
    res.end('Hello, World!');
  })
  .listen(8124);

console.log('Server running on port 8124');
```

Discussion

A web server response to a browser request is the "Hello World" application for Node. It demonstrates not only how a Node application functions, but how you can communicate with it using a fairly traditional communication method: requesting a web resource.

Starting from the top, the first line of the solution loads the `http` module using Node's `require()` function. This instructs Node's modular system to load a specific library resource for use in the application. The `http` module is one of the many that come, by default, with a Node installation.

Next, an HTTP server is created using `http.createServer()`, passing in an anonymous function, known as the `RequestListener` with two parameters. Node attaches this function as an event handler for every server request. The two parameters are

request and *response*. The request is an instance of the `http.IncomingMessage` object and the response is an instance of the `http.ServerResponse` object.

The `http.ServerResponse` is used to respond to the web request. The `http.IncomingMessage` object contains information about the request, such as the request URL. If you need to get specific pieces of information from the URL (e.g., query string parameters), you can use the Node `url` utility module to parse the string. Example 17-1 demonstrates how the query string can be used to return a more custom message to the browser.

Example 17-1. Parsing out query string data

```
// load http module
const http = require('http');
const url = require('url');

// create http server
http
  .createServer((req, res) => {
    // get query string and parameters
    const { query } = url.parse(req.url, true);

    // content header
    res.writeHead(200, { 'content-type': 'text/plain' });

    // write message and signal communication is complete
    const name = query.first ? query.first : 'World';

    // write message and signal communication is complete
    res.end(`Hello, ${name}!`);
  })
  .listen(8124);

console.log('Server running on port 8124');
```

A URL like the following:

```
http://localhost:8124/?first=Reader
```

results in a web page that reads "Hello, Reader!"

In the code, the `url` module object has a `parse()` method that parses out the URL, returning various components of it (`href`, `protocol`, `host`, etc.). If you pass `true` as the second argument, the string is also parsed by another module, `querystring`, which returns the query string as an object with each parameter as an object property, rather than just returning a string.

In both the solution and in Example 17-1, a text message is returned as page output, using the `http.ServerResponse` `end()` method. I could also have written the message out using `write()`, and then called `end()`:

```
res.write(`Hello, ${name}!`);
res.end();
```

The important takeaway from either approach is you *must* call the response `end()` method after all the headers and response body have been set.

Chained to the end of the `createServer()` function call is another function call, this time to `listen()`, passing in the port number for the server to listen in on. This port number is also an especially important component of the application.

Traditionally, port 80 is the default port for most web servers (that aren't using HTTPS, which has a default port of 443). By using port 80, requests for the web resource don't need to specify a port when requesting the service's URL. However, port 80 is also the default port used by our more traditional web server, Apache. If you try to run the Node service on the same port that Apache is using, your application will fail. The Node application either must be standalone on the server, or run off a different port.

You can also specify an IP address (host) in addition to the port. Doing this ensures that people make the request to a specific host, as well as port. Not providing the host means the application will listen for the request for any IP address associated with the server. You can also specify a domain name, and Node resolves the host.

There are other arguments for the methods demonstrated, and a host of other methods, but this will get you started. Refer to the Node documentation for more information.

17.3 Interactively Trying Out Node Code Snippets with REPL

Problem

You want to easily run server-based Node code snippets.

Solution

Use Node's REPL (Read-Evalute-Print-Loop), an interactive command-line version of Node that can run any code snippet.

To use REPL, type `node` at the command line without specifying an application to run:

```
$ node
```

You can then specify JavaScript in a simplified Emacs (sorry, no vi) line-editing style. You can import libraries, create functions—whatever you can do within a static application. The main difference is that each line of code is interpreted instantly:

```
> const add = (x, y) => { return x + y };
undefined
> add(2, 2);
4
```

When you're finished, exit the program with `.exit`:

```
> .exit
```

Discussion

REPL can be started standalone or within another application if you want to set certain features. You type in the JavaScript as if you're typing in the script in a text file. The main behavioral difference is you might see a result after typing in each line, such as the `undefined` that shows up in the runtime REPL.

But you can import modules:

```
> const fs = require('fs');
```

And you can access the global objects, which we just did when we used `require()`.

The `undefined` that shows after typing in some code is the return value for the execution of the previous line of code. Setting a new variable and creating a function are some of the JavaScript that return `undefined`, which can get quickly annoying. To eliminate this behavior, as well as make some other modifications, you can use the `REPL.start()` function within a small Node application that triggers REPL (but with the options you specify).

The options you can use are:

`prompt`
: Changes the prompt that shows (default is >)

`input`
: Changes the input readable stream (default is `process.stdin`, which is the standard input)

`output`
: Changes the output writable stream (default is `process.stdout`, the standard output)

`terminal`
: Set to `true` if the stream should be treated like a TTY, and have ANSI/VT100 escape codes written

`eval`
: Function used to replace the asynchronous `eval()` function used to evaluate the JavaScript

`useColors`
: Set to `true` to set output colors for the `writer` function (default is based on the terminal's default values)

`useGlobal`
: Set to `true` to use the `global` object, rather than running scripts in a separate context

`ignoreUndefined`
: Set to `true` to eliminate the `undefined` return values

`writer`
: The function that returns the formatted result from the evaluated code to the display (default is the `util.inspect` function)

The following is an example application that starts REPL with a new prompt, ignoring the undefined values, and using colors:

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

const options = {
  prompt: '-> ',
  useColors: true,
  ignoreUndefined: true
};

repl.start(options);
```

The options we want are defined in the `options` object and then passed as parameters to `repl.start()`. When we run the application, REPL is started but we no longer have to deal with undefined values:

```
-> const add = (x, y) => { return x + y };
-> add(2, 2);
4
```

As you can see, this is a cleaner output without all those messy `undefined` printouts.

Extra: Wait a Second, What Global Object?

Caught that, did you?

One difference between JavaScript in Node and JavaScript in the browser is the global scoping. Traditionally in a browser, when you create a variable outside a function, using `var`, it belongs to the top-level global object, which we know as `window`:

```
var test = 'this is a test';
console.log(window.test); // 'this is a test'
```

Similarly, when using `let` or `const` in the browser, the variables are globally scoped, though not attached to the `window` object.

In Node, each module operates within its own separate context, so modules can declare the same variables, and they won't conflict if they're all used in the same application.

However, there are objects accessible from Node's `global` object. We've used a few in previous examples, including `console`, the Buffer object, and `require()`. Others include some very familiar old friends: `setTimeout()`, `clearTimeout()`, `setInterval()`, and `clearInterval()`.

17.4 Reading and Writing File Data

Problem

You want to read from or write to a locally stored file.

Solution

Node's filesystem management functionality is included as part of the Node core, via the `fs` module:

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

To read a file's contents, use the `readFile()` function:

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

fs.readFile('main.txt', 'utf8', (err, data) => {
  if (err) throw err;
  console.log(data);
});
```

To write to a file, use `writeFile()`:

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

const buf = "I'm going to write this text to a file";
fs.writeFile('main2.txt', buf, err => {
  if (err) throw err;
  console.log('wrote text to file');
});
```

The `writeFile()` function overwrites the existing file. To append text to the file, use `appendText()`:

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

const buf = "\nI'm going to add this text to a file";
fs.appendFile('main.txt', buf, err => {
  if (err) throw err;
  console.log('appended text to file');
});
```

Discussion

Node's filesystem support is both comprehensive and simple to use. To read from a file, use the `readFile()` function, which supports the following parameters:

- The filename, including the operating system path to the file if it isn't local to the application
- An options object, with options for `encoding`, as demonstrated in the solution, and `flag`, which is set to `r` by default (for reading)
- A callback function with parameters for an error and the read data

In the solution, if I didn't specify the encoding in my application, Node would have returned the file contents as a raw buffer. Since I did specify the encoding, the file content is returned as a string.

The `writeFile()` and `appendFile()` functions for writing and appending, respectively, take parameters similar to `readFile()`:

- The filename and path
- The string or buffer for the data to write to the file
- The options object, with options for `encoding` (`w` as default for `writeFile()` and `a` as the default for `appendFile()`) and `mode`, with a default value of `438` (`0666` in Octal)
- The callback function, with only one parameter: the error

The options value of `mode` can be used to set the file's permissions if the file was created by write or append. By default, the file is created as readable and writable by the owner, and readable by the group and the world.

I mentioned that the data to write can be either a buffer or a string. A string cannot handle binary data, so Node provides the buffer, which is capable of dealing with either strings or binary data. Both can be used in all of the filesystem functions discussed in this section, but you'll need to explicitly convert between the two types if you want to use them both.

For example, instead of providing the *utf8* `encoding` option when you use `writeFile()`, you convert the string to a buffer, providing the desired encoding when you do:

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

const str = "I'm going to write this text to a file";
const buf = Buffer.from(str, 'utf8');
fs.writeFile('mainbuf.txt', buf, err => {
  if (err) throw err;
  console.log('wrote text to file');
});
```

The reverse—that is, to convert the buffer to a string—is just as simple:

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

fs.readFile('main.txt', (err, data) => {
  if (err) throw err;
  const str = data.toString();
  console.log(str);
});
```

The buffer `toString()` function has three optional parameters: encoding, where to begin the conversion, and where to end it. By default, the entire buffer is converted using the *utf8* encoding.

The `readFile()`, `writeFile()`, and `appendFile()` functions are *asynchronous*, meaning they won't wait for the operation to finish before proceeding in the code. This is essential when it comes to notoriously slow operations such as file access. There are synchronous versions of each: `readFileSync()`, `writeFileSync()`, and `appendFileSync()`. I can't stress enough that you should *not* use these variations. I only include a reference to them to be comprehensive.

Advanced

Another way to read or write from a file is to use the `open()` function in combination with `read()` for reading the file contents, or `write()` for writing to the file. The advantages to this approach is more finite control of what happens during the process. The disadvantage is the added complexity associated with all of the functions, including only being able to use a buffer for reading from and writing to the file.

The parameters for `open()` are:

- Filename and path
- Flag

- Optional mode
- Callback function

The same `open()` is used with all operations, with the *flag* controlling what happens. There are quite a few flag options, but the ones that interest us the most at this time are:

`r`
: Opens the file for reading; the file must exist

`r+`
: Opens the file for reading and writing; an exception occurs if the file doesn't exist

`w`
: Opens the file for writing, truncates the file, or creates it if it doesn't exist

`wx`
: Opens the file for writing, but fails if the file *does* exist

`w+`
: Opens the file for reading and writing; creates the file if it doesn't exist; truncates the file if it exists

`wx+`
: Similar to `w+`, but fails if the file exists

`a`
: Opens the file for appending, creates it if it doesn't exist

`ax`
: Opens the file for appending, fails if the file exists

`a+`
: Opens the file for reading and appending; creates the file if it doesn't exist

`ax+`
: Similar to `a+`, but fails if the file exists

The mode is the same one mentioned earlier, a value that sets the *sticky* and *permission* bits on the file if created, and defaults to `0666`. The callback function has two parameters: an error object, if an error occurs, and a *file descriptor*, used by subsequent file operations.

The `read()` and `write()` functions share the same basic types of parameters:

- The `open()` methods callback file descriptor
- The buffer used to either hold data to be written or appended, or read

- The offset where the input/output (I/O) operation begins
- The buffer length (set by read operation, controls write operation)
- Position in the file where the operation is to take place; *null* if the position is the current position

The callback functions for both methods have three arguments: an error, bytes read (or written), and the buffer.

That's a lot of parameters and options. The best way to demonstrate how it all works is to create a complete Node application that opens a brand new file for writing, writes some text to it, writes some more text to it, and then reads all the text back and prints it to the `console`. Since `open()` is asynchronous, the read and write operations have to occur within the callback function. Be ready for it in Example 17-2, because you're going to get your first taste of a concept known as *callback hell*.

Example 17-2. Demonstrating open, read, and write

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

fs.open('newfile.txt', 'a+', (err, fd) => {
  if (err) {
    throw err;
  } else {
    const buf = Buffer.from('The first string\n');
    fs.write(fd, buf, 0, buf.length, 0, (err, written) => {
      if (err) {
        throw err;
      } else {
        const buf2 = Buffer.from('The second string\n');
        fs.write(fd, buf2, 0, buf2.length, buf.length, (err, written2) => {
          if (err) {
            throw err;
          } else {
            const length = written + written2;
            const buf3 = Buffer.alloc(length);
            fs.read(fd, buf3, 0, length, 0, err => {
              if (err) {
                throw err;
              } else {
                console.log(buf3.toString());
              }
            });
          }
        });
      }
    });
  }
});
```

Taming callbacks is covered in Chapter 19.

To find the length of the buffers, I used `length`, which returns the number of bytes for the buffer. This value doesn't necessarily match the length of a string in the buffer, but it does work in this usage.

That many levels of indentation can make your skin crawl, but the example demonstrates how `open()`, `read()`, and `write()` work. These combinations of functions are what's used within the `readFile()`, `writeFile()`, and `appendFile()` functions to manage file access. The higher-level functions just simplify the most common file operations.

See Chapter 19 for a solution to all that nasty indentation.

17.5 Getting Input from the Terminal

Problem

You want to get input from the application user via the terminal.

Solution

Use Node's Readline module.

To get data from the standard input, use code such as the following:

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

const rl = readline.createInterface({
  input: process.stdin,
  output: process.stdout
});

rl.question(">>What's your name?  ", answer => {
  console.log(`Hello ${answer}`);
  rl.close();
});
```

Discussion

The Readline module provides the ability to get lines of text from a readable stream. You start by creating an instance of the Readline interface with `createInterface()` passing in, at minimum, the readable and writable streams. You need both, because you're writing prompts, as well as reading in text. In the solution, the input stream is `process.stdin`, the standard input stream, and the output stream is `process.stdout`. In other words, input and output are from, and to, the command line.

The solution uses the `question()` function to post a question, and provides a callback function to process the response. Within the function, `close()` is called, which closes the interface, releasing control of the input and output streams.

You can also create an application that continues to listen to the input, taking some action on the incoming data, until something signals the application to end. Typically that something is a letter sequence signaling the person is done, such as the word *exit*. This type of application makes use of other Readline functions, such as `setPrompt()` to change the prompt given the individual for each line of text; `prompt()`, which prepares the input area, including changing the prompt to the one set by `setPrompt()`; and `write()`, to write out a prompt. In addition, you'll also need to use event handlers to process events, such as `line`, which listens for each new line of text.

Example 17-3 contains a complete Node application that continues to process input from the user until they type in `exit`. Note that the application makes use of `process.exit()`. This function cleanly terminates the Node application.

Example 17-3. Access numbers from stdin until the user types in exit

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

let sum = 0;

const rl = readline.createInterface({
  input: process.stdin,
  output: process.stdout
});

console.log("Enter numbers, one to a line. Enter 'exit' to quit.");

rl.setPrompt('>> ');
rl.prompt();

rl.on('line', input => {
  const userInput = input.trim();
  if (userInput === 'exit') {
    rl.close();
    return;
  }
```

```
  sum += Number(userInput);
  rl.prompt();
});

// user typed in 'exit'
rl.on('close', () => {
  console.log(`Total is ${sum}`);
  process.exit(0);
});
```

Running the application with several numbers results in the following output:

```
Enter numbers, one to a line. Enter 'exit' to quit.
>> 55
>> 209
>> 23.44
>> 0
>> 1
>> 6
>> exit
Total is 294.44
```

I used `console.log()` rather than the Readline interface `write()` to write the prompt, followed by a new line, and to differentiate the output from the input.

See Also

Chapter 19 covers passing and reading command-line arguments in Node applications.

17.6 Getting the Path to the Current Script

Problem

Your application needs to read the path of the script that is being executed.

Solution

Use the `__dirname` or `__filename` variables, which are in the scope of the module executing it:

```
// logs the directory of the currently executed file
// ex: /Users/Adam/Projects/js-cookbook/node
console.log(__dirname);

// logs the directory and filename of the currently executed file
// ex: /Users/Adam/Projects/js-cookbook/node/example.js
console.log(__filename);
```

Discussion

The `__dirname` or `__filename` variables appear to be in the global scope, but they actually exist in the scope of the module itself. Let's assume that you have a project with the following directory structure:

```
example-app
|   index.js
├──dir1
|   |   example.js
|   └──dir3
|       |   nested.js
```

If you were to read the `__dirname` in the index.js file, it would be the path to the project's root directory. However, reading the `__dirname` in from a script in the *nested.js* file would read the path to the *dir3* directory. This allows you to read the path of a module as it's executed, rather than being limited to the parent directory itself.

A useful example of `__dirname` in action is when creating a new file or directory within the current directory. In the following example, the script creates a new subdirectory named *cache* within the current file's directory:

```
const fs = require('fs');
const path = require('path');
const newDirectoryPath = path.join(__dirname, '/cache');

fs.mkdirSync(newDirectoryPath);
```

17.7 Working with Node Timers and Understanding the Node Event Loop

Problem

You need to use a timer in a Node application, but you're not sure which of Node's three timers to use, or how accurate they are.

Solution

If your timer doesn't have to be precise, you can use `setTimeout()` to create a single timer event, or `setInterval()` if you want a reccurring timer:

```
setTimeout(() => {}, 3000);

setInterval(() => {}, 3000);
```

Both function timers can be canceled:

```
const timer1 = setTimeout(() => {}, 3000);
clearTimeout(timer1);
```

```
const timer2 = setInterval(() => {}, 3000);
clearInterval(timer2);
```

However, if you need more finite control of your timer, and immediate results, you might want to use `setImmediate()`. You don't specify a delay for it, as you want the callback to be invoked *immediately* after all I/O callbacks are processed but before any `setTimeout()` or `setInterval()` callbacks:

```
setImmediate(() => {});
```

It, too, can be cleared, with `clearImmediate()`.

Discussion

Node, being JavaScript based, runs on a single thread. It is *synchronous*. However, input/output (I/O) and other native API access either runs *asynchronously* or on a separate thread. Node's approach to managing this timing disconnect is the *event loop*.

In your code, when you perform an I/O operation, such as writing a chunk of text to a file, you specify a callback function to do any post-write activity. Once you've done so, the rest of your application code is processed. It doesn't wait for the file write to finish. When the file write has finished, an event signaling the fact is returned to Node, and pushed on to a queue, waiting for processing. Node processes this event queue, and when it gets to the event signaled by the completed file write, it matches the event to the callback, and the callback is processed.

As a comparison, think of going into a deli and ordering lunch. You wait in line to place your order, and are given an order number. You sit down and read the paper, or check your Twitter account while you wait. In the meantime, the lunch orders go into another queue for deli workers to process the orders. But each lunch request isn't always finished in the order received. Some lunch orders may take longer. They may need to bake or grill for a longer time. So the deli worker processes your order by preparing your lunch item and then placing it in an oven, setting a timer for when it's finished, and goes on to other tasks.

When the timer pings, the deli worker quickly finishes their current task, and pulls your lunch order from the oven. You're then notified that your lunch is ready for pickup by your order number being called out. If several time-consuming lunch items are being processed at the same time, the deli worker processes them as the timer for each item pings, in order.

All Node processes fit the pattern of the deli order queue: first in, first to be sent to the deli (thread) workers. However, certain operations, such as I/O, are like those lunch orders that need extra time to bake in an oven or grill, but don't require the deli worker to stop any other effort and wait for the baking and grilling. The oven or grill

timers are equivalent to the messages that appear in the Node event loop, triggering a final action based on the requested operation.

You now have a working blend of synchronous and asynchronous processes. But what happens with a timer?

Both `setTimeout()` and `setInterval()` fire after the given delay, but what happens is a message to this effect is added to the event loop, to be processed in turn. So if the event loop is particularly cluttered, there is a delay before the the timer functions' callbacks are called:

> It is important to note that your callback will probably not be called in exactly (delay) milliseconds. Node.js makes no guarantees about the exact timing of when the callback will fire, nor of the ordering things will fire in. The callback will be called as close as possible to the time specified.
>
> —Node Timers documentation

For the most part, whatever delay happens is beyond the kin of our human senses, but it can result in animations that don't seem to run smoothly. It can also add an odd effect to other applications.

In Example 17-4, I created a scrolling timeline in SVG, with data fed to the client via WebSockets. To emulate real-world data, I used a three-second timer and randomly generated a number to act as a data value. In the server code, I used `setInterval()`, because the timer is reccurring:

Example 17-4. Scrolling timeline example

```
const app = require('http');
const fs = require('fs');
const ws = require('nodejs-websocket');

let server;

// serve static page
const handler = (req, res) => {
  fs.readFile(`${__dirname}/drawline.html`, (err, data) => {
    if (err) {
      res.writeHead(500);
      return res.end('Error loading drawline.html');
    }
    res.writeHead(200);
    res.end(data);
    return data;
  });
};

/// start the webserver
// connections on Port 8124 will be handled by the handler
app.listen(8124);
```

```
app.createServer(handler);

// data timer
const startTimer = () => {
  setInterval(() => {
    const newval = Math.floor(Math.random() * 100) + 1;
    if (server.connections.length > 0) {
      console.log(`sending ${newval}`);
      const counter = { counter: newval };
      server.connections.forEach(conn => {
        conn.sendText(JSON.stringify(counter), () => {
          console.log('conn sent');
        });
      });
    }
  }, 3000);
};

// Create a websocket connection handler on a different port
server = ws
  .createServer(conn => {
    console.log('connected');
    conn.on('close', () => {
      console.log('Connection closed');
    });
  })
  .listen(8001, () => {
    startTimer();
  });
```

I included `console.log()` to call in the code so you can see the timer event in comparison to the communication responses. When the `setInterval()` function is called, it's pushed into the process. When its callback is processed, the WebSocket communications are also pushed into the queue.

The solution uses `setInterval()`, one of Node's three different types of timers. The `setInterval()` function has the same format as the one we use in the browser. You specify a callback for the first function, provide a delay time (in milliseconds), and any potential arguments. The timer is going to fire in three seconds, but we already know that the callback for the timer may not be immediately processed.

The same applies to the callbacks passed in the WebSocket `sendText()` calls. These are based on Node's Net (or TLS, if secure) sockets, and as the `socket.write()` (what's used for `sendText()`) documentation notes:

> The optional callback parameter will be executed when the data is finally written out—this may not be immediately.
>
> —Node documentation

If you set the timer to invoke immediately (giving zero as the delay value), you'll see that the data sent message is interspersed with the communication sent message (before the browser client freezes up, overwhelmed by the socket communications—you don't want to use a zero value in the application again).

However, the timelines for all the clients remain the same because the communications are sent within the timer's callback function, *synchronously*, so the data is the same for all of the communications—it's just the callbacks that are handled, seemingly out of order.

Earlier I mentioned using `setInterval()` with a delay of zero. In actuality, it isn't exactly zero—Node follows the HTML5 specification that browsers adhere to, and "clamps" the timer interval to a minimum value of four milliseconds. While this may seem to be too small of an amount to cause a problem, when it comes to animations and time-critical processes, the time delay can impact the overall appearance and/or function.

To bypass the constraints, Node developers utilize Node's `process.nextTick()` instead. The callback associated with `process.nextTick()` is processed on the next event loop go around, usually before any I/O callbacks (though there are constraints, which I'll get to in a minute). No more pesky four-millisecond throttling. But then, what happens if there's an enormous number of recursively called `process.nextTick()` calls?

To return to our deli analogy, during a busy lunch hour, workers can be overrun with orders and so caught up in trying to process new orders that they don't respond in a timely manner to the oven and grill pings. Things burn when this happens. If you've ever been to a well-run deli, you'll notice the counter person taking the orders will assess the kitchen before taking the order, tossing in some slight delay, or even taking on some of the kitchen duties, letting the people wait just a tiny bit longer in the order queue.

The same happens with Node. If `process.nextTick()` were allowed to be the spoiled child, always getting its way, I/O operations would get starved out. Node uses another value, `process.maxTickDepth`, with a default value of 1000 to constrain the number of `process.next()` callbacks that are processed before the I/O callbacks are allowed to play. It's the counter person in the deli.

In more recent releases of Node, the `setImmediate()` function was added. This function attempts to resolve all of the issues associated with the timing operations and create a happy medium that should work for most folks. When `setImmediate()` is called, its callback is added after the I/O callbacks, but before the `setTimeout()` and `setInterval()` callbacks. We don't have the four-millisecond tax for the traditional timers, but we also don't have the brat that is `process.nextTick()`.

To return one last time to the deli analogy, `setImmediate()` is a customer in the order queue who sees that the deli workers are overwhelmed with pinging ovens, and politely states they'll wait to give their order.

However, you do *not* want to use `setImmediate()` in the scrolling timeline example, as it will freeze your browser up faster than you can blink.

CHAPTER 18
Node Modules

One of the great aspects of writing Node.js applications is the built-in modularity the environment provides. It's simple to download and install any number of Node modules, and using them is equally simple: just include a single `require()` statement naming the module, and you're off and running.

The ease with which the modules can be incorporated is one of the benefits of JavaScript *modularization*. Modularizing ensures that external functionality is created in such a way that it isn't dependent on other external functionality, a concept known as *loose coupling*. This means I can use a `Foo` module, without having to include a `Bar` module, because `Foo` is tightly dependent on having `Bar` included.

JavaScript modularization is both a discipline and a contract. The discipline comes in having to follow certain mandated criteria in order for external code to participate in the module system. The contract is between you, me, and other JavaScript developers: we're following an agreed-on path when we produce (or consume) external functionality in a module system, and we all have expectations based on the module system.

One major dependency on virtually all aspects of application and library management and publication is the use of Git, a source control system, and GitHub, an extremely popular Git *endpoint*. How Git works and using Git with GitHub are beyond the scope of this book. I recommend the *Git Pocket Guide* by Richard Silverman (O'Reilly) to get more familiar with Git, and GitHub's own documentation for more on using this service.

18.1 Searching for a Specific Node Module via npm

Problem

You're creating a Node application and want to use existing modules, but you don't know how to discover them.

Solution

Chapter 1 explains how to install packages with npm, Node's popular package manager (and the glue that holds the Node universe together). But you haven't yet considered how to *find* the useful packages that you need in npm's sprawling registry.

In most cases, you'll discover modules via recommendations from your friends and codevelopers, but sometimes you need something new. You can search for new modules directly at the npm website. You can also use the npm command-line interface directly to search for a module. For instance, if you're interested in modules that do something with PDFs, run the following search at the command line:

```
$ npm search pdf
```

Discussion

The npm website provides more than just documentation for using npm; it also provides an interface for searching for modules. If you access each module's page at npm, you can see how popular the module is, what other modules are dependent on it, the license, and other relevant information.

However, you can also search for modules, directly, using npm. The process can take a fair amount of time and when it finishes, you're likely to get a huge number of modules in return, especially with a broader topic such as modules that work with PDFs.

You can refine the results by listing multiple terms:

```
$ npm search PDF generation
```

This query returns a much smaller list of modules, specific to PDF generation.

Once you do find a module that sounds interesting, you can get detailed information about it with:

```
$ npm view electron
```

You'll get useful information from the *package.json* of the module, which can tell you what it's dependent on, who wrote it, and when it was created. We still recommend checking out the module's npm website page and GitHub repository page directly. There you'll be able to determine if the module is being actively maintained, get a sense of how popular the module is, review open issues, and look at the source code.

18.2 Converting Your Library into a Node Module

Problem

You want to use one of your libraries in Node.

Solution

Convert the library into a Node module. In Node, each file is treated as a module. For example, if the library is a file containing a function stored at `/lib/hello.js`:

```
const hello = val => {
  return console.log(`Hello ${val}`);
};
```

You can convert it to work as a Node module with the `exports` keyword:

```
const hello = val => {
  return console.log(`Hello ${val}`);
};

module.exports = hello;
```

Alternately, can also `export` the function directly:

```
module.exports = val => {
  return console.log(`Hello ${val}`);
};
```

You can then use the module in your application:

```
var hello = require('./lib/hello.js');

// logs 'Hello world'
hello('world');
```

Discussion

Node's default module system is based on CommonJS, which uses three constructs: `exports` to define what's exported from the library, `require()` to include the module in the application, and `module`, which includes information about the module but also can be used to export a function directly.

If your library returns an object with several functions and data objects, you can assign each to the comparably named property on `module.exports`, or you could return an object:

```
const greeting = {
  hello: val => {
    return console.log(`Hello ${val}`);
  },
```

```
  ciao: val => {
    return console.log(`Ciao ${val}`);
  }
};

module.exports = greeting;
```

or:

```
const hello = val => {
  return console.log(`Hello ${val}`);
};

const ciao = val => {
  return console.log(`Ciao ${val}`);
};

module.exports = { hello, ciao };
```

And then access the object properties directly:

```
const greeting = require('./lib/greeting.js')

// logs 'Hello world'
greeting.hello('world');
// logs 'Ciao mondo'
greeting.ciao('mondo');
```

Because the module isn't installed using npm, and just resides in the directory where the application resides, it's accessed by the file location and name, not just the name.

See Also

In Recipe 18.3, we cover how to make sure your library code works in both CommonJS and ECMAScript module environments.

In Recipe 18.4, we cover how to create an standalone module.

18.3 Taking Your Code Across Module Environments

Problem

You've written a library that you'd like to share with others, but folks are using a variety of Node versions with both CommonJS and ECMAScript modules. How can you ensure your library works in all of the various environments?

Solution

Use CommonJS modules with an ECMAScript module wrapper.

First, write the library as a CommonJS module, saved with the *.cjs* file extension:

```
const bbarray = {
  concatArray: (str, array) => {
    return array.map(element => {
      return `${str} ${element}`;
    });
  },
  splitArray: (str, array) => {
    return array.map(element => {
      return element.substring(str.length + 1);
    });
  }
};

module.exports = bbarray;
exports.concatArray = bbarray.concatArray;
exports.splitArray = bbarray.splitArray;
```

Followed by an ECMAScript wrapper module, which uses the *.mjs* file extension:

```
import bbarray from './index.cjs';

export const { concatArray, splitArray } = bbarray;
export default bbarray;
```

And a *package.json* file, which includes the `type`, `main`, and `exports` fields:

```
"type": "module",
"main": "./index.cjs",
"exports": {
  ".": "./index.cjs",
  "./module": "./wrapper.mjs"
},
```

Users of our module, using CommonJS syntax, can use the `require` syntax to import the module:

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

bbarray.concatArray('is', ['test', 'three']);
bbarray.splitArray('is', ['is test', 'is three']);
```

or:

```
const { concatArray, splitArray } = require('bbarray');

concatArray('is', ['test', 'three']);
splitArray('is', ['is test', 'is three']);
```

While those using ECMAScript modules can specify the `module` version of the library to use the ES `import` syntax:

```
import bbarray from 'bbarray/module';

bbarray.concatArray('is', ['test', 'three']);
bbarray.splitArray('is', ['is test', 'is three']);
```

or:

```
import { concatArray, splitArray } from 'bbarray/module';

concatArray('is', ['test', 'three']);
splitArray('is', ['is test', 'is three']);
```

At the time of writing, it is possible to avoid the */module* naming convention for ECMAScript modules using the `--experimental-conditional-exports` flag. However, due to the current experimental nature and the potential of future changes in the syntax, we currently recommend against it. In future versions of Node, this will likely become the standard. You can read more about this approach in the Node documentation.

Discussion

CommonJS modules have been the standard in Node since the beginning, and tools such as Browserify brought this syntax out of the Node ecosystem, allowing developers to use Node style modules in the browser. The ECMAScript 2015 (also known as ES6) standard introduced a native JavaScript module syntax, which was introduced in Node 8.5.0 and could be used behind an `--experimental-module` flag. Beginning with Node 13.2.0, Node ships with native support for ECMAScript modules.

A common pattern is to write a module using either the CommonJS or ECMAScript module syntax and use a compile tool to ship both as either separate module entry points or exported paths. However, this runs the risk of a module being loaded twice if it is loaded directly via one syntax by the application and either loaded directly or by a dependency using the other syntax.

In *package.json* there are three key fields:

```
  "type": "module",
  "main": "./index.cjs",
  "exports": {
    ".": "./index.cjs",
    "./module": "./wrapper.mjs"
  },
```

`"type"`
: Specifies that this is a `module`, meaning that this library is using the ECMAScript module syntax. For libraries that exclusively use CommonJS, the `"type"` would be `"commonjs"`.

`"main"`
: Specifies the main entry point of the application, for which we will point to the CommonJS file.

`"exports"`
: Defines the exported paths of our modules. Through this consumers of the default `package` will receive the CommonJS module directly, while those using `package/module` will import the file from the ECMAScript module wrapper.

If we wish to avoid using the *.cjs* and *.mjs* file extensions, we may do so:

```
"type": "module",
"main": "./index.js",
"exports": {
  ".": "./index.js",
  "./module": "./wrapper.js"
},
```

See Also

In Recipe 18.5, we cover how to make sure your library code works across multiple module environments in both Node and the browser by using Webpack as a code bundler.

18.4 Creating an Installable Node Module

Problem

You've either created a Node module from scratch, or converted an existing library to one that will work in the browser or in Node. Now, you want to know how to modify it into a module that can be installed using npm.

Solution

Once you've created your Node module and any supporting functionality (including module tests), you can package the entire directory. The key to packaging and publishing the Node module is creating a *package.json* file that describes the module, any dependencies, the directory structure, what to ignore, and so on. You can generate a *package.json* file by running the `npm init` command in the root of the project's directory and following the prompts.

The following is a relatively basic *package.json* file:

```
{
  "name": "bbArray",
  "version": "0.1.0",
  "description": "A description of what my module is about",
  "main": "./lib/bbArray",
  "author": {
    "name": "Shelley Powers"
  },
  "keywords": [
```

```
    "array",
    "utility"
  ],
  "repository": {
    "type": "git",
    "url": "https://github.com/accountname/bbarray.git"
  },
  "engines" : {
    "node" : ">=0.10.0"
  },
  "bugs": {
    "url": "https://github.com/accountname/bbarray/issues"
  },
  "licenses": [
    {
      "type": "MIT",
      "url": "https://github.com/accountname/bbarray/raw/master/LICENSE"
    }
  ],
  "dependencies": {
     "some-module": "~0.1.0"
  },
  "directories":{
     "doc":"./doc",
     "man":"./man",
     "lib":"./lib",
     "bin":"./bin"
  },
  "scripts": {
    "test": "nodeunit test/test-bbarray.js"
  }
}
```

Once you've created *package.json*, package all the source directories and the *package.json* file as a gzipped tarball. Then install the package locally, or install it in npm for public access.

Discussion

The *package.json* file is key to packaging up a Node module for local installation or uploading to npm for management. At a minimum, it requires a `name` and a `version`. The other fields given in the solution are:

`description`
: A description of what the module is and does

`main`
: Entry file for the module

`author`
: Author(s) of the module

`keywords`
: List of keywords that can help others find the module

`repository`
: Place where the code lives, typically GitHub

`engines`
: Node versions you know your module works with

`bugs`
: Where to file bugs

`licenses`
: License for your module

`dependencies`
: A list of dependencies required by the module

`directories`
: A hash describing the directory structure for your module

`scripts`
: A hash of object commands that are run during the module life cycle

There are a host of other options that are described at the npm website. You can also use a tool to help you fill in many of these fields. Typing the following at the command line runs the tool that asks questions and then generates a basic *package.json* file:

```
$ npm init
```

Once you have your source set up and your *package.json* file, you can test whether everything works by running the following command in the top-level directory of your module:

```
$ npm install . -g
```

If you have no errors, then you can package the file as a gzipped tarball. At this point, if you want to publish the module, you'll first need to add yourself as a user in the npm registry:

```
$ npm add-user
```

To publish the Node module to the npm registry, use the following in the root directory of the module, specifying a URL to the tarball, a filename for the tarball, or a path:

```
$ npm publish ./
```

If you have development dependencies for your module, such as using a testing framework like Jest, one excellent shortcut to ensure these are added to your

package.json file is to use the following, in the same directory as the *package.json* file, when you're installing the dependent module:

```
$ npm install jest --save-dev
```

Not only does this install Jest (discussed later, in Recipe 2.6), this command also updates your *package.json* file with the following command:

```
"devDependencies": {
   "jest": "^24.9.0"
 }
```

You can also use this same type of option to add a module to `dependencies` in *package.json*. The following:

```
$ npm install express --save
```

adds the following to the *package.json* file:

```
"dependencies": {
    "express": "^3.4.11"
  }
```

If the module is no longer needed and shouldn't be listed in *package.json*, remove it from the `devDependencies` with:

```
$ npm remove jest
```

And remove a module to `dependencies` with:

```
$ npm remove express
```

If the module is the last in either `dependencies` or `devDependencies`, the property isn't removed. It's just set to an empty value:

```
"dependencies": {}
```

npm provides a decent developer guide for creating and installing a Node module. You should consider the use of an *.npmignore* or *.gitignore* file for keeping stuff *out* of your module. And though this is beyond the scope of the book, you should also become familiar with Git and GitHub, and make use of it for your applications/modules.

Extra: The README File and Markdown Syntax

When you package your module or library for reuse and upload it to a source repository such as GitHub, you'll need to provide how-to information about installing the module/library and basic information about how to use it. For this, you need a README file.

You've likely seen files named *README.md* with applications and Node modules. They're text-based with some odd, unobtrusive markup that you're not sure is useful, until you see it in a site like GitHub, where the README file provides all of the project page installation and usage information. The markup translates into HTML, making for readable web-based help.

The content for the README is marked up with annotation known as Markdown. The popular website Daring Fireball calls Markdown easy to read and write, but "Readability, however, is emphasized above all else." Unlike with HTML, the Markdown markup doesn't get in the way of reading the text.

Daring Fireball also provides an overview of generic Markdown, but if you're working with GitHub files, you might also want to check out GitHub's Flavored Markdown.

Here is a sample *REAMDE.md* file:

```
# Project Title

Provide a brief description of the project and what it does.
If the project has a UI, include a screenshot as well.

If more comprehensive documentation exists, link to it here.

## Features

Describe the core features of the project (what does it do?)
in the form of a bulleted list:

- Feature #1
- Feature #2
- Feature #3

## Getting Started

Provide installation instructions, general usage guidance, API examples,
and build and deployment information. Assume as little prior knowledge
as possible, describing everything in clear and coherent steps.

### Installation/Dependencies

How does a user get up and running with your project? What dependencies
does the project have? Aim to describe these in clear and simple steps.
Provide external links.

### Usage

Provide examples of how the project may be used. For large projects with
```

```
external documentation, provide a few examples and link to the full docs here.

### Build/Deployment

If the user will be building or deploying the project, add any useful guidance.

## Getting Help

What should users do and expect when they encounter bugs or get stuck using
your project? Set expectations for support, link to the issue tracker and
roadmap, if applicable.

Where should users go if they have a question? (Stack Overflow, Gitter, IRC,
mailing list, etc.)

If desired, you may also provide links to core contributor email addresses.

## Contributing Guidelines

Include instructions for setting up the development environment, code standards,
running tests, and submitting pull requests. It may be useful to link to a
separate CONTRIBUTING.md file. See this example from the Hoodie project:
https://github.com/hoodiehq/hoodie/blob/master/CONTRIBUTING.md

## Code of Conduct

Provide a link to the Code of Conduct for your project. I recommend using the
Contributor Covenant: http://contributor-covenant.org/

## License

Include a license for your project. If you need help choosing a license,
use this guide: https://choosealicense.com
```

Most popular text editors include Markdown syntax highlighting and previewing capabilities. There are also desktop Markdown editors available for all platforms. I can also use a CLI tool, like Pandoc, to covert the *README.md* file into readable HTML:

```
$ pandoc README.md -o readme.html
```

Figure 18-1 displays the generated content. It's not fancy, but it is eminently readable.

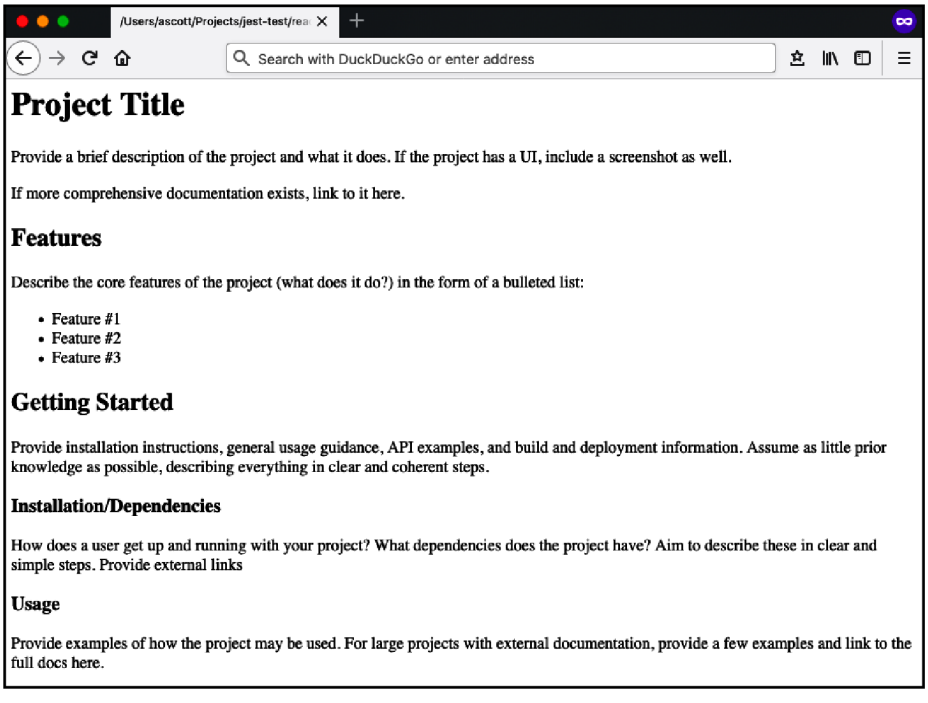

Figure 18-1. Generated HTML from README.md text and Markdown annotation

When you host your source code at a site such as GitHub, GitHub uses the *README.md* file to generate the cover page for the repository.

18.5 Writing Multiplatform Libraries

Problem

You've created a library that is useful both in the browser and in Node.js, and would like to make it available in both environments.

Solution

Use a bundling tool, such as Webpack, to bundle your library so that it works as an ES2015 module, CommonJS module, and AMD module, and can be loaded as a script tag in the browser.

In Webpack's *webpack.config.js* file, include the `library` and `libraryTarget` fields, which signify that the module should be bundled as a library and target multiple environments:

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

module.exports = {
  entry: './src/index.js',
  output: {
    path: path.resolve(__dirname, 'dist'),
    filename: 'my-library.js',
    library: 'myLibrary',
    libraryTarget: 'umd',
    globalObject: 'this'
  },
};
```

The `library` field specifies a name for the library that will be used in ECMAScript, CommonJS, and AMD module environments. The `libraryTarget` field allows you to specify how the module will be exposed. The default is `var`, which will expose a variable. Specifying `umd` will utilize the JavaScript Universal Module Definition (UMD), enabling the ability for multiple module styles to consume the library. To make the UMD build available in both browser and Node.js environments, you will need to set the `output.globalObject` option to `this`.

For more details on using Webpack to bundle code, see Chapter 1.

Discussion

In the example, I've created a simple math library. Currently, the only function is one called `squareIt`, which accepts a number as a parameter and returns the value of that number multiplied by itself. This is at *src/index.js*:

```
export function squareIt(num) {
    return num * num;
};
```

The *package.json* file contains Webpack and the Webpack command-line interface (CLI) as development dependencies. It also points the `main` distribution at the bundled version of the library, which Webpack will output to the *dist* folder. I've also added a build script that will run the Webpack bundler, aptly named `build`. This will allow me to generate the bundle by typing `npm run build` (or `yarn run build` if using Yarn).

```
{
  "name": "my-library",
  "version": "1.0.0",
  "description": "An example library bundled by Webpack",
  "main": "dist/my-library.js",
```

```
  "scripts": {
    "build": "webpack"
  },
  "keywords": ["example"],
  "author": "Adam Scott <adam@jseverywhere.io>",
  "license": "MIT",
  "devDependencies": {
    "webpack": "4.44.1",
    "webpack-cli": "3.3.12"
  }
}
```

Finally, my project contains a *webpack.config.js*, as described in the recipe:

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

module.exports = {
  entry: './src/index.js',
  output: {
    path: path.resolve(__dirname, 'dist'),
    filename: 'my-library.js',
    library: 'myLibrary',
    libraryTarget: 'umd',
    globalObject: 'this'
  },
};
```

With this setup, the command `npm run build` will bundle the library and place it within the *dist* directory of the project. This bundled file is what consumers of the library will use.

To test the package locally, before publishing it to npm, run `npm link` from the root of the project directory. Then in a separate project, where you'd like to use the module, type `npm link <library name>`. Doing so will create a symbolic link to the package, as though it is globally installed.

Publishing the library

Once your library is complete, you will most likely want to publish it to npm for distribution. Make sure that your project is version controlled with Git and has been pushed to a public remote repository (such as GitHub or GitLab). From the root of your project's directory:

```
$ git init
$ git remote add origin git://git-remote-url
$ npm publish
```

Once published to a remote Git repository and the npm registry, the library can be consumed by running `npm install`, downloading or cloning the Git repository, or

directly referencing the library in a web page using *https://unpkg.com/<library-name>*. The library can be consumed across the multiple JavaScript library formats.

As an ES 2015 module:

```
import * as myLibrary from 'my-library';

myLibrary.squareIt(4);
```

As a CommonJS module:

```
const myLibrary = require('my-library');

myLibrary.squareIt(4);
```

As an AMD module:

```
require(['myLibrary'], function (myLibrary) {
  myLibrary.squareIt(4);
});
```

And using a script tag on a web page:

```
<!doctype html>
<html>
  <script src="https://unpkg.com/my-library"></script>
  <script>
    myLibrary.squareIt(4);
  </script>
</html>
```

Handling library dependencies

Oftentimes a library may contain subdependencies. With our current setup, all dependencies will be packaged and bundled with the library itself. To limit the outputted bundle and to ensure that library consumers are not installing multiple instances of a subdependency, it may be best to treat them as a "peer dependency," which must also be installed or referenced on its own. To do so, add an `externals` property to your *webpack.config.js*. In the instance below, `moment` is being used as a peer dependency:

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

module.exports = {
  entry: './src/index.js',
  output: {
    path: path.resolve(__dirname, 'dist'),
    filename: 'my-library.js',
    library: 'myLibrary',
    libraryTarget: 'umd',
    globalObject: 'this'
  },
  externals: {
```

```
      moment: {
        commonjs: 'moment',
        commonjs2: 'moment',
        amd: 'moment',
        root: 'moment',
      }
    }
};
```

With this configuration, `moment` will be treated as a global variable by our library.

18.6 Unit Testing Your Modules

Problem

You want to make sure your module is functioning correctly and ready to be used by others.

Solution

Add *unit tests* as part of your production process.

Given the following module, named `bbarray`, and created in a file named *index.js*:

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

const bbarray = {
  concatArray: (str, array) => {
    if (!util.isArray(array) || array.length === 0) {
      return -1;
    }

    if (typeof str !== 'string') {
      return -1;
    }

    return array.map(element => {
      return `${str} ${element}`;
    });
  },
  splitArray: (str, array) => {
    if (!util.isArray(array) || array.length === 0) {
      return -1;
    }

    if (typeof str !== 'string') {
      return -1;
    }

    return array.map(element => {
      return element.substring(str.length + 1);
```

```
    });
  }
};

module.exports = bbarray;
```

Using Jest, a JavaScript testing framework, the following unit test (created as *index.js* and located in the project's *test* subdirectory) should result in the successful pass of six tests:

```
const bbarray = require('../index.js');

describe('concatArray()', () => {
  test('should return -1 when not using array', () => {
    expect(bbarray.concatArray(9, 'str')).toBe(-1);
  });

  test('should return -1 when not using string', () => {
    expect(bbarray.concatArray(9, ['test', 'two'])).toBe(-1);
  });

  test('should return an array with proper args', () => {
    expect(bbarray.concatArray('is', ['test', 'three'])).toStrictEqual([
      'is test',
      'is three'
    ]);
  });
});

describe('splitArray()', () => {
  test('should return -1 when not using array', () => {
    expect(bbarray.splitArray(9, 'str')).toBe(-1);
  });

  test('should return -1 when not using string', () => {
    expect(bbarray.splitArray(9, ['test', 'two'])).toBe(-1);
  });

  test('should return an array with proper args', () => {
    expect(bbarray.splitArray('is', ['is test', 'is three'])).toStrictEqual([
      'test',
      'three'
    ]);
  });
});
```

The result of the test is shown in Figure 18-2, run using `npm test`.

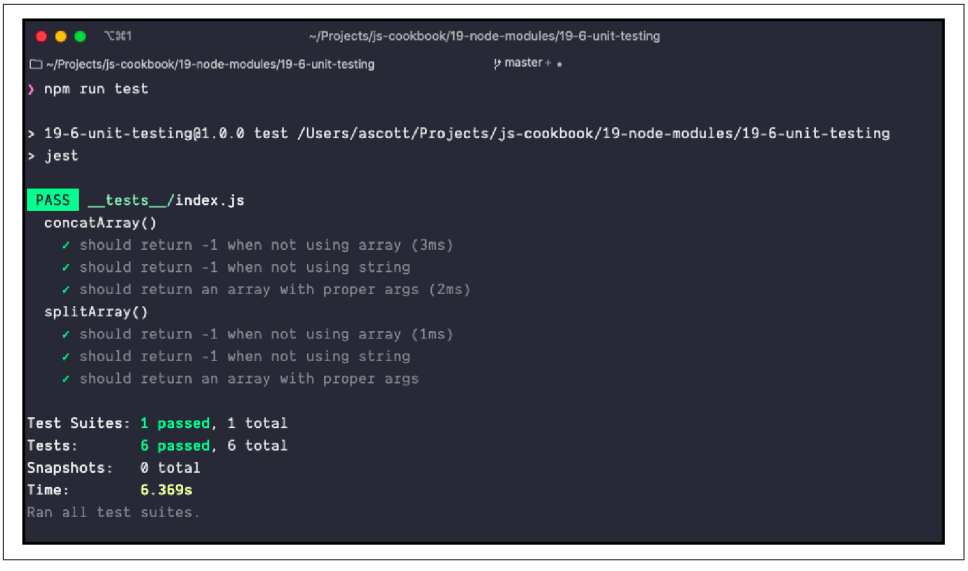

Figure 18-2. Running unit tests based on Jest

Discussion

A *unit test* is a way that developers test their code to ensure it meets the specifications. It involves testing functional behavior, and seeing what happens when you send bad arguments—or no arguments at all. It's called unit testing because it's used with individual units of code, such as testing one module in a Node application, as compared to testing the entire Node application. It becomes one part of *integration testing*, where all the pieces are plugged together, before going to *user acceptance testing*: testing to ensure that the application does what users expect it to do (and that they generally don't hate it when they use it).

Unit testing is one of those development tasks that may seem like a pain when you first start, but can soon become second nature. A good goal is to develop both tests and code in parallel to one another. Many developers even practice *test-driven development*, where unit tests are written prior to the code itself.

In the solution, we use Jest, a sophisticated testing framework. The module is simple, so we're not using some of the more complex Jest testing mechanisms. However, this provides an example of the building blocks of writing unit tests.

To install Jest, use the following:

```
$ npm install jest --save-dev
```

I'm using the `--save-dev` flag, because I'm installing Jest into the module's development dependencies. In addition, I modify the module's *package.json* file to add the following section:

```
"scripts": {
   "test": "jest"
 },
```

The test script is saved as *index.js* in the *tests* subdirectory under the project. Jest automatically looks for files in a *tests* directory or files following the *filename.test.js* naming pattern. The following command runs the test:

```
$ npm test
```

The Jest unit tests makes use of *expect matchers* to test for the returned values.

CHAPTER 21

Building Web Applications with Express

Express is a lightweight web framework that has been the long-standing leader in web application development in Node. Similar to Ruby's Sinatra and Python's Flask, the Express framework by itself is very minimal, but can be extended to build any type of web application. Express is also the backbone of batteries included in web application frameworks, such as Keystone.js, Sails, and Vulcan.js. If you are doing web application development in Node, you are likely to encounter Express. This chapter focuses on a handful of basic recipes for working with Express, which can be extended to build out all sorts of web applications.

21.1 Using Express to Respond to Requests

Problem

Your Node application needs to respond to HTTP requests.

Solution

Install the Express package:

```
$ npm install express
```

To set up Express, we require the module, call the module, and specify a port for connections in a file named *index.js*:

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

const app = express();
const port = process.env.PORT || '3000';

app.listen(port, () => console.log(`Listening on port ${port}`));
```

To respond to a request, specify a route and the response using Express's `.get` method:

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

const app = express();
const port = process.env.PORT || '3000';

app.get('/', (req, res) => res.send('Hello World'));

app.listen(port, () => console.log(`Listening on port ${port}`));
```

To serve static files, we can specify a directory with the `express.static` middleware

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

const app = express();
const port = process.env.PORT || '3000';

// middleware for static files
// will serve static files from the 'files' directory
app.use(express.static('files'));

app.listen(port, () => console.log(`Listening on port ${port}`));
```

To respond with HTML generated from a template, first install the templating engine:

```
$ npm install pug --save
```

Next, in the *index.js* file, set the `view engine` and specify the route that will respond with the template content:

```
app.set('view engine', 'pug')

app.get('/template', (req, res) => {
  res.render('template');
});
```

And then create a template file in the *views* subdirectory of the project with a new file. The template filename should match the name specified in `res.render`. In *views/template.pug*:

```
html
  head
    title="Using Express"
  body
    h1="Hello World"
```

Now requests to *http://localhost:3000/template* will return the template content as HTML.

Discussion

Express is a minimalist, but highly configurable framework for responding to HTTP requests and building out web applications. In the example, we set the port to `process.env.PORT` or port `3000`. In development, we can then specify a new port using an environment variable, such as:

```
$ PORT=7777 node index.js
```

or by using a *.env* file paired with the `dotenv` Node module. When deploying the application, the application hosting platform may require a specific port number or allow us to configure the port number ourselves.

With the Express `get` method, the application receives a request to a specific URI and then responds. In our example, when the application receives a request to the root URI (*/*), we respond with the text "Hello World":

```
app.get('/', (req, res) => res.send('Hello World'));
```

These responses can also be HTML, templates rendered to HTML, static files, and formatted data (such as JSON or XML).

Due to its minimal nature, Express itself contains minimal functionality, but can be extended using middleware. In Express, middleware functions have access to the `request` and `response` objects. Application-level middleware is bound to an instance of the `app` object through `app.use(MIDDLEWARE)`. In the example, we're making use of the built-in static files middleware:

```
app.use(express.static('files'));
```

Middleware packages can be used to extend Express's functionality in many ways. The `helmet` middleware package can be used to improve the Express security defaults:

```
const express = require('express');
const helmet = require('helmet');

const app = express();

app.use(helmet());
```

Templating engines simplify the process of writing HTML and allow you to pass data from your application to the page.

Here I am passing the data from the `userData` object to the template found at *views/user.pug*, which will be accessible at the */user* route:

```
// a user object of data to send to the template
const userData = {
  name: 'Adam',
  email: 'adam@jseverywhere.io',
```

```
  avatar: 'https://s.gravatar.com/avatar/33aab819d1ffa11fc4b31a4eebaf0c5a?s=80'
};

// render the template with user data
app.get('/user', (req, res) => {
  res.render('user', { userData });
});
```

Then in our template, we can make use of the data:

```
html
  head
    title User Page
  body
    h1 #{userData.name} Profile
    ul
      li
        image(src=userData.avatar)
      li #{userData.name}
      li #{userData.email}
```

The Pug templating engine is maintained by the Express core team and is a popular choice for Express applications, but its whitespace-driven syntax is not for everyone. EJS is an excellent alternative that offers a more HTML-like syntax. Here's how the above example would look using EJS.

First, specify to install the `ejs` package:

```
$ npm install ejs
```

Then set EJS as the view engine in your Express application:

```
app.set('view engine', 'ejs');
```

And in *views/user.ejs*:

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <title>User Page</title>
  </head>
  <body>
    <h1><%= userData.name %> Profile</h1>
    <ul>
      <li><img src=<%= userData.avatar %> /></li>
      <li><%= userData.name %></li>
      <li><%= userData.email %></li>
    </ul>
  </body>
</html>
```

21.2 Using the Express-Generator

Problem

You're interested in using Express to manage your server-side data application, but you don't want to manage all of the setup yourself.

Solution

To kickstart your Express application, use the Express-Generator. This is a command-line tool that generates the skeleton infrastructure of a typical Express application.

First, create a working directory where the tool can safely install a new application subdirectory. Next, run the `express-generator` command with `npx`:

```
$ npx express-generator --pug --git
```

I've passed two options with the command: `--pug` will result in the use of the Pug templating engine, while `--git` will generate a default *.gitignore* file in the project directory. For the full list of options, run the generator with the `-h` option:

```
$ npx express-generator -h
```

The generator creates a new directory with several subdirectories, some basic files to get you started, and a *package.json* file with all of the dependencies. To install the dependencies, change to the newly created directory and type:

```
$ npm install
```

Once all of the dependencies are installed, run the application using the following:

```
$ npm start
```

You can now access the generated Express application, using your IP address or domain and port 3000, the default Express port.

Discussion

Express provides a web application framework based on Node and with support for multiple templating engines and CSS preprocessors. In the solution, the options I chose for the example application are Pug as the template engine (the default) and the default of plain CSS (no CSS preprocessor). Though building the application from scratch enables a wider selection, Express supports only the following template engines:

`--ejs`
: Adds support for the EJS template engine

`--pug`
: Adds support for the Pug template engine

`--hbs`
: Adds support for the Handlebar template engine

`--hogan`
: Adds support for the Hogan.js template engine

Express also supports the following CSS preprocessors:

`express --css sass`
: Support for Sass

`express --css less`
: Support for Less

`express --css stylus`
: Support for Stylus

`express --css compass`
: Support for Compass

Not specifying any CSS preprocessor defaults to plain CSS.

Express also assumes that the project directory is empty. If it isn't, force the Express generator to generate the content by using the `-f` or `--force` option.

The newly generated subdirectory has the following structure (disregarding `node_modules`):

```
    app.js
    package-lock.json
    package.json
    /bin
       www
    /node_modules
    /public
       /images
       /javascripts
       /stylesheets
          style.css
          style.styl
    /routes
       index.js
       users.js
    /views
       error.pug
       index.pug
       layout.pug
```

The *app.js* file is the core of the Express application. It includes the references to the necessary libraries:

```
var createError = require('http-errors');
var express = require('express');
var path = require('path');
var cookieParser = require('cookie-parser');
var logger = require('morgan');

var indexRouter = require('./routes/index');
var usersRouter = require('./routes/users');
```

Although the convention followed in this book is to use `const` and `let` to define variables, at the time of writing, the Express generator uses `var`.

It also creates the Express app with the following line:

```
var app = express():
```

Next, it establishes Pug as the view engine by defining the `views` and `view engine` variables:

```
app.set('views', path.join(__dirname, 'views'));
app.set('view engine', 'pug');
```

The *middleware* calls are loaded next with `app.use()`. Middleware is functionality that sits between the raw request and the routing, processing specific types of requests. The rule for the middleware is if a path is not given as the first parameter, it defaults to a path of `/`, which means the middleware functions are loaded with the default path. In the following generated code:

```
app.use(logger('dev'));
app.use(express.json());
app.use(express.urlencoded({ extended: false }));
app.use(cookieParser());
app.use(express.static(path.join(__dirname, 'public')));
```

The first several middleware are loaded with every app request. Among the middleware includes support for development logging, as well as parsers for both JSON and *urlencoded* bodies. It's only when we get to the `static` entry that we see assignment to specific paths: the static file request middleware are loaded when requests are made to the *public* directory.

The routing is handled next:

```
app.use('/', indexRouter);
app.use('/users', usersRouter);
```

The top-level web request (/) is directed to the `routes` module, while all user requests (*/users*) get routed to the `users` module.

Read more about routing with Express in Recipe 21.3.

What follows is the error handling. First up is `404` error handling when a request is made to a nonexistent web resource:

```
app.use(function(req, res, next) {
  next(createError(404));
});
```

Next comes the server error handling, for both production and development:

```
app.use(function(err, req, res, next) {
  // set locals, only providing error in development
  res.locals.message = err.message;
  res.locals.error = req.app.get('env') === 'development' ? err : {};

  // render the error page
  res.status(err.status || 500);
  res.render('error');
});
```

The last line of the generated file is the `module.exports` for the `app`:

```
module.exports = app;
```

In the *routes* subdirectory, the default routing is included in the *routes/index.js* file:

```
var express = require('express');
var router = express.Router();

/* GET home page. */
router.get('/', function(req, res, next) {
  res.render('index', { title: 'Express' });
});

module.exports = router;
```

What's happening in the file is the Express router is used to route any HTTP GET requests to `/` to a callback where the request response receives a view rendered for the specific resource page. This is in contrast to what happens in the *routes/users.js* file, where the response receives a text message rather than a view:

```
var express = require('express');
var router = express.Router();
```

```
/* GET users listing. */
router.get('/', function(req, res, next) {
  res.send('respond with a resource');
});

module.exports = router;
```

What happens with the view rendering in the first request? There are three Pug files in the *views* subdirectory: one for error handling, one defining the page layout, and one, *index.pug*, that renders the page. The *index.pug* file contains:

```
extends layout

block content
  h1= title
  p Welcome to #{title}
```

It extends the *layout.pug* file, which contains:

```
doctype html
html
  head
    title= title
    link(rel='stylesheet', href='/stylesheets/style.css')
  body
    block content
```

The *layout.pug* file defines the overall structure of the page, regardless of content, including a reference to an automatically generated CSS file. The `block content` setting defines where the location of the content is placed. The format for the content is defined in *index.js*, in the equivalently named `block content` setting.

The Pug templating engine (formerly known as Jade) was popularized by Express and offers a minimalist take on templating that makes use of whitespace in place of traditional HTML style tags. This approach may not be for everyone, and the Pug alternatives (Handlebars, Hogan.js, and EJS) all offer a more HTML-like syntax.

The two Pug files define a basic web page with an `h1` element assigned a title variable, and a paragraph with a welcome message. Figure 21-1 shows the default page.

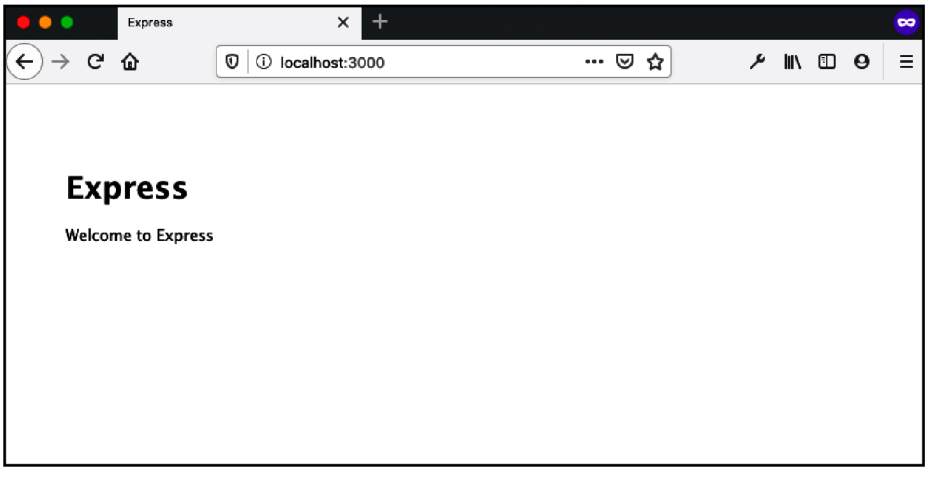

Figure 21-1. The Express-generated web page

Figure 21-1 shows that the page isn't especially fascinating, but it does represent how the pieces are holding together: the application router routes the request to the appropriate route module, which directs the response to the appropriate rendered view, and the rendered view uses data passed to it to generate the web page. If you make the following web request:

```
http://yourdomain.com:3000/users
```

you'll see the plain text message, rather than the rendered view.

By default, Express is set up to run in *development mode*. To change the application to *production mode*, you need to set an *environment variable*, `NODE-ENV` to "production." In a Linux or Unix environment, the following could be used:

```
$ export NODE_ENV=production
```

21.3 Routing

Problem

You want to route users to different resources in your application based on the request.

Solution

Use routes in Express to send specific resources based on the request path and parameters:

```
// respond with different route paths
app.get('/', (req, res) => res.send('Hello World'));
app.get('/users', (req, res) => res.send('Hello users'));

// parameters
app.get('/users/:userId', (req, res) => {
  res.send(`Hello user ${req.params.userId}`);
});
```

Discussion

In Express, we can return a response to the user when they make an HTTP request. In the above examples, I'm using `get` requests, but Express supports a number of additional methods. The most common of these methods are:

- `app.get`: request data
- `app.post`: send data
- `app.put`: send or update data
- `app.delete`: delete data

```
app.post('/new', (req, res) => {
  res.send('POST request to the `new` route');
});
```

Often we may want to enable multiple HTTP methods to a specific route. We can accomplish this by chaining them together:

```
app
  .route('/record')
  .get((req, res) => {
    res.send('Get a record');
  })
  .post((req, res) => {
    res.send('Add a record');
  })
  .put((req, res) => {
    res.send('Update a record');
  });
```

Often requests have parameters with specific values that we will make use of in our application. We can specify these in the URL using a colon (`:`):

```
app.get('/users/:userId', (req, res) => {
  res.send(`Hello user ${req.params.userId}`);
});
```

In the above example, when a user visits a URL at */users/adam123*, the browser will send the response of `Hello user adam123`. While this is a simple example, we could

also make use of the URL parameter to retrieve data from our database, passing the information on to a template.

We're also able to specify formats for the request parameters. In the following example, I make use of a regular expression to limit the `noteId` parameter to a six-digit integer:

```
app.get('^/users/:userId/notes/:noteId([0-9]{6})', (req, res) => {
  res.send(`This is note ${req.params.noteId}`);
});
```

We are also able to use a regular expression to define an entire route:

```
app.get(/.*day$/, (req, res) => {
  res.send(`Every day feels like ${req.path}`);
});
```

The above example will route any request ending in `day`. For example, in local development a request to *http://localhost:3000/Sunday* will result in "Every day feels like Sunday" being printed to the page.

21.4 Working with OAuth

Problem

You need access to a third-party API (such as GitHub, Facebook, or Twitter) in your Node application, but it requires authorization. Specifically, it requires OAuth authorization.

Solution

You'll need to incorporate an OAuth client in your application. You'll also need to meet the OAuth requirements demanded by the resource provider.

See the discussion for details.

Discussion

OAuth is an authorization framework used with most popular social media and cloud content applications. If you've ever gone to a site and it's asked you to authorize access to data from a third-party service, such as GitHub, you've participated in the OAuth authorization *flow*.

There are two versions of OAuth, 1.0 and 2.0, which are not compatible with one another. OAuth 1.0 was based on proprietary APIs developed by Flickr and Google, was heavily web page focused, and didn't gracefully transcend the barrier among web, mobile, and service applications. When wanting to access resources in a mobile phone app, the app would have the user log in to the app in a mobile browser and

then copy access tokens to the app. Other criticisms of OAuth 1.0 is that the process required that the authorization server be the same as the resource server, which doesn't scale when you're talking about service providers such as Twitter, Facebook, and Amazon.

OAuth 2.0 presents a simpler authorization process, and also provides different types of authorization (different flows) for different circumstances. Some would say, though, that it does so at the cost of security, as it doesn't have the same demands for encrypting hash tokens and request strings.

Most developers won't have to create an OAuth 2.0 server, and doing so is way beyond the scope of this book, much less this recipe. But it's common for applications to incorporate an OAuth client (1.0 or 2.0) for one service or another, so I'm going to present different types of OAuth use. First, though, let's discuss the differences between authorization and authentication.

Authorization isn't authentication

Authorization is saying, "I authorize this application to access my resources on your server." Authentication is the process of authenticating whether you are, indeed, the person who owns this account and has control over these resources. An example would be if I want to comment on an article at a newspaper's online site. It will likely ask me to log in via some service. If I pick my Facebook account to use as the login, the news site will most likely want some data from Facebook.

The news site is, first, authenticating me as a legitimate Facebook user, with an established Facebook account. In other words, I'm not just some random person coming in and commenting anonymously. Secondly, the news site wants something from me in exchange for the privilege of commenting: it's going to want data about me. Perhaps it will ask for permission to post for me (if I post my comment to Facebook as well as the news site). This is both an authentication and an authorization request.

If I'm not already logged in to Facebook, I'll have to log in. Facebook is using my correct application of username and password to authenticate that, yes, I own the Facebook account in question. Once logged in, Facebook asks whether I agree to giving the newspaper site the authorization to access the resources it wants. If I agree (because I desperately want to comment on a particular story), Facebook gives the news site the authorization, and there's now a persistent connection from the newspaper to my Facebook account (which you can see in your Facebook settings). I can make my comment, and make comments at other stories, until I log out or revoke the Facebook authorization.

Of course, none of this implies that Facebook or the news site are actually authenticating who I am. Authentication, in this case, is about establishing that I am the owner of the Facebook account. The only time *real* authentication enters the picture is in a social media context such as Twitter's authenticated accounts for celebrities.

Our development task is made simpler by the fact that software to handle authorization is frequently the same software that authenticates the individual, so we're not having to deal with two different JavaScript libraries/modules/systems. There are also several excellent OAuth (1.0 and 2.0) modules we can use in Node applications. One of the most popular is Passport, and there are extensions for various authorization services created specifically for the Passport system. However, there are also very simple OAuth clients that provide barebones authorization access for a variety of services, and some modules that are created specifically for one service.

Passport.js is covered in Recipe 21.5. You can also read more about Passport and its various *strategies* supporting different servers at its website.

Now, on to the technology.

Client Credentials Grant

There are few web resources that nowadays provide an API you can access without having some kind of authorization credential. This means having to incorporate a round-trip directive to the end users—asking them to authorize access to their account at the service before the application can access data. The problem is that sometimes all you need is simple read-only access without update privileges, without a frontend login interface, and without having a specific user make an authorizing grant.

OAuth 2.0 accounts for this particular type of authorizing flow with the *Client Credentials Grant*. The diagram for this simplified authorization is shown in Figure 21-2.

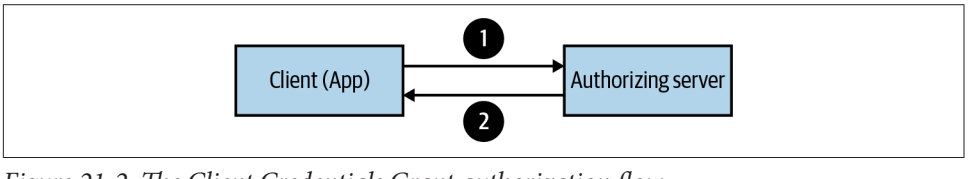

Figure 21-2. The Client Credentials Grant authorization flow

Twitter provides what it calls application-only authorization, which is based on OAuth 2.0's Client Credentials Grant. We can use this type of authorization to access Twitter's Search API.

In the following example, I used the Node module `oauth` to implement the authorization. It's the most basic of the authorization modules, and supports both OAuth 1.0 and OAuth 2.0 authorization flows:

```
const OAuth = require('oauth');
const fetch = require('node-fetch');
const { promisify } = require('util');

// read Twitter keys from a .env file
require('dotenv').config();

// Twitter's search API endpoint and the query we'll be searching
const endpointUrl = 'https://api.twitter.com/2/tweets/search/recent';
const query = 'javascript';

async function getTweets() {
  // consumer key and secret passed in from environment variables
  const oauth2 = new OAuth.OAuth2(
    process.env.TWITTER_CONSUMER_KEY,
    process.env.TWITTER_CONSUMER_SECRET,
    'https://api.twitter.com/',
    null,
    'oauth2/token',
    null
  );

  // retrieve the credentials from Twitter
  const getOAuthAccessToken = promisify(
    oauth2.getOAuthAccessToken.bind(oauth2)
  );
  const token = await getOAuthAccessToken('', {
    grant_type: 'client_credentials'
  });

  // make the request for data with the retrieved token
  const res = await fetch(`${endpointUrl}?query=${query}`, {
    headers: {
      authorization: `Bearer ${token}`
    }
  });

  const json = await res.json();
  return json;
}

(async () => {
  try {
    // Make request
    const response = await getTweets();
    console.log(response);
  } catch (e) {
    console.log(e);
    process.exit(-1);
  }
  process.exit();
})();
```

To use the Twitter authorization API, the client application has to register its application with Twitter. Twitter provides both a *consumer key* and a *consumer secret*.

Using the `oauth` module, a new OAuth2 object is created, passing in:

- Consumer key
- Consumer secret
- API base URI (API URI minus the query string)
- A value of null signals OAuth to use the default */oauth/authorize*
- The access token path
- Null, because we're not using any custom headers

The `oauth` module takes this data and forms a POST request to Twitter, passing along the consumer key and secret, as well as providing a *scope* for the request. Twitter's documentation provides an example POST request for an access token (line breaks inserted for readability):

```
POST /oauth2/token HTTP/1.1
Host: api.twitter.com
User-Agent: My Twitter App v1.0.23
Authorization: Basic eHZ6MWV2RlM0d0VFUFRHRUZQSEJvZzpMOHFxOVBaeVJn
               NmllS0dFS2hab2xHQzB2SldMdzhpRUo4OERSZHlPZw==
               Content-Type: application/x-www-form-urlencoded;charset=UTF-8
Content-Length: 29
Accept-Encoding: gzip

grant_type=client_credentials
```

The response includes the access token (again, line breaks for readability):

```
HTTP/1.1 200 OK
Status: 200 OK
Content-Type: application/json; charset=utf-8
...
Content-Encoding: gzip
Content-Length: 140

{"token_type":"bearer","access_token":"AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
%2FAAAAAAAAAAAAAAAAAAAA%3DAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA"}
```

The access token has to be used with any of the API requests. There are no further authorization steps, so the process is very simple. In addition, since the authorization is at the application level, it doesn't require an individual's authorization, making it less disruptive to the user.

Twitter provides wonderful documentation. I recommend reading the "Application-only authentication overview".

Read/write authorization with OAuth 1.0

Application-Only authentication is great for accessing read-only data, but what if you want to access a user's specific data, or even make a change to their data? Then you'll need the full OAuth authorization. In this section, we'll again use Twitter for the demonstration because of its use of OAuth 1.0 authorization. In the next recipe, we'll look at OAuth 2.0.

I refer to it as OAuth 1.0, but Twitter's service is based on OAuth Core 1.0 Revision A. However, it's a lot easier just to say OAuth 1.0.

OAuth 1.0 requires a digital signature. The steps to derive this digital signature, graphically represented in Figure 21-3, and as outlined by Twitter, are:

1. Collect the HTTP method and the base URI, minus any query string.
2. Collect the parameters, including the consumer key, request data, nonce, signature method, and so on.
3. Create a signature base string, which consists of the data we've gathered, formed into a string in a precise manner, and encoded just right.
4. Create a signing key, which is a combination of consumer key and OAuth token secret, again combined in a precise manner.
5. Pass the signature base string and the signing key to an HMAC-SHA1 hashing algorithm, which returns a binary string that needs further encoding.

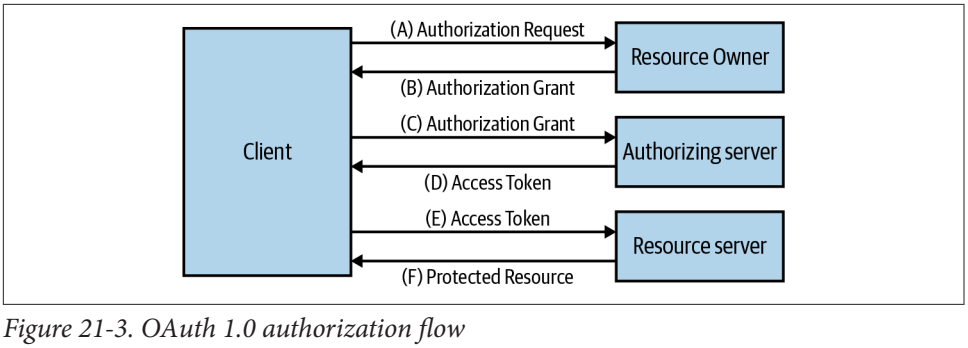

Figure 21-3. OAuth 1.0 authorization flow

You have to follow this process for *every* request. Thankfully, we have modules and libraries that do all of this mind-numbing work for us. I don't know about you, but if I had to do this, my interest in incorporating Twitter data and services into my application would quickly wane.

Our friend `oauth` provides the underlying OAuth 1.0 support, but we don't have to code to it directly this time. Another module, `node-twitter-api`, has wrapped all of the OAuth pieces. All we need do is create a new `node-twitter-api` object, passing in our consumer key and secret, as well as the callback/redirect URL required by the resource services, as part of the authorization process. Processing the `request` object in that URL provides us the access token and secret we need for API access. Every time we make a request, we pass in the access token and secret.

The `twitter-node-api` module is a thin wrapper around the REST API: to make a request, we extrapolate what the function is from the API. If we're interested in posting a status update, the REST API endpoint is:

```
https://api.twitter.com/1.1/statuses/update.json
```

The `twitter-node-api` object instance function is `statuses()`, and the first parameter is the verb, `update`:

```
 twitter.statuses('update', {
        "status": "Hi from Shelley's Toy Box. (Ignore--developing Node app)"
        }, atoken, atokensec, function(err, data, response) {...});

twitter.statuses(
  'update',
  {
    status: 'Ignore learning OAuth with Node'
  },
  tokenValues.atoken,
  tokenValues.atokensec,
  (err, data) => { ... });
```

The callback function arguments include any possible error, requested data (if any), and the raw response.

A complete example is shown in Example 21-1. It uses Express as a server and provides a primitive web page for the user, and then uses another module.

Example 21-1. Twitter app fully authorized via OAuth 1.0

```
const express = require('express');
const TwitterAPI = require('node-twitter-api');

require('dotenv').config();

const port = process.env.PORT || '8080';
```

```
// keys and callback URL are configured in the Twitter Dev Center
const twitter = new TwitterAPI({
  consumerKey: process.env.TWITTER_CONSUMER_KEY,
  consumerSecret: process.env.TWITTER_CONSUMER_SECRET,
  callback: 'http://127.0.0.1:8080/oauth/callback'
});

// object for storing retrieved token values
const tokenValues = {};

// twitter OAuth API URL
const twitterAPI = 'https://api.twitter.com/oauth/authenticate';

// simple HTML template
const menu =
  '<a href="/post/status/">Say hello</a><br />' +
  '<a href="/get/account/">Account Settings<br />';

// Create a new Express application.
const app = express();

// request Twitter permissions when the / route is visited
app.get('/', (req, res) => {
  twitter.getRequestToken((error, requestToken, requestTokenSecret) => {
    if (error) {
      console.log(`Error getting OAuth request token : ${error}`);
      res.writeHead(200);
      res.end(`Error getting authorization${error}`);
    } else {
      tokenValues.token = requestToken;
      tokenValues.tokensec = requestTokenSecret;
      res.writeHead(302, {
        Location: `${twitterAPI}?oauth_token=${requestToken}`
      });
      res.end();
    }
  });
});

// callback url as specified in the Twitter Developer Center
app.get('/oauth/callback', (req, res) => {
  twitter.getAccessToken(
    tokenValues.token,
    tokenValues.tokensec,
    req.query.oauth_verifier,
    (err, accessToken, accessTokenSecret) => {
      res.writeHead(200);
      if (err) {
        res.end(`problems getting authorization with Twitter${err}`);
      } else {
        tokenValues.atoken = accessToken;
        tokenValues.atokensec = accessTokenSecret;
```

```
        res.end(menu);
      }
    }
  );
});

// post a status update from an authenticated and authorized users
app.get('/post/status/', (req, res) => {
  twitter.statuses(
    'update',
    {
      status: 'Ignore teaching OAuth with Node'
    },
    tokenValues.atoken,
    tokenValues.atokensec,
    (err, data) => {
      res.writeHead(200);
      if (err) {
        res.end(`problems posting ${JSON.stringify(err)}`);
      } else {
        res.end(`posting status: ${JSON.stringify(data)}<br />${menu}`);
      }
    }
  );
});

// get account details for an authenticated and authorized user
app.get('/get/account/', (req, res) => {
  twitter.account(
    'settings',
    {},
    tokenValues.atoken,
    tokenValues.atokensec,
    (err, data) => {
      res.writeHead(200);
      if (err) {
        res.end(`problems getting account ${JSON.stringify(err)}`);
      } else {
        res.end(`<p>${JSON.stringify(data)}</p>${menu}`);
      }
    }
  );
});

app.listen(port, () => console.log(`Listening on port ${port}!`));
```

The routes of interest in the app are:

- `/`: Page that triggers a redirect to Twitter for authorization
- `/auth`: The callback or redirect URL registered with the app, and passed in the request
- `/post/status/`: Post a status to the Twitter account
- `/get/account/`: Get account information for the individual

In each case, the appropriate `node-twitter-api` function is used:

- `/`: Get a request token and request token secret, using `getRequestToken()`
- `/auth/`: Get the API access token and token secret, caching them locally, display menu
- `/post/status/`: `status()` with *update* as first parameter, status, access token and secret, and callback function
- `/get/account/`: `account()` with *settings* as the first parameter, an empty object, since no data is needed for the request, and the access token, secret, and callback

The Twitter authorization page that pops up is displayed in Figure 21-4, and the web page that displays account information for yours truly is displayed in Figure 21-5.

Though it is no longer actively maintained, you can read more about the `node-twitter-api` module at its GitHub repository page. Other libraries are more actively maintained and provide the same type of functionality, but I found `node-twitter-api` offers the simplest functional example for the purpose of demonstration.

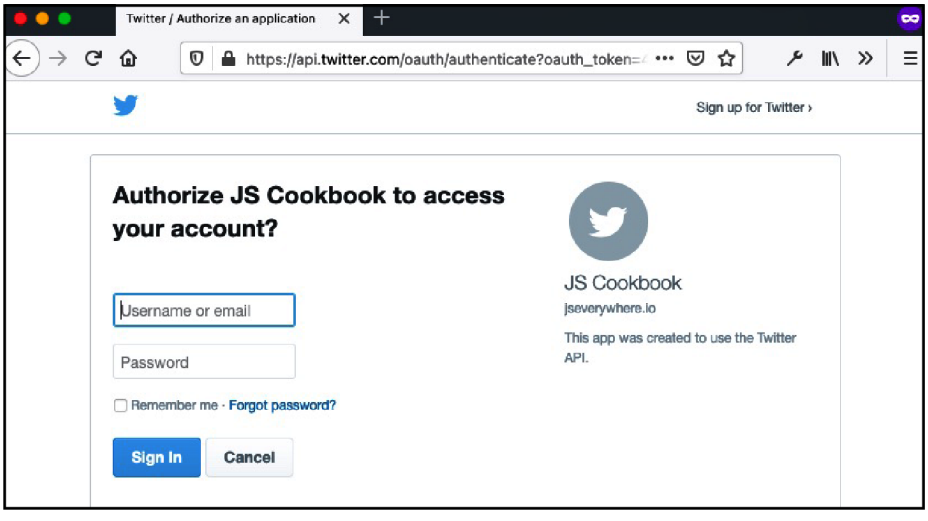

Figure 21-4. Twitter authorization page, redirected from the recipe app

Figure 21-5. Display of Twitter user account data in app

21.5 OAuth 2 User Authentication with Passport.js

Problem

You want to authenticate users in your application through a third-party service.

Solution

Use the Passport.js library paired with the appropriate strategy for the authentication provider you've chosen. In this example, I'll make use of the GitHub strategy, but the workflow will be identical for any OAuth 2 provider, including Facebook, Google, and Twitter.

You can make use of the GitHub strategy, first by visiting GitHub's website and registering a new OAuth application. Once the application is registered, you can integrate the Passport.js OAuth code into the application.

To begin, configure the Passport strategy, which will include the GitHub-provided client ID and client secret, along with the callback URL that you have specified:

```
const express = require('express');
const passport = require('passport');
const { Strategy } = require('passport-github');

passport.use(
  new Strategy(
    {
      clientID: GITHUB_CLIENT_ID,
      clientSecret: GITHUB_CLIENT_SECRET,
      callbackURL: 'login/github/callback'
    },
    (accessToken, refreshToken, profile, cb) => {
      return cb(null, profile);
    }
  )
);
```

To restore authentication state across HTTP requests, Passport needs to serialize and deserialize users:

```
passport.serializeUser((user, cb) => {
  cb(null, user);
});

passport.deserializeUser((obj, cb) => {
  cb(null, obj);
});
```

To preserve user logins across browser sessions, make use of the `express-session` middleware:

```
app.use(
  require('express-session')({
    secret: SESSION_SECRET,
    resave: true,
    saveUninitialized: true
  })
);
```

```
app.use(passport.session());
```

You can then authenticate requests using `passport.authenticate`:

```
app.use(passport.initialize());

app.get('/login/github', passport.authenticate('github'));

app.get(
  '/login/github/callback',
  passport.authenticate('github', { failureRedirect: '/login' }),
  (req, res) => {
    res.redirect('/');
  }
);
```

And reference the `user` object from requests:

```
app.get('/', (req, res) => {
  res.render('home', { user: req.user });
});
```

Discussion

OAuth is an open standard for user authentication. It allows us to authenticate users through third-party applications. This can be useful when allowing users to easily create accounts and log in to your applications, as well as for authenticating to use data from a third-party source.

OAuth requests follow a specific flow:

1. Your application makes an authorization request to the third-party service.
2. The user approves that request.
3. The service redirects the user back to your application, along with an authorization code.
4. The application makes a request to the third-party service with the authorization code.
5. The service responds with an access token (and optionally a refresh token).
6. The application makes a request to the service with the access token.
7. The service responds with the protected resource (in our case, the user account information).

Using Passport.js along with a Passport.js strategy for the OAuth provider simplifies this flow in an Express.js application. In this example, we'll build a small Express application that authenticates with GitHub and persists user logins across sessions.

Once we have registered our application with the service provider, we can begin development by installing the appropriate dependencies:

```
# install general application dependencies
npm install express pug dotenv
# install passport dependencies
npm install passport passport-github
# install persistent user session dependencies
npm install connect-ensure-login express-session
```

To store our OAuth client ID, client secret, and session secret values, we will use a *.env* file. Alternately, you could use a JavaScript file (such as a *config.js* file). It is critical that we not check this file into public source control, and I recommend adding it to your *.gitignore* file. In *.env*:

```
GITHUB_CLIENT_ID=<Your client ID>
GITHUB_CLIENT_SECRET=<Your client secret>
SESSION_SECRET=<A session secret - this can be any value you decide>
```

Next, we'll set up our Express application with Passport.js. In *index.js*:

```
const express = require('express');
const passport = require('passport');
const { Strategy } = require('passport-github');

require('dotenv').config();

const port = process.env.PORT || '3000';

// Configure the Passport strategy
passport.use(
  new Strategy(
    {
      clientID: process.env.GITHUB_CLIENT_ID,
      clientSecret: process.env.GITHUB_CLIENT_SECRET,
      callbackURL: `http://localhost:${port}/login/github/callback`
    },
    (accessToken, refreshToken, profile, cb) => {
      return cb(null, profile);
    }
  )
);

// Serialize and deserialize the user
passport.serializeUser((user, cb) => {
  cb(null, user);
});

passport.deserializeUser((obj, cb) => {
  cb(null, obj);
});

// create the Express application
```

```
const app = express();
app.set('views', `${__dirname}/views`);
app.set('view engine', 'pug');

// use the Express session middleware for preserving user session
app.use(
  require('express-session')({
    secret: process.env.SESSION_SECRET,
    resave: true,
    saveUninitialized: true
  })
);

// Initialize passport and restore the authentication state from the session
app.use(passport.initialize());
app.use(passport.session());

// listen on port 3000 or the PORT set as an environment variable
app.listen(port, () => console.log(`Listening on port ${port}!`));
```

You can then build your view templates, which can access the user data.

In *views/home.pug*:

```
if !user
  p Welcome! Please
    a(href='/login/github') Login with GitHub
else
  h1 Hello #{user.username}!
  p View your
    a(href='/profile') profile
```

In *views/login.pug*:

```
h1 Login
a(href='/login/github') Login with GitHub
```

In *views/profile.pug*:

```
h1 Profile
ul
  li ID: #{user.id}
  li Name: #{user.username}
  if user.emails
    li Email: #{user.emails[0].value}
```

Finally, we can set up our routes in the *index.js* file:

```
app.get('/', (req, res) => {
  res.render('home', { user: req.user });
});

app.get('/login', (req, res) => {
  res.render('login');
});
```

```
app.get('/login/github', passport.authenticate('github'));

app.get(
  '/login/github/callback',
  passport.authenticate('github', { failureRedirect: '/login' }),
  (req, res) => {
    res.redirect('/');
  }
);

app.get(
  '/profile',
  require('connect-ensure-login').ensureLoggedIn(),
  (req, res) => {
    res.render('profile', { user: req.user });
  }
);
```

This example was designed to closely match the Express 4.x Facebook example, which provides well-documented code for working with Express and Facebook authentication. You can view hundreds of additional Passport.js strategies.

21.6 Serving Up Formatted Data

Problem

Instead of serving up a web page or sending plain text, you want to return formatted data, such as XML, to the browser.

Solution

Use Node module(s) to help format the data. For example, if you want to return XML, you can use a module to create the formatted data:

```
const builder = require('xmlbuilder');

const xml = builder
  .create('resources')
  .ele('resource')
  .ele('title', 'Ecma-262 Edition 10')
  .up()
  .ele('url', 'https://www.ecma-international.org/ecma-262/10.0/index.html')
  .up()
  .end({ pretty: true });
```

Then create the appropriate header to go with the data, and return the data to the browser:

```
app.get('/', (req, res) => {
  res.setHeader('Content-Type', 'application/xml');
  res.end(xml.toString(), 'utf8');
});
```

Discussion

Web servers frequently serve up static or server-side generated resources, but just as frequently, what's returned to the browser is formatted data that's then processed in the web page before display.

There are two key elements to generating and returning formatted data. The first is to make use of whatever Node library to simplify the generation of the data, and the second is to make sure that the header data sent with the data is appropriate for the data.

In the solution, the `xmlbuilder` module is used to assist us in creating proper XML. This isn't one of the modules installed with Node by default, so we have to install it using npm, the Node Package Manager:

```
npm install xmlbuilder
```

Then it's a matter of creating a new XML document, a root element, and then each resource element, as demonstrated in the solution. It's true, we could build the XML string ourselves, but that's a pain. And it's too easy to make mistakes that are then hard to discover. One of the best things about Node is the enormous number of modules available to do most anything we can think of. Not only do we not have to write the code ourselves, but most of the modules have been thoroughly tested and actively maintained.

Once the formatted data is ready to return, create the header that goes with it. In the solution, because the document is XML, the header content type is set to `application/xml` before the data is returned as a string.

21.7 Building a RESTful API

Problem

You want to build a REST API using Node.js.

Solution

Use Express with the `app.get`, `app.post`, `app.put`, and `app.delete` methods:

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

const app = express();
const port = process.env.PORT || 3000;
```

```
app.get('/', (req, res) => {
  return res.send('Received a GET HTTP method');
});
app.post('/', (req, res) => {
  return res.send('Received a POST HTTP method');
});
app.put('/', (req, res) => {
  return res.send('Received a PUT HTTP method');
});
app.delete('/', (req, res) => {
  return res.send('Received a DELETE HTTP method');
});
app.listen(port, () => console.log(`Listening on port ${port}!`));
```

Discussion

REST stands for "Representational State Transfer," and is the most common architectural approach for building APIs. REST allows us to interact with a remote data source over HTTP, using the standard HTTP methods of `GET`, `POST`, `PUT`, and `DELETE`. We can make use of the Express routing methods to accept these requests.

In the following example, I'll create several routes that serve as API endpoints. Each endpoint will respond to an HTTP request:

`/todos`
: Will accept a `get` request for a list of todos as well as a `post` request for creating a new todo.

`/todos/:todoId`
: Will accept a `get` request that will return a specific todo as well as a `put` request, which will allow the user to update the todo content or completed state, and a `delete` request, which will delete the specific todo.

With these routes defined, we can develop a REST API that responds to these requests appropriately:

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

const port = process.env.PORT || 3000;
const app = express();
app.use(express.json());
app.use(express.urlencoded({ extended: true }));

// an array of data
let todos = [
  {
    id: '1',
    text: 'Order pizza',
    completed: true
  },
```

```
  {
    id: '2',
    text: 'Pick up pizza',
    completed: false
  }
];

// get the list of todos
app.get('/todos', (req, res) => {
  return res.send({ data: { todos } });
});

// get an individual todo
app.get('/todos/:todoId', (req, res) => {
  const foundTodo = todos.find(todo => todo.id === req.params.todoId);
  return res.send({ data: foundTodo });
});

// create a new todo
app.post('/todos', (req, res) => {
  const todo = {
    id: String(todos.length + 1),
    text: req.body.text,
    completed: false
  };

  todos.push(todo);
  return res.send({ data: todo });
});

// update a todo
app.put('/todos/:todoId', (req, res) => {
  const todoIndex = todos.findIndex(todo => todo.id === req.params.todoId);
  const todo = {
    id: req.params.todoId,
    text: req.body.text || todos[todoIndex].text,
    completed: req.body.completed || todos[todoIndex].completed
  };

  todos[todoIndex] = todo;
  return res.send({ data: todo });
});

// delete a todo
app.delete('/todos/:todoId', (req, res) => {
  const deletedTodo = todos.find(todo => todo.id === req.params.todoId);
  todos = todos.filter(todo => todo.id !== req.params.todoId);
  return res.send({ data: deletedTodo });
});

// listen on port 3000 or the PORT set as an environment variable
app.listen(port, () => console.log(`Listening on port ${port}!`));
```

From the terminal, you can use `curl` to test our responses:

```
# get the list of todos
curl http://localhost:3000/todos

# get an individual todo
curl http://localhost:3000/todos/1

# create a new todo
curl -X POST -H "Content-Type:application/json" /
  http://localhost:3000/todos -d '{"text":"Eat pizza"}'

# update a todo
curl -X PUT -H "Content-Type:application/json" /
  http://localhost:3000/todos/2 -d '{"completed": true }

# delete a todo
curl -X DELETE http://localhost:3000/todos/3
```

Manually testing with `curl` can quickly become tedious. For API development, you may also want to make use of a REST client UI, such as Insomnia or Postman (see Figure 21-6).

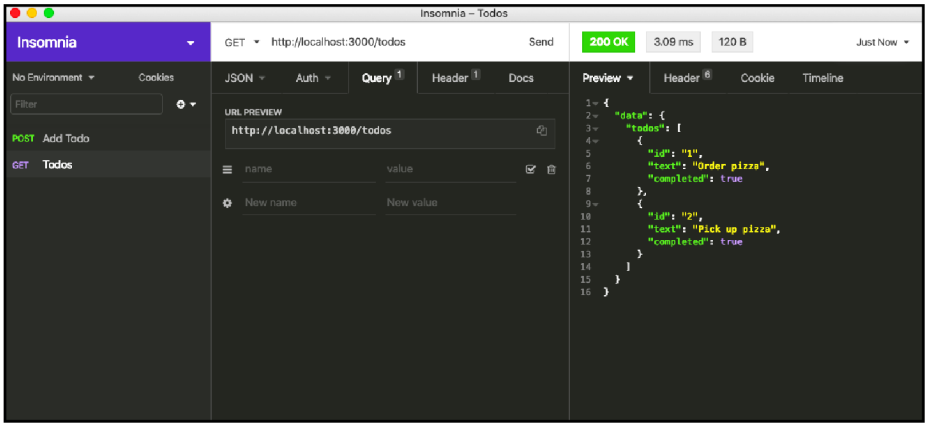

Figure 21-6. A GET request in the Insomnia REST client

In the above example, I'm using an in-memory data store. When building an API, you will most likely want to connect to a database. To do so, you can reach for a library such as Sequelize (for SQL databases), Mongoose (for MongoDB), or an online data store such as Firebase.

21.8 Building a GraphQL API

Problem

You would like to build a GraphQL API server application or add GraphQL endpoints to an existing Express application.

Solution

Use the Apollo Server package to include GraphQL type definitions, GraphQL resolvers, and the GraphQL Playground:

```
const express = require('express');
const { ApolloServer, gql } = require('apollo-server-express');

const port = process.env.PORT || 3000;
const app = express();

const typeDefs = gql`
  type Query {
    hello: String
  }
`;

const resolvers = {
  Query: {
    hello: () => 'Hello world!'
  }
};
const server = new ApolloServer({ typeDefs, resolvers });
server.applyMiddleware({ app, path: '/' });
app.listen({ port }, () => console.log(`Listening on port ${port}!`));
```

Apollo Server provides access to the GraphQL Playground (see Figure 21-7), which allows us to easily interact with the API during development (and in production, if desired).

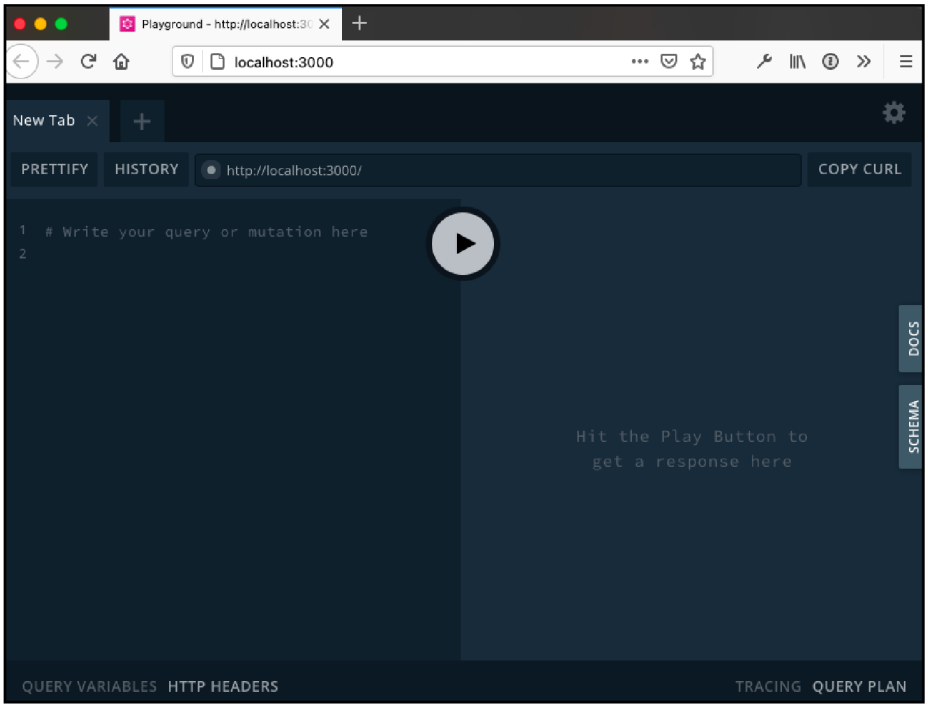

Figure 21-7. A GraphQL query in the GraphQL Playground

The GraphQL Playground also provides automatically generated documentation for the API, based on the type definitions you've provided (see Figure 21-8).

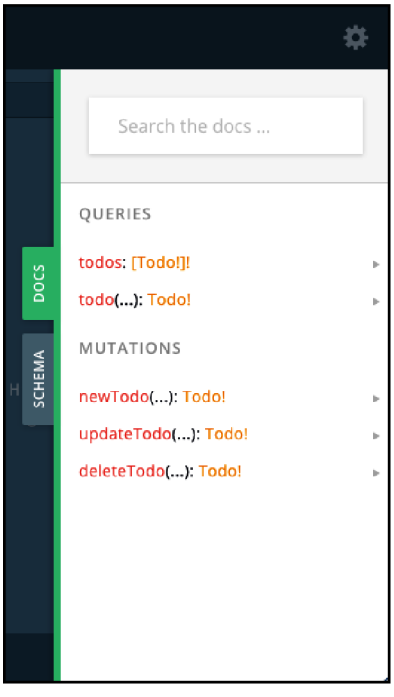

Figure 21-8. The generated documentation in GraphQL Playground

Discussion

GraphQL is an open source query language for APIs. It was developed with the goal of providing single endpoints for data, allowing applications to request the specific data that is needed. Apollo Server can be used as a standalone package or integrated as middleware for popular Node.js server application libraries, such as Express, Hapi, Fastify, and Koa.

In GraphQL, a type definition schema is a written representation of our data and interactions. By requiring a schema, GraphQL enforces a strict plan for our API. This is because your API can only return data and perform interactions that are defined within the schema. The fundamental component of GraphQL schemas are object types. GraphQL contains five built-in scalar types:

- String: A string with UTF-8 character encoding
- Boolean: A true or false value
- Int: A 32-bit integer
- Float: A floating-point value

- ID: A unique identifier

Once the schema is written, we provide the API with a series of resolvers. These are functions that specify how the data should be returned in a query or changed within a data mutation.

In the previous example, we're using the `apollo-server-express` package, which should be installed alongside the `express` and `gql` packages:

```
$ npm install express apollo-server-express gql
```

To create a CRUD application, we can define our GraphQL type definitions and the appropriate resolvers. The following example mimics the one found in Recipe 21.7:

```
const express = require('express');
const { ApolloServer, gql } = require('apollo-server-express');

const port = process.env.PORT || 3000;
const app = express();

// an array of data
let todos = [
  {
    id: '1',
    text: 'Order pizza',
    completed: true
  },
  {
    id: '2',
    text: 'Pick up pizza',
    completed: false
  }
];

// GraphQL Type Definitions
const typeDefs = gql`
  type Query {
    todos: [Todo!]!
    todo(id: ID!): Todo!
  }

  type Mutation {
    newTodo(text: String!): Todo!
    updateTodo(id: ID!, text: String, completed: Boolean): Todo!
    deleteTodo(id: ID!): Todo!
  }

  type Todo {
    id: ID!
    text: String!
    completed: Boolean
  }
`;

// GraphQL Resolvers
```

```
const resolvers = {
  Query: {
    todos: () => todos,
    todo: (parent, args) => {
      return todos.find(todo => todo.id === args.id);
    }
  },
  Mutation: {
    newTodo: (parent, args) => {
      const todo = {
        id: String(todos.length + 1),
        text: args.text,
        completed: false
      };

      todos.push(todo);
      return todo;
    },

    updateTodo: (parent, args) => {
      const todoIndex = todos.findIndex(todo => todo.id === args.id);
      const todo = {
        id: args.id,
        text: args.text || todos[todoIndex].text,
        completed: args.completed || todos[todoIndex].completed
      };

      todos[todoIndex] = todo;
      return todo;
    },
    deleteTodo: (parent, args) => {
      const deletedTodo = todos.find(todo => todo.id === args.id);
      todos = todos.filter(todo => todo.id !== args.id);
      return deletedTodo;
    }
  }
};

// Apollo + Express server setup
const server = new ApolloServer({ typeDefs, resolvers });
server.applyMiddleware({ app, path: '/' });
app.listen({ port }, () => console.log(`Listening on port ${port}!`));
```

In the above example, I'm using an in-memory data store. When building an API, you will most likely want to connect to a database. To do so, you can reach for a library such as Sequelize (for SQL databases), Mongoose (for MongoDB), or an online data store such as Firebase.

The defined queries return data directly from the API, while the mutations allow us to perform changes to the data, such as create a new item, update an item, or delete an item.

About the Authors

Adam D. Scott is an engineering leader, web developer, educator, and artist based in Connecticut. He has worked at the crossroads of technology and education for over a decade, teaching and writing curriculum on a range of technical topics. This is his seventh book.

Matthew MacDonald is a tech writer and long-ago Microsoft MVP who's written enough heavy books to prop open all the doors in his house. Visit his website to learn about his free JavaScript book for kids, or to follow his semi-regular hot-takes programming publication, *Young Coder*.

Shelley Powers has been working with, and writing about, web technologies—from the first release of JavaScript to the latest graphics and design tools—for more than 12 years. Her recent O'Reilly books have covered the semantic web, Ajax, JavaScript, and web graphics. She's an avid amateur photographer and web development aficionado, who enjoys applying her latest experiments on her many websites.

Colophon

The bird on the cover of *JavaScript Cookbook* is a little egret (*Egretta garzetta*). This small white heron, the smallest and most common in Singapore, is a lot like the new world snowy egret. Its original breeding distribution included the large inland and coastal wetlands in warm temperate parts of Europe, Asia, Africa, Taiwan, and Australia. Little egrets in warmer locations are permanent residents, while the northern birds migrate to Africa and southern Asia.

Adult little egrets are 55–65 cm long with an 88–106 cm wingspan and weigh 350–550 grams. Their plumage is all white. They have long black legs, yellow feet, and slim black bills. In the breeding season, adults have two long nape plumes, gauzy plumes on their backs and breasts, and red or blue skin between their bills and eyes.

Little egrets are lively hunters with a wide variety of techniques: they patiently stalk prey in shallow waters; stand on one leg and stir the mud with the other to scare up prey; and stand on one leg and wave the other foot over the water's surface as a lure. They eat fish, insects, amphibians, crustaceans, and reptiles. They nest in colonies on platforms of sticks in trees or shrubs, reed beds, or bamboo groves, often with other wading birds. Many of the animals on O'Reilly covers are endangered; all of them are important to the world.

The cover illustration is by Karen Montgomery, based on a black and white engraving from Cassell's *Natural History*. The cover fonts are Gilroy Semibold and Guardian Sans. The text font is Adobe Minion Pro; the heading font is Adobe Myriad Condensed; and the code font is Dalton Maag's Ubuntu Mono.