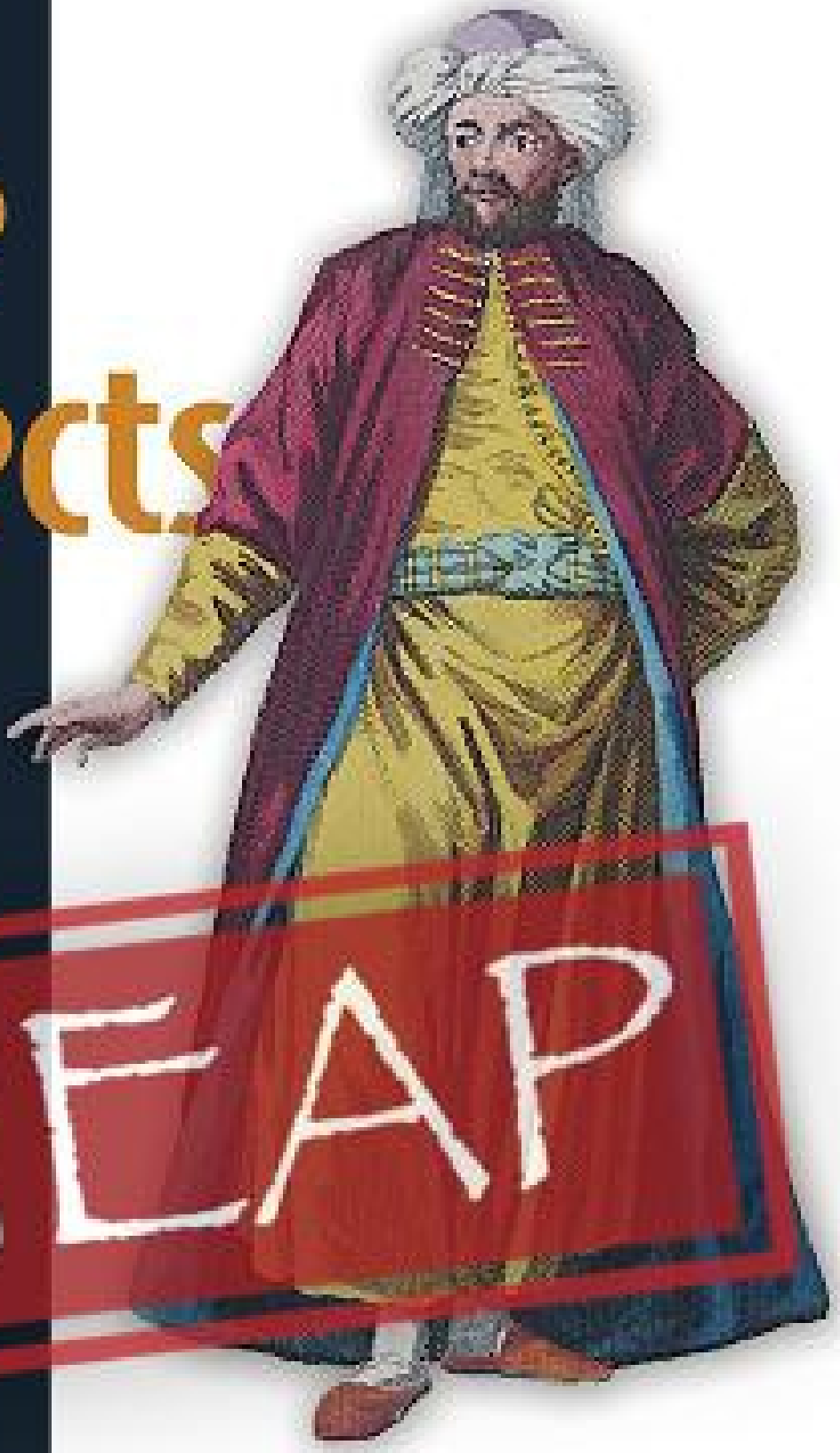
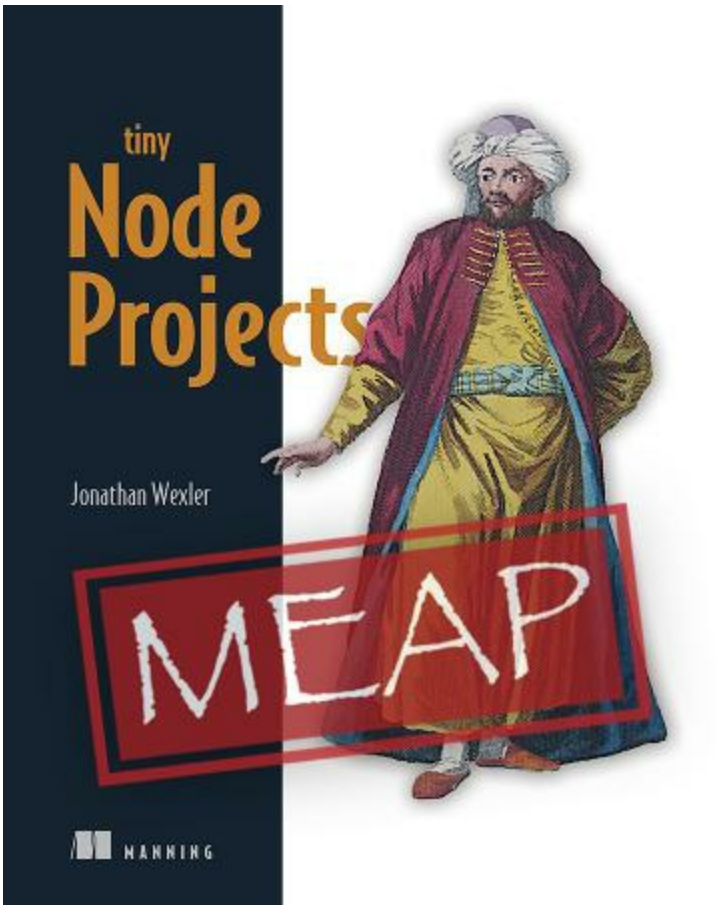
tiny Node Projects

Jonathan Wexler



MEAP

MANNING


tiny
Node
Projects
Jonathan Wexler
MEAP
MANNING

Tiny Node Projects MEAP V03

tiny Node Projects

Jonathan Wexler

MEAP

MANNING

MEAP VERSION 3

Welcome

Thank you for purchasing the MEAP for *Tiny Node Projects*. To get the most out of this book, you'll want to have some familiarity with ES6 style JavaScript syntax and concepts. It will help to have some understanding of how web protocols like HTTP work and the basics of a web server. Knowledge of HTML and CSS are not required but may assist you in enriching your projects. Read this book with creative energy to explore Node.js and you'll be prepared to apply the concepts best in practice.

When I first wrote *Get Programming with Node.js* I had envisioned a style of learning I had not yet seen in a book's format: a 3-month coding boot camp intensive course. I designed that book to bring entry-level developers to an intermediate-level understanding of Node.js web applications. Readers of that book were receptive to the style, encouraging its use to propel one's career in JavaScript development and even as a standard in college-level coding courses. While I enjoyed the writing process, Node.js, and JavaScript, were still moving quickly. I knew that by the time that book was published there would be new popular changes to the language and leading npm packages to recommend for use in projects. Fast forward to today, the Tiny Projects series presents a unique opportunity well suited for Node.js projects. *Tiny Node Projects* gives me the opportunity to dive into meaningful application architecture concepts across multiple types of projects.

No longer limited to teaching one big web application, Tiny Node Projects separates each chapter so that you can pick and choose what you want to learn, when you want to learn it. The projects are designed to be built in a single day and I'll be adding content-specific guides so that you can choose when you want to purely read and learn, and when you want to apply and practice. I want these tiny projects to range from quick wins for new developers, to challenging coding obstacles or new terrain for experienced developers. That's why you are an important part of shaping the book's final form.

As you read each chapter, pay close attention to what your learning and how

it can apply across many projects. Write down some of the questions you feel are unanswered so I can help address them. Think about concepts you enjoy and would like to see more of an in depth conversation. It's important to me that you enjoy working through this book and that you find it to be a helpful and guiding resource on your journey with Node.js and server-side JavaScript development. The goal of this book is to help you realize just how versatile Node.js is for building your own creative solutions to life's problems. You'll start with some introductory projects that can be built with Node.js' prepackaged modules. Then you'll move on to working with external npm packages that deliver highly performant algorithms and functions. Because this book is not focused on the web front, many of the projects can be built for a command line client. Though, there will be steps to support any additional surfaces like web browsers and mobile devices.

I am grateful that you've chosen to give Tiny Node Projects a chance, and I hope it proves to be a consistent source of insight and encouragement in this MEAP and beyond. Thank you, and enjoy! If you have any questions, comments, or suggestions, please share them in Manning's liveBook discussion forum.

Jon Wexler

In this book

1 Practical Application

This chapter covers

- Getting started with a Node app
- Reading user input on the command line
- Writing data to a CSV

Whether you are new to programming or a seasoned engineer, you probably opened this chapter to get started with Node already. Many Node projects span thousands of lines of code, but some of the most useful and practical applications can be written in only a handful of lines. Let’s skip the part where you get overwhelmed and jump into the *practical application*.

In this chapter, you’ll get a first glance at what Node can offer out of the box and off the grid. You will learn to build a simple program that can run on anyone’s computer and save real people real time and money. By the end of the chapter, you’ll have Node locked and loaded on your computer, with a development environment ready to build, *practically*, anything.

Tools & applications used in this chapter

Before you get started, you’ll need to install and configure the following tools and applications that are used in this project. Detailed instructions are provided for you in the specified appendix. When you’ve finished, return here and continue.

- Appendix A.1.2 Installing VS Code
- Appendix A.1.3 Installing Node

1.1 Your prompt

A travel agent wants to hire you to convert their Rolodex (physical information cards) into some digital format. They just bought a few new

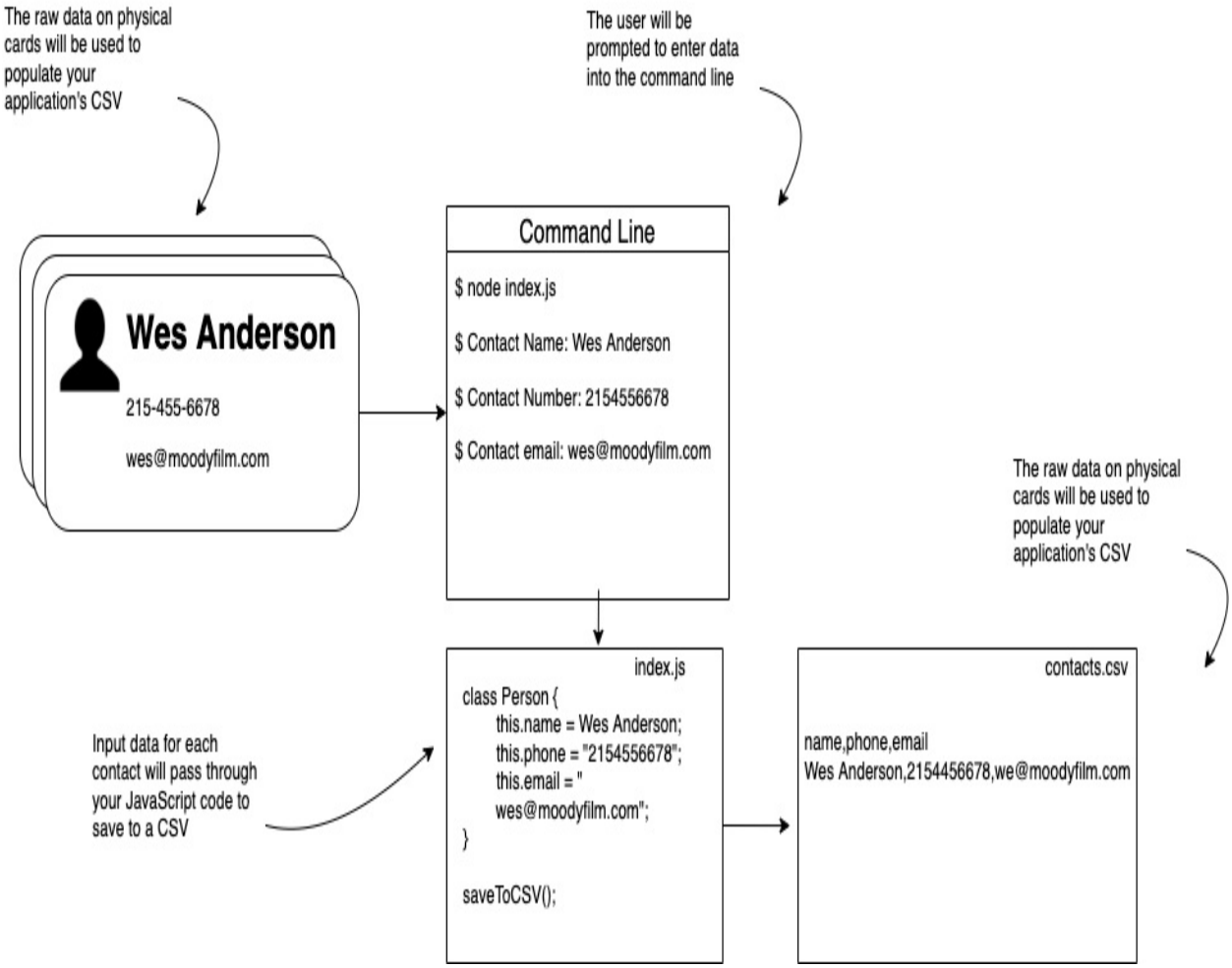
computers and want to start importing data. They don’t need anything fancy like a website or mobile application, just a computer prompt to create a tabular format (CSV) of data.

1.1.1 Get planning

Luckily, you just started building Node applications and you see this as a great opportunity to use some of Node’s default, out of the box, libraries. This business only wants to manually enter data that can be written to a `.csv` file. So you start diagramming the requirements of the project and your result is shown in figure 1.1. The diagram details the following steps:

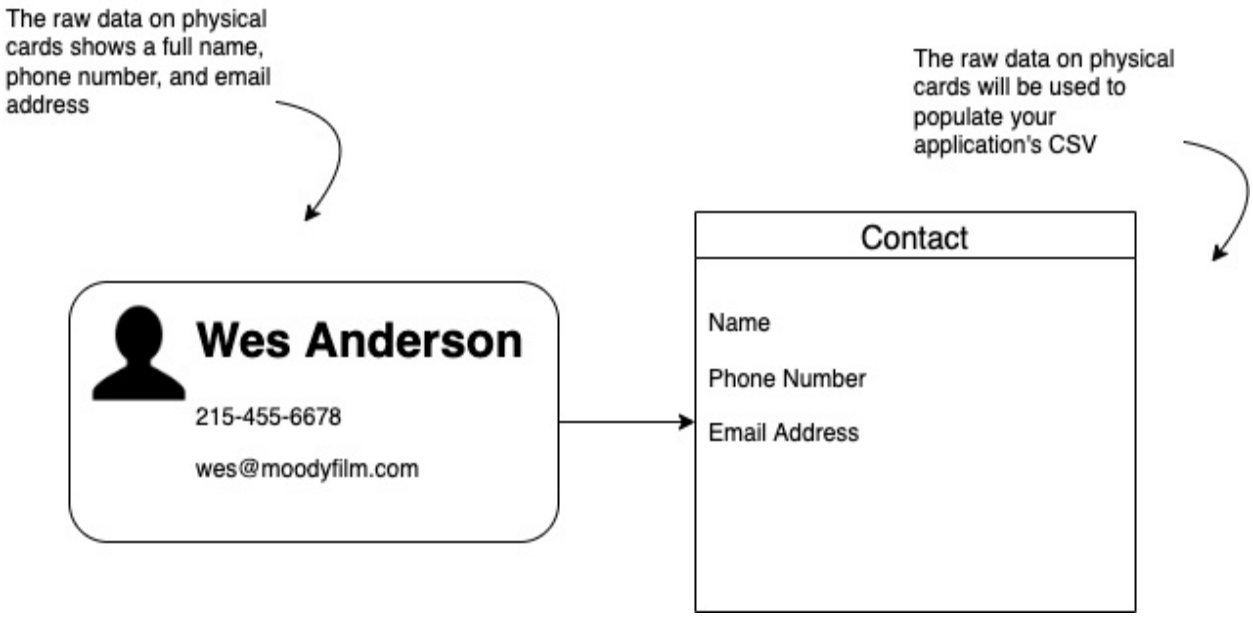
1. Physical information cards are collected by the user.
2. Open the command line and start the application by running `node index`.
3. Enter the name, phone number, and email address in the command line prompt.
4. After each entry, contact data is saved through your application to a CSV file on your computer.

Figure 1.1. Diagramming the project blueprint and flow of information



Each contact has a full name, phone number, and email address listed on their card. You decide to build a simple Node command line application so users can manually enter these three data values for each contact card. The data you will digitize is shown in figure 1.2.

Figure 1.2. Application data required for development



When you're finished, the user should be able to open their command line, run `node index.js` and follow the prompts to enter contact information. After each contact is entered, it will be saved to a CSV file as a comma-delimited string.

Note

`.csv` is a precursor to many of the mainstream database tables used today. While not ideal for storing long-term data, it's a great way to visualize rows of data for viewing and processing without a database.

1.1.2 Get programming

With Node set up, you begin setting up your project. Choose a location where you'd like to store your project code for this chapter and run `mkdir csv_writer` in your command line to create the project folder.

Tip

To keep your coding projects separate from other work on your computer, you can dedicate a directory for coding. Creating a directory called `src` at the

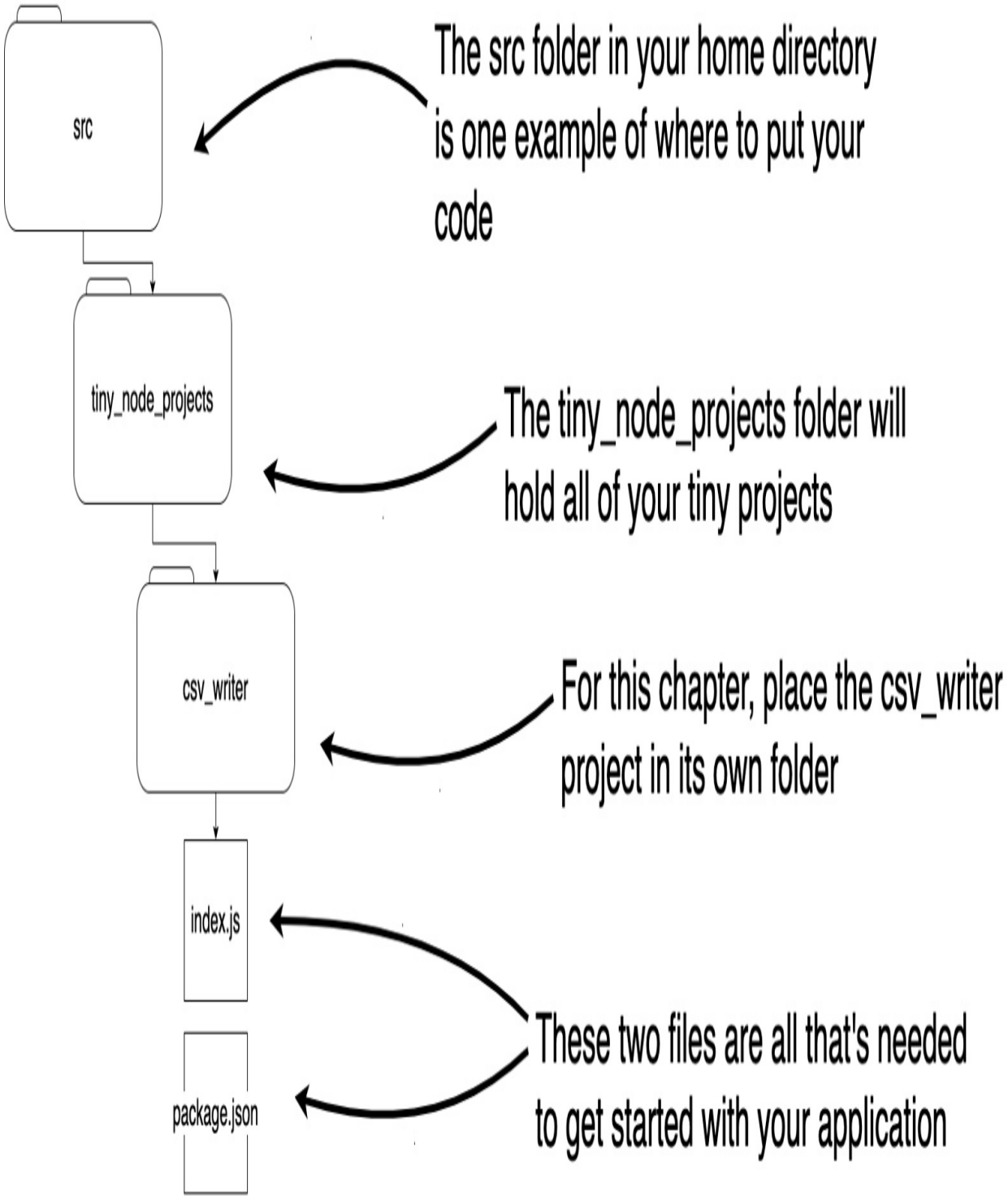
root level of your computer's user directory ids a good place. That location is `/Users/<USERNAME>/src` for Macs (`~/src`), and `C:\Users\<USERNAME>\src` for Windows computers.

Next, you'll follow steps to initialize your Node app. When complete, your project directory structure should look like figure 1.3.

 **Note**

These steps are also available in Appendix B: Setting up Node app essentials.

Figure 1.3. Project directory structure



Within the `csv_writer` folder run `npm init` to start the process of creating a new Node application configuration file. You will see a prompt and can fill

out your responses as in listing 1.1.

Listing 1.1. Prompt for `npm init`

```
javascript
package name: (csv_writer) #1
version: (1.0.0)
description: An app to write contact information to a csv file.
entry point: (index.js) #2
test command:
git repository:
keywords:
author: Jon Wexler
license: (ISC)
```

This process has created a new file for you: `package.json` (listing 1.2). This is your application's configuration file, where you'll instruct your computer on how to run your Node app.

What's `package.json`?

Every Node project contains it's own unique arrangement of libraries, configurations, and metadata that allows it to operate. Just like how a baker might need a kitchen, oven, and cooking instruments to follow a recipe for baking a loaf of bread, a Node app needs the appropriate conditions before it follows its set of instructions (your code) to run.

Your project's `package.json` file provides everything an engineer needs to get the Node app running in the exact condition it was left from the previous engineer. If your project requires code from a third-party library, the `package.json` will specify the name of that library. If you specify scripts that must run before you start your app, the `package.json` file will list those scripts and their alias names. Last, if your project code will be made available to the public, the `package.json` file will contain your name, project description, and licensing information. The following are some of the most significant fields of a `package.json` file in mainstream applications:

- `dependencies`-- This section defines the libraries on which your project depends. For example, if you want to incorporate a Google Maps API to your app, you may add the `@google/maps` package to this portion of the

file. The next person to download your project code, will then know to install this library ahead of running the app.

- `scripts`—This section is where you specify commands to run certain scripts for your application. You may define build scripts for precompiling HTML-ready content for a web app. More commonly, you'll define a `start` script, which is essentially how you'll start your node app: `npm start`.
- `version`—This section is where you may define various versions of your app throughout the development process. As you modify your app, you'll make major, minor, and patch updates reflected in the version number.

In addition to giving a baker a recipe for a loaf of bread, you would instruct them on the type of oven, materials, and environment to use. The `package.json` file contains those preparatory instructions and more. For more information, visit https://nodejs.org/en/knowledge/getting-started/npm/what-is-the-file-package-json/.

Your first step is to add `"type": "module"` to this file so we can use ES6 module imports. Since Node v13.2.0, ES6 modules have had stable support and growing popularity in Node projects over CommonJS syntax for importing code.

Listing 1.2. Contents of `package.json`

```
javascript
{
  "name": "contact-list",
  "version": "1.0.0",
  "type": "module", #1
  "description": "An app to write contact information to a csv fi
  "main": "index.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "author": "Jon Wexler",
  "license": "ISC"
}
```

Next, you create the entry point to your application, a file titled `index.js`.

This is where the guts of your application will go.

For this project, you realize that Node comes prepackaged with everything you need already. You can make use of the `fs` module - a library that helps your application interact with your computer's file system - to create the CSV file. Within `index.js` you add the `writeFileSync` function from the `fs` module by writing `import { writeFileSync } from "fs"`.

You now have access to functions that can create files on your computer. Now you can make use of the `writeFileSync` function to test that your Node app can successfully create files and add text to them. Add the code in listing 1.3 to `index.js`. Here you are creating a variable called `content` with some dummy text. Then, within a `try-catch` block, you run `writeFileSync`, a synchronous, blocking function, to create a new file called `test.txt` within your project's directory. If the file is created with your added content, you will see a message logged to your command line window with the text `Success!` Otherwise, if an error occurs, you'll see the stacktrace and contents of that error logged to your command line window.

 **Note**

With ES6 module import syntax you may make use of destructuring assignment. Instead of importing an entire library, you may import the functions or modules you need only through destructuring. For more information, read https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Destructuring_assignment.

Listing 1.3. Contents of `index.js`

```
javascript
import { writeFileSync } from "fs"; #1

const content = "Test content!"; #2

try { #3
  writeFileSync("./test.txt", content) #4
  console.log("Success!"); #5
} catch (err) {
  console.error(err); #6
```

```
}
```

You're ready to test this by navigating to your `csv_writer` project folder within your terminal window and running `node index.js`. Now, check to see if a new file called `text.txt` was created. If so, open it up to see `Test content!` within. Now you're ready to move to the next step and cater this app to handle user input and write to a CSV file.

1.2 Translating user input to CSV

Now that your app is set up to save content to a file, you begin writing the logic to accept user input in the command line. Node comes with a module called `readline` that does just that. By importing the `createInterface` function, you can map the application's standard input and output to the `readline` functionality as seen in listing 1.4. Like the `fs` module, no additional installations are required other than the default out-of-the-box Node installation on your computer.

In this code, `process.stdin` and `process.stdout` are ways that your Node app's `process` streams data to and from the command line and filesystem on your computer.

Listing 1.4. Mapping input and output to `readline` in `index.js`

```
javascript
import { createInterface } from "readline";  #1

const readline = createInterface({  #2
  input: process.stdin,
  output: process.stdout
});
```

Next, you make use of this mapping by using the `question` function in your `readline` interface. This function takes a `message`, or prompt you'll display to the user on the command line, and will return the user's input as a return value.

 **Note**

Because this function is asynchronous, you can wrap its return value in a Promise to make use of the async/await syntax from ES6. If you need a refresher on JavaScript Promises, visit https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Promise

Create a function called `readLineAsync` that waits for the user to reply and press the enter key before the string value is resolved (listing 1.5). In this way, your custom `readLineAsync` function will eventually resolve with a response containing the user's input without holding up the Node app.

Listing 1.5. Promise wrapped `readLineAsync` function in `index.js`

```
javascript
const readLineAsync = message => { #1
  return new Promise(resolve => { #2
    readline.question(message, answer => { #3
      resolve(answer); #4
    });
  });
}
```

With these functions in place, all you need is to call `readLineAsync` for your application to start retrieving user input. Because your prompt has three specific data values to save, you can create a class to encapsulate that data for each contact. As seen in listing 1.6, within `index.js` you import `appendFileSync` from the `fs` module, which will create and append to a given file name. Then you define a `Person` class which takes `name`, `number`, and `email` as arguments in its constructor. Finally, you add a `saveToCSV` method to the `Person` class to save each contact's information in a comma-delimited format suitable for CSV to a file called `contacts.csv`.

Listing 1.6. Defining the `Person` class in `index.js`

```
javascript
import { appendFileSync } from "fs"; #1

class Person { #2
  constructor(name = "", number = "", email = "") {  #3
    this.name = name;
    this.number = number;
    this.email = email;
```

```
  }
  saveToCSV() { #4
    const content = `${this.name},${this.number},${this.email}\n`
    try {
      appendFileSync("./contacts.csv", content); #6
      console.log(`${this.name} Saved!`);
    } catch (err) {
      console.error(err);
    }
  }
}
```

The last step is to instantiate a new `Person` object for each new contact you're manually entering into your application. To do that you create an async `startApp` function that defines the new `person` object and assigns the `name`, `number`, and `email` values in synchronous order. This way you way for the user input to collect each value before moving to the next one. After all the required values are collected, you call `saveToCSV()` on the `person` instance and ask the user if they would like to continue entering more data. If so, they can enter the letter `y`. Otherwise, you close the `readline` interface and end your application (listing 1.7).

Listing 1.7. Collecting user input within `startApp` in `index.js`

```
javascript
const startApp = async () => { #1
  const person = new Person(); #2
  person.name = await readLineAsync("Contact Name: "); #3
  person.number = await readLineAsync("Contact Number: ");
  person.email = await readLineAsync("Contact Email: ");
  person.saveToCSV(); #4
  const response = await readLineAsync("Continue? [y to continue]
  if (response === "y") await startApp(); #6
  else readline.close(); #7
};
```

Then add `startApp()` at the bottom of `index.js` to start the app when the file is run. Your final `index.js` file should look like listing 1.8.

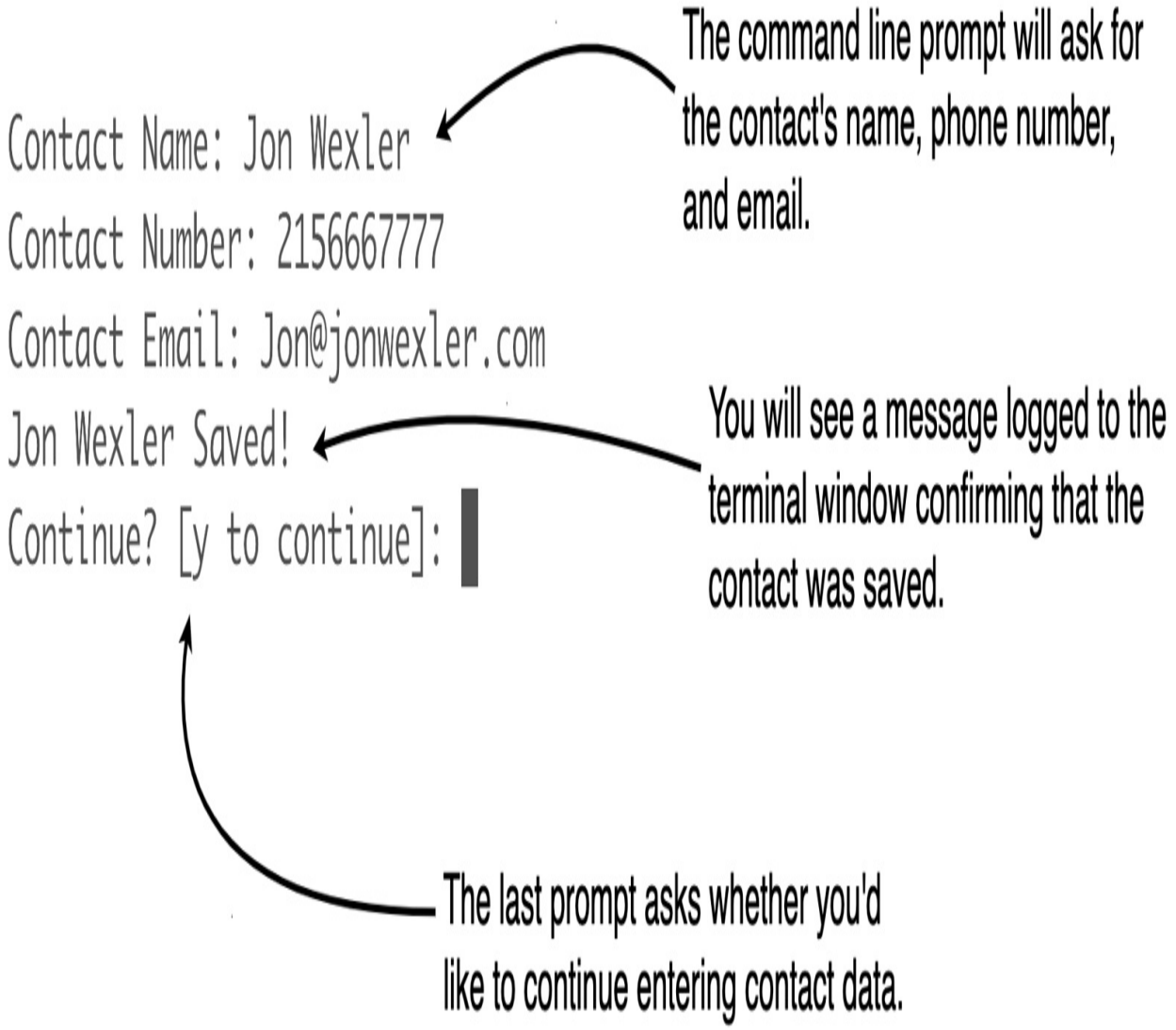
Listing 1.8. Complete `index.js`

```
javascript
import { appendFileSync } from "fs";
```

```
import { createInterface } from "readline";

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

const readLineAsync = (message) => {
  return new Promise((resolve) => {
    readline.question(message, (answer) => {
      resolve(answer);
    });
  });
};

class Person {
  constructor(name = "", number = "", email = "") {
    this.name = name;
    this.number = number;
    this.email = email;
  }
  saveToCSV() {
    const content = `${this.name},${this.number},${this.email}\n`
    try {
      appendFileSync("./contacts.csv", content);
      console.log(`${this.name} Saved!`);
    } catch (err) {
      console.error(err);
    }
  }
}

const startApp = async () => {
  const person = new Person();
  person.name = await readLineAsync("Contact Name: ");
  person.number = await readLineAsync("Contact Number: ");
  person.email = await readLineAsync("Contact Email: ");
  person.saveToCSV();
  const response = await readLineAsync("Continue? [y to continue]
  if (response === "y") await startApp();
  else readline.close();
};

startApp();
```

In the project folder on your command line, run `node index` to start seeing text prompts as seen in figure 1.4.

Figure 1.4. Command line prompts for user input



When you are done entering all the contact's details, you can then see that the information has been saved to a file called `contacts.csv` in the same folder. Each line of that file should be comma-delimited, looking like `Jon Wexler,2156667777,Jon@jonwexler.com`.

This should be just what the travel agency needs for now to convert their physical contact cards into a CSV file they can use in many other ways. In the next section, you'll explore how third-party libraries can simplify your

code even further.

1.3 Working with external packages

It didn't take much code to build an application that is as functional and effective as `csv_writer`. The good news is your work is done. There's even better news! While Node offers built-in modules for a variety of tasks, there are plenty of external libraries (npm packages) you can install to reduce your written code even further.

To improve the readability of your code from section 2, you can install the `prompt` and `csv-writer` packages by running `npm i prompt csv-writer` in your project folder on your command line. This command will also list these two packages in your `package.json` file.

 **Note**

You can read more about `prompt` at npmjs.com/package/prompt and `csv-writer` at npmjs.com/package/csv-writer.

In your `index.js` file import `prompt` and replace your `readLineAsync` calls with `prompt` as seen in listing 1.9. You first instantiate the `prompt` keyword with `prompt.start`, followed by setting the prompt message to an empty string. Now you can delete your whole `readLineAsync` function, realine interface mappings, and `readline` module imports. `prompt.get` allows the user to respond to multiple prompts before returning the resulting values back to the Node app. The user's responses are assigned to a `responses` object with each prompt's response matching the prompt's name. `Object.assign` sets the `name`, `number`, and `email` response fields on the `person` object. After the `person's values are saved to a CSV, another prompt to continue follows. This time, the prompt's response is destructured and assigned to the `again` variable.

Listing 1.9. Replacing `readLineAsync` with `prompt` in `index.js`

```
js
import prompt from "prompt";  #1
```

```
prompt.start();  #2
prompt.message = ""; #3

const startApp = async () => {
  const person = new Person();
  const responses = await prompt.get([ #4
    {
      name: "name",
      description: "Contact Name",
    },
    {
      name: "number",
      description: "Contact Number",
    },
    {
      name: "email",
      description: "Contact Email",
    },
  ]);
  Object.assign(person, responses); #5
  person.saveToCSV();
  const { again } = await prompt.get([ #6
    {
      name: "again",
      description: "Continue? [y to continue]",
    },
  ]);
  if (again === "y") await startApp();
};
...
```

Similar to how the external `prompt` package displaced the `fs` module, `csv-writer` replaces the need of your `fs` module imports and define a more structured approach for writing to your CSV by including a header, as shown in listing 1.10.

Listing 1.10. Importing and setting up `csv-writer` in `index.js`

```
js
import { createObjectCsvWriter } from "csv-writer"; #1
...
const csvWriter = createObjectCsvWriter({ #2
  path: "./contacts.csv",
  append: true,
  header: [
    { id: "name", title: "NAME" },
```

```
    { id: "number", title: "NUMBER" },
    { id: "email", title: "EMAIL" },
  ],
});
```

Finally, you modify your `saveToCSV` method on the `Person` class to use `csvWriter.writeRecords` instead (listing 1.11).

Listing 1.11. Update `saveToCSV` in `Person` class to use `csvWriter.writeRecords` in `index.js`

```
js
...
saveToCSV() {
  try {
    const { name, number, email } = this; #1
    csvWriter.writeRecords([{ name, number, email }]); #2
    console.log(`${name} Saved!`);
  } catch (err) {
    console.error(err);
  }
}
...
```

With these two changes in place your new `index.js` file should look like listing 1.12.

Listing 1.12. Using external packages in `index.js`

```
javascript
import { createObjectCsvWriter } from "csv-writer";
import prompt from "prompt";
prompt.start();
prompt.message = "";

const csvWriter = createObjectCsvWriter({
  path: "./contacts.csv",
  append: true,
  header: [
    { id: "name", title: "NAME" },
    { id: "number", title: "NUMBER" },
    { id: "email", title: "EMAIL" },
  ],
});

class Person {
```

```
  constructor(name = "", number = "", email = "") {
    this.name = name;
    this.number = number;
    this.email = email;
  }
  saveToCSV() {
    try {
      const { name, number, email } = this;
      csvWriter.writeRecords([{ name, number, email }]);
      console.log(`${name} Saved!`);
    } catch (err) {
      console.error(err);
    }
  }
}

const startApp = async () => {
  const person = new Person();
  const responses = await prompt.get([
    {
      name: "name",
      description: "Contact Name",
    },
    {
      name: "number",
      description: "Contact Number",
    },
    {
      name: "email",
      description: "Contact Email",
    },
  ]);
  Object.assign(person, responses);
  person.saveToCSV();
  const { again } = await prompt.get([
    {
      name: "again",
      description: "Continue? [y to continue]",
    },
  ]);
  if (again === "y") await startApp();
};

startApp();
```

Now when you run `node index`, the application's behavior should be exactly the same as in section 2. This time your `contacts.csv` file should list headers

at the top of the file. This is a great example of how you can use Node out of the box to solve a real-world problem, then refactor and improve your code by using external packages built by the thriving online Node community!

1.4 Summary

In this chapter you

- Built your first Node app
- Created and wrote to a CSV file via the command line
- Installed and imported external packages to your project

2 Building a Node web server

This chapter covers

- Using Node as a web interface
- Building a Node app with Express.js
- Serving static pages with dynamic content

Node is about using JavaScript on the server. JavaScript itself is already an asynchronous language by nature, but not until 2009 was it used outside your standard web browser. As dependence on the internet grew worldwide, businesses demanded new innovative development strategies that also took into account hirable skill sets already in the market. Thereafter, JavaScript took off for both frontend and backend development, setting up new application design patterns with Node's single-threaded event loop.

In this chapter, you'll explore the most common use-case for Node, a web application, and how the event loop plays a role. By the end of this chapter, you'll be able to use Node's most popular application framework, Express.js, to build both simple web servers and more extensive applications.

Tools & applications used in this chapter

Before you get started, you'll need to install and configure the following tools and applications that are used in this project. Detailed instructions are provided for you in the specified appendix. When you've finished, return here and continue.

- Appendix A.1.2 Installing VS Code
- Appendix A.1.3 Installing Node

2.1 Your prompt

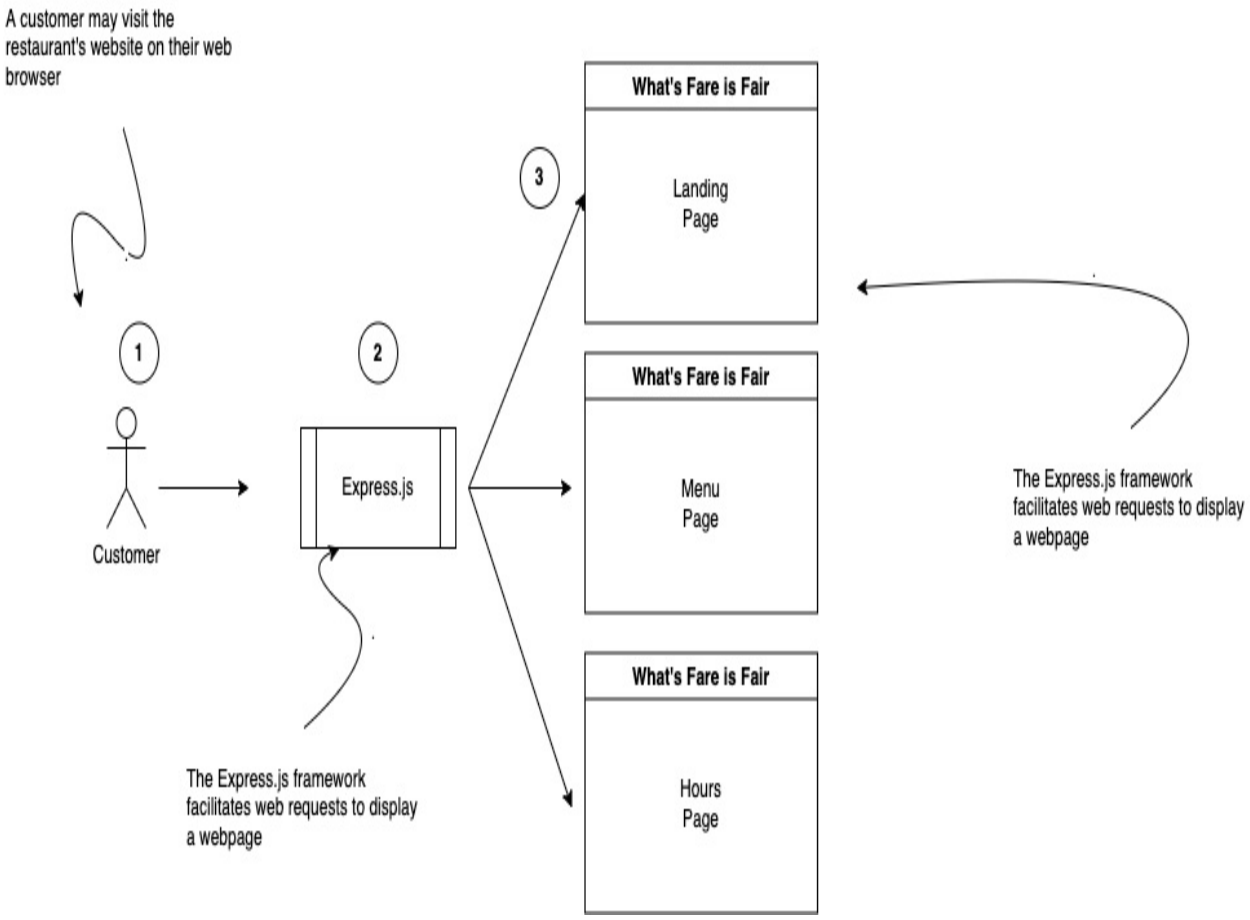
A local restaurant, What's Fare is Fair, has decided to invest some money

into a home-grown web application to better serve its customers. They've reached out to you, an eager software engineer, to help craft a lightweight application that can display their landing page, static menu, and working hours.

2.1.1 Get planning

After careful consideration, you realize this business needs only a simple web server to serve static content to their customers. In this case you only need to support three web pages of content. Being an experienced Node engineer, you know that you may use the built-in `http` module, but find it more flexible to work with a popular external library called Express.js. Before you get programming, you diagram the requirements of the project and your result, as shown in figure 2.1.

Figure 2.1. Project blueprint



A word on web servers

Like many other platforms, Node's runtime environment can create an application that can communicate across the internet via various standard protocols. The most common is Hypertext Transfer Protocol (HTTP). Packages like `http` come prepackaged with Node, but require a lot of code to set up a basic website.

For that reason, packages like `express` offer a full web application framework. This framework not only implements `http` to support web requests and responses, but it offers an intuitive structure to organizing your application's files, importing other supportive packages, and building web applications in a shorter time frame. In fact, many other web frameworks for Node use Express as a foundation for their additional tooling. To learn more about what Express offers visit https://expressjs.com/.

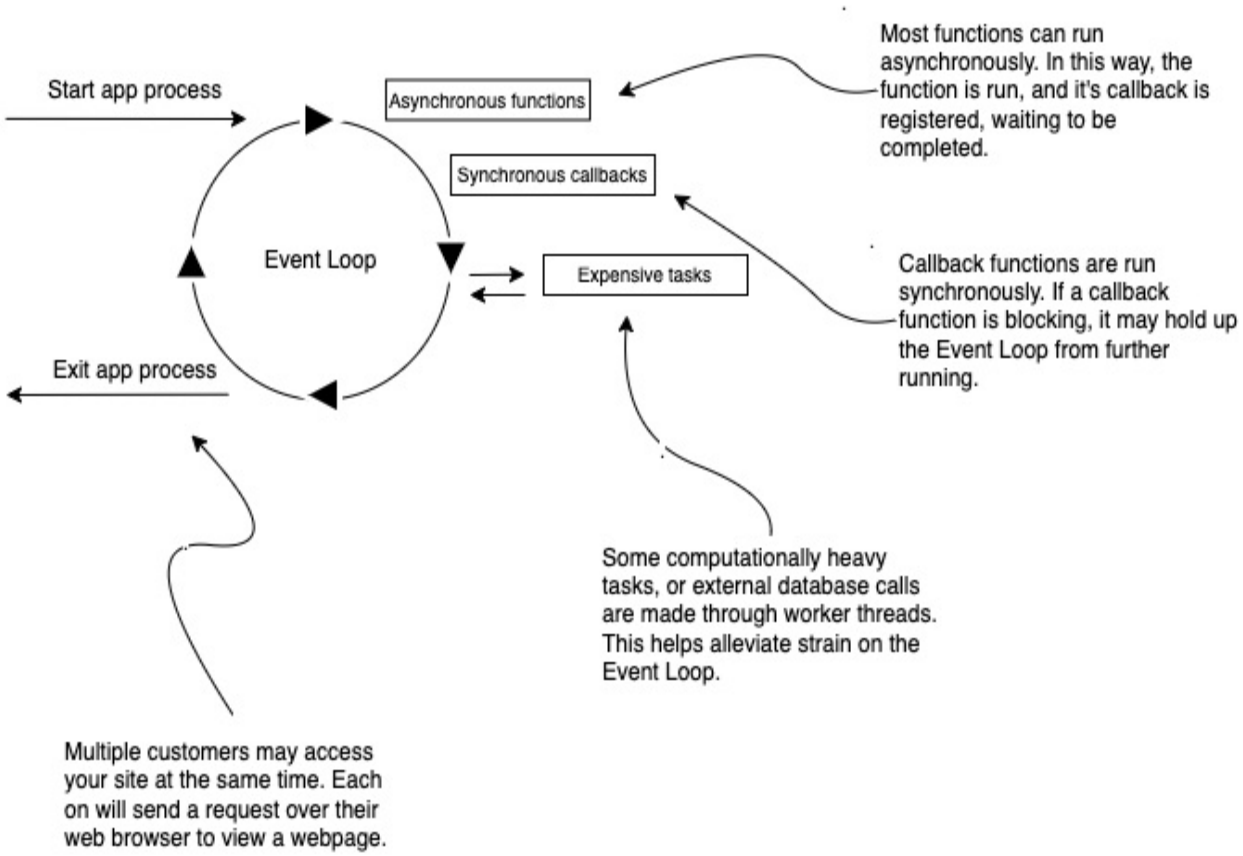
As customers visit the restaurant's site, your Express app will route them to the page they've requested. Because Express uses Node, a single-threaded event loop will process requests to the web server as they are received. Each request is a customer's attempt to visit the restaurant's website via a URL. Node's single thread will receive requests in the order they are received and process them individually. That means only one customer is served at any given moment. Node is fast, though, and as long as you're not running any *expensive* code (like computing the 50th number in the Fibonacci sequence), your application should respond to its users instantaneously, without any request blocking another. You sketch out how Node's event loop might handle requests.

Blocking the Event Loop

The Node Event Loop is at the core of how every Node app operates. Because JavaScript runs off a single thread, it's important that your app allows the single thread to process as many tasks as it can. Figure 2.2 shows some of the ways in which you can block the Event Loop.

While the Event Loop runs on a single thread, Node can spawn a new thread from a "thread pool", or "worker thread". These worker threads are assigned traditionally more expensive tasks like filesystem or database operations (I/O) or encryption tasks. While worker threads are spawned in order to free the single threaded Event Loop, they too may slow down or halt your app from running correctly. Meanwhile, the main thread is largely responsible for registering asynchronous functions and processing their callbacks when ready. If your callbacks contain nested loops, processing large quantities of data, or CPU-intensive code, the Event Loop will not be able to respond to requests from new clients.

Figure 2.2. How to block the Event Loop



When you're building a web server, your Event Loop requests map directly to requests from your client. That means if you are blocking the Event Loop by rendering a web page for one user, then other users will have to wait longer until they get to load their page. As you build your web servers, keep track of which functions run in constant time (a few lines of code execution) versus exponential time (nested loops or computationally heavy tasks).

For more information, visit https://nodejs.org/en/docs/guides/dont-block-the-event-loop/#don-t-block-the-event-loop.

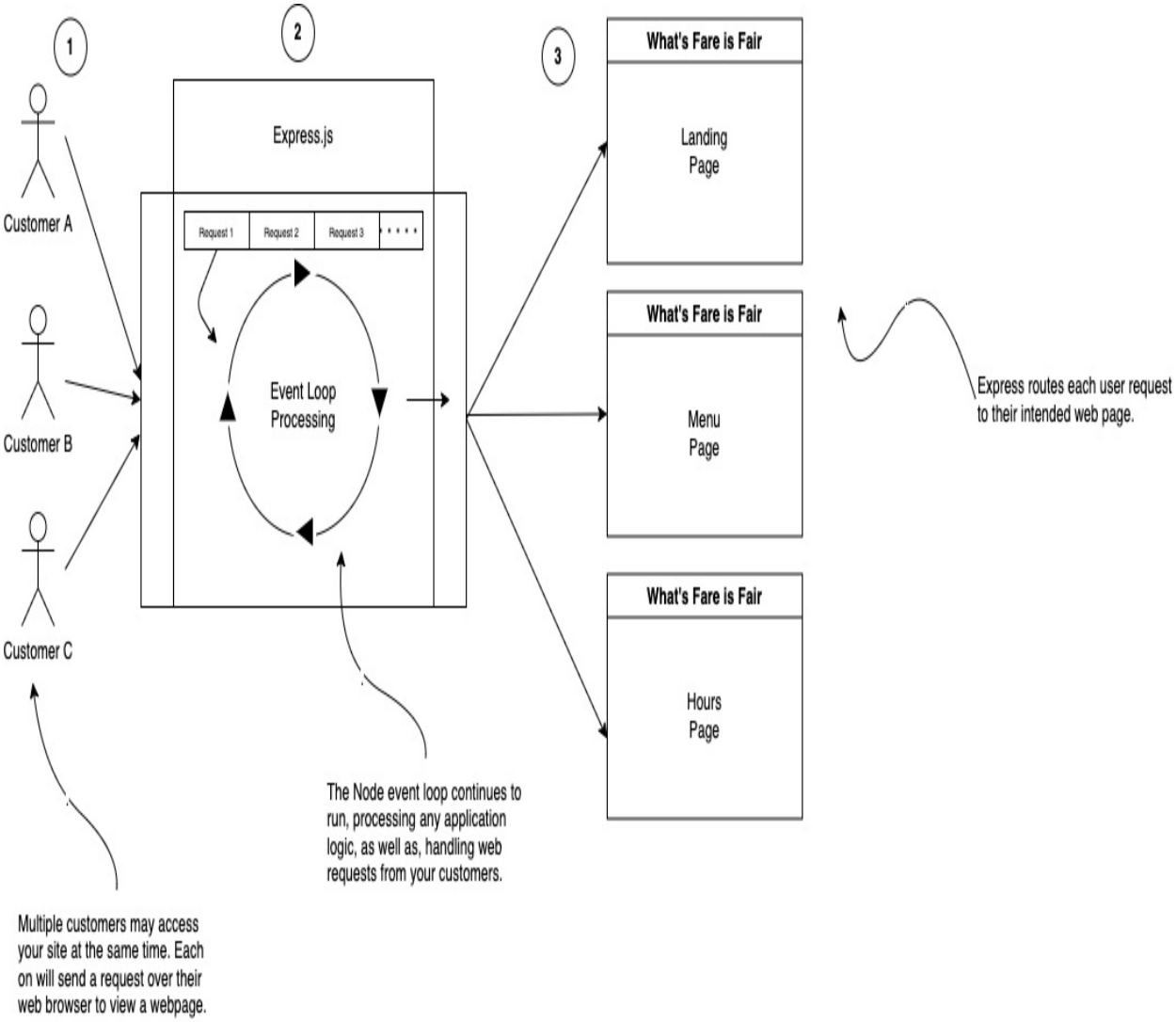
Figure 2.3 demonstrates how web requests may be processed within your completed web application running on Express. Like in figure 2.1, customers visit the restaurant's website. You don't necessarily know how many customers are visiting that URL, or at what rate, but each request enters your application to be processed individually. Node's event loop is able to quickly queue incoming requests and assign a response to each request in the queue, all while continuing to process new incoming requests. As long as there are

queues to temporarily store the order of requests, all the event loop needs to do is handle each request as soon as it is able to. If a customer requests the web page for the restaurant's menu, Express will process that request by routing the customer to a static webpage with the business' hours listed.

 **Note**

For more information about Node's event loop see the official documentation page at https://nodejs.org/en/docs/guides/event-loop-timers-and-nexttick/

Figure 2.3. Development workflow



With the big picture captured, it’s time to get programming.

2.2 Building the application skeleton

While this project can be relatively straightforward, it’s often helpful to break tasks down into more manageable segments. To start, you create an application skeleton that contains the Express framework and some of your application logic.

2.2.1 Working with Express.js

Initialize a new Node application by creating a new folder called `restaurant_web_server`, entering the folder on your command line, and running `npm init` (listing 2.1).

Listing 2.1. Prompt for `npm init`

```
bash
package name: (restaurant_web_server) #1
version: (1.0.0)
description: An web application for a local restaurant.
entry point: (index.js) #2
test command:
git repository:
keywords:
author: Jon Wexler
license: (ISC)
```

After running through the prompt, your `package.json` file will appear in the project’s folder. This file contains both the application’s general configurations and the dependent modules as well.

Next, you run `npm i express` to install the `express` package. You’ll need an internet connection, as running this command will fetch the contents of the `express` package from the npm registry at https://www.npmjs.com and add them to your `node_modules` folder at the root level of your project. Unlike the `fs` and `http` modules that are prepackaged with Node, the `express` module is not offered with your initial installation. Instead, Express.js is bundled into a packaged called `express` that can be downloaded and installed separately

through Node's package management registry tool, npm.

 **Note**

Both Node and Express are projects supported by the OpenJS Foundation. For more information about open-source JavaScript projects from OpenJS visit https://openjsf.org/

After installing `express` you notice that `express` is added to your `package.json` file under a section called `dependencies`, as seen in listing 2.2.

 **Note**

There are a variety of ways to write npm commands, some shorter, and others more explicit in their phrasing. Learn more about npm command line shorthands and flags at https://docs.npmjs.com/cli/v8/using-npm/config

Listing 2.2. Package dependencies in `package.json`

```
json
"dependencies": {
  "express": "^4.17.2" #1
},
```

 **Note**

^ in npm package versioning means your application will ensure that this version, or any compatible versions of the package with minor or patch updates, will be installed to your application. ~ before the version number means only patch updates will be installed, but not minor version changes. For more about `package.json` and how versioning works see https://docs.npmjs.com/cli/v8/configuring-npm/package-json

With `express` installed, you create a file called `index.js` at the root level of your project folder. Next you import express into your application on the first line by adding `import express from 'express'`.

 **Note**

As of Node v17, ES6 imports are not supported by default. You need to add `"type": "module"` to your `package.json` file to use the `import` syntax.

As shown in listing 2.3, you then type `const app = express()` to instantiate a new instance of an Express application and assign it to a variable called `app`. You also assign another variable called `port` a development port number of 3000.

 **Note**

You can use nearly any port number to test your code in development. With ports like 80, 443, and 22 typically reserved for standard unencrypted web pages, SSL, and SSH, respectively, 3000 has become a reliable standard go-to port for software engineers.

Your `app` object has functions you can use to interact with incoming web requests. You add `app.get` on "/", which will listen for HTTP GET requests to your web app's home page. From there you can process the incoming request and reply with a response. To test the application you add `res.send("Welcome to What's Fare is Fair!")` in the `app.get` callback function to reply with plain text to the requesting customer's web browser. Last, you add `app.listen` and pass in your previously defined `port` value and a callback function within which you log a message to your command line console.

 **Note**

Express' `app.get` is named according to the HTTP request type. The most common requests are GET, POST, PUT, and DELETE. To get more familiar with these request methods read more at https://developer.mozilla.org/en-US/docs/Web/HTTP/Methods

Listing 2.3. Setting up your Express app in `index.js`

```
javascript
import express from "express";  #1
const app = express(); #2
const port = 3000; #3

app.get("/", (req, res) => { #4
  res.send("Welcome to What's Fare is Fair!"); #5
});

app.listen(port, () => { #6
  console.log(`Web Server is listening at localhost:${port}`); #7
});
```

Now, you can start your application by running `node index` in your project's command line window. You should then see a logged statement that reads: `Web Server is listening at localhost:3000`. This means you can open your favorite web browser and visit `localhost:3000` to see the text in figure 2.4.

Figure 2.4. Viewing your web server's response in your web browser

With your application's foundation out of the way, it's time to add some flair to What's Fare is Fair's site.

2.3 Adding routes and data

With your application running, you move on to add more routes and context to your restaurant's site. You already added one route: a GET request to the homepage (`/`). Now, you can add two more routes for the menu page and working hours page, as depicted in listing 2.4. Each `app.get` provides a new route at which your web pages are reachable.

Listing 2.4. Adding two more routes in `index.js`

```
javascript
app.get("/menu", (req, res) => { #1
  res.send("TODO: Menu Page");
});

app.get("/hours", (req, res) => { #2
  res.send("TODO: Hours Page");
});
```

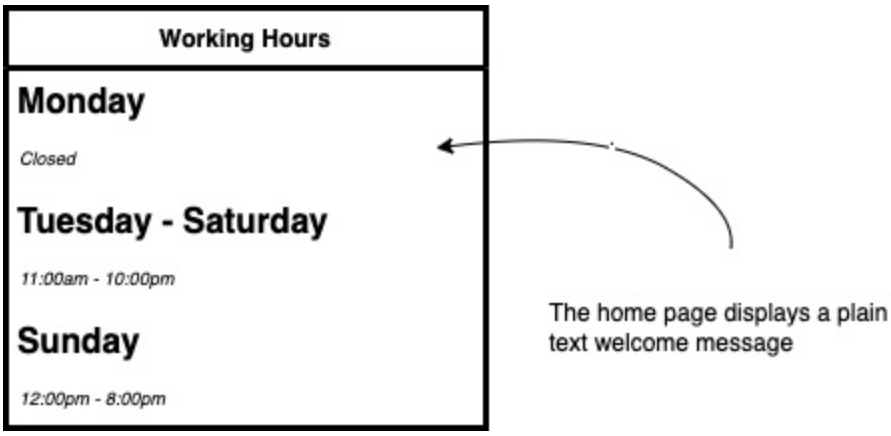
You can stop your Node server by pressing **Ctrl**+**C** in the command line of your running application. With your new changes in place, you can start your application again by running `node index`. Now when you navigate to localhost:3000/menu and localhost:3000/hours you'll see the text change to your `TODO` messages.

This is a good start, but you'll need to fill in some meaningful data here. Your contact at What's Fare is Fair provides you with pictures of their menu (figure 2.5). This image provides insight into the structure of the data in the restaurant's menu. For example, each item has a title, price, and description.

Figure 2.5. Sample menu image

Similarly, the restaurant provides a visual of their working hours, as shown in figure 2.6. Here, you notice that certain days share the same hours of operation, while one day has different hours, and one day the restaurant is closed. Being able to examine this information ahead of building your web pages can help you design your application in an efficient way.

Figure 2.6. Sample working hours image



With these two references for data, you can convert the menu and hours list into JavaScript-friendly data modules. First, you create a folder called `data` in your project directory, where you'll add a `menuItem.js` and a `workingHours.js` file. From these files you use the ES6 `export default` syntax to export all of the files contents for use in other modules, as shown in listing 2.5 and listing 2.6.

Listing 2.5. Menu data in `menuItems.js`

```
js
export default [ #1
  {
    name: "Broccoli Pie",
    description: "A green pie with an earthy crust",
    cost: 12.99,
  },
  {
    name: "Eggplant Smoothie",
    description: "A purple shake with an earthy quake",
    cost: 5.99,
  },
  {
    name: "Watermelon Sushi",
    description: "A red roll with atmospheric sweetness",
    cost: 8.99,
  },
];
```

Transforming the data from a physical menu to this digital JSON-like structure will make it easier for you to systematically display relevant menu items to the restaurant's customers. Because working hours are the same every day except for Monday and Sunday, the normal hours can be listed as `default` values in `workingHours.js`.

Listing 2.6. Hours data in `workingHours.js`

```
js
export default { #1
  default: {
    open: 11,
    closed: 22,
  },
  monday: null,
  sunday: {
    open: 12,
    closed: 20,
  },
};
```

To make use of this data, you import the two relative modules at the top of `index.js` (listing 2.7).

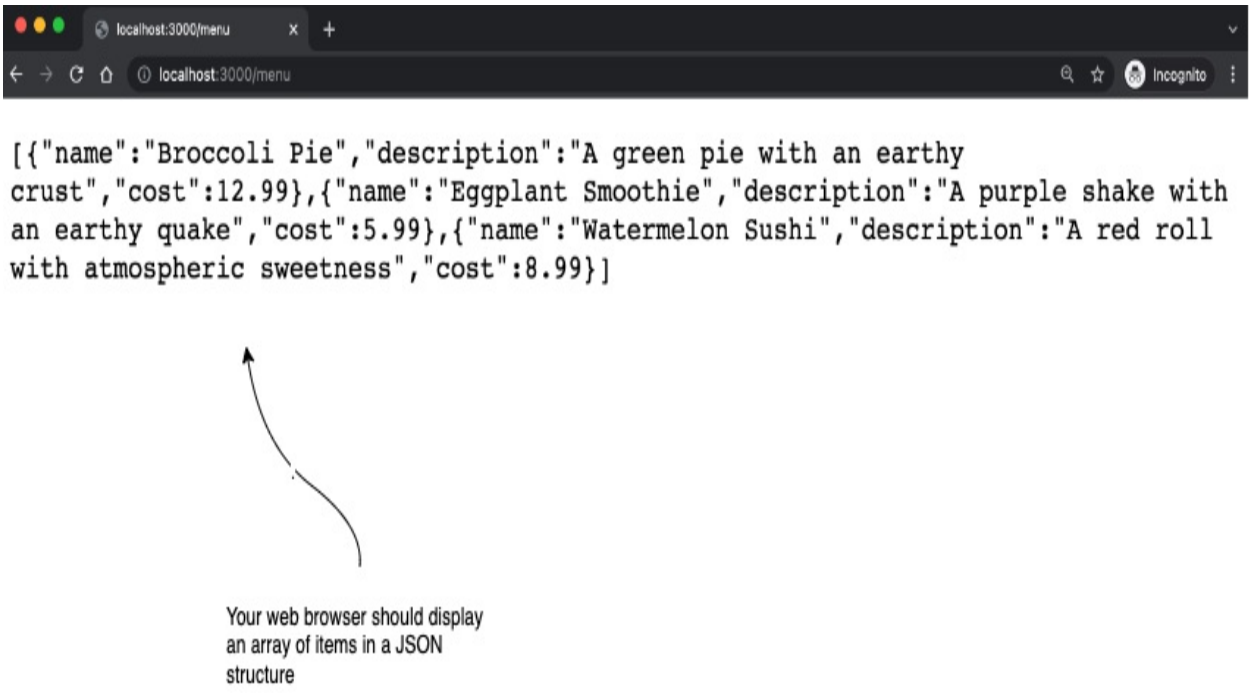
Listing 2.7. Importing data modules into `index.js`

```
js
import workingHours from "./data/workingHours.js"; #1
import menuItems from "./data/menuItems.js";
```

To test that these values are being loaded properly, you replace the `res.send` statements in the `/menu` and `/hours` routes, and replace them with `res.json(menuItems)` and `res.json(workingHours)`, respectfully. Doing so should replace the static text you previously saw when loading your web page and replace it with more meaningful data provided to you by the restaurant. This step in the process can help you validate that the data you're expecting is properly flowing to the webpages you intend them to reach.

Restart your Node server and visit localhost:3000/menu and localhost:3000/hours. Your result for the menu page on your web browser should look like figure 2.7.

Figure 2.7. Sample menu image



With this data displayed on the browser, the next step is to format it to be more visually appealing.

2.4 Building your UI

You could build a user interface using a frontend framework like React.js, Vue.js, or Angular.js. To keep this app simple, you set up a server-side rendered (SSR) template by installing Embedded JavaScript Templates (EJS).

 **Note**

There are a variety of templating engines that work well with Node and Express. Check out https://ejs.co/ and https://pugjs.org/api/getting-started.html to learn more about EJS and Pug.

On the command line, you navigate to your project folder and run `npm install ejs`. This installs the `ejs` package, which facilitates converting HTML content with dynamic data into static HTML pages.

Next, you make use of the `ejs` package by setting it as your view engine in Express (listing 2.8). You can now use the `render` function to display pages written with HTML and EJS.

Listing 2.8. Updating Express routes to render EJS files

```
js
app.set("view engine", "ejs"); #1

app.get("/", (req, res) => {
  res.render("index", { name: "What's Fare is Fair" }); #2
});

app.get("/menu", (req, res) => {
  res.render("menu", { menuItems }); #3
});

app.get('/hours', (req, res) => {
  const days = [ #4
    "monday",
    "tuesday",
    "wednesday",
    "thursday",
    "friday",
```

```
    "saturday",
    "sunday",
  ];
 res.render("hours", { workingHours, days });
});
```

To be able to properly render these pages, you need only to create a folder called `views` at the root level of your project and then add three new files: `index.ejs`, `menu.ejs`, and `hours.ejs`. These three files will be located by the EJS templating engine in express on each request. To complete the process you fill these files with a mix of HTML and EJS. Listing 2.9 shows an example of your landing page, `index.ejs`.

Listing 2.9. Landing page content in `index.ejs`

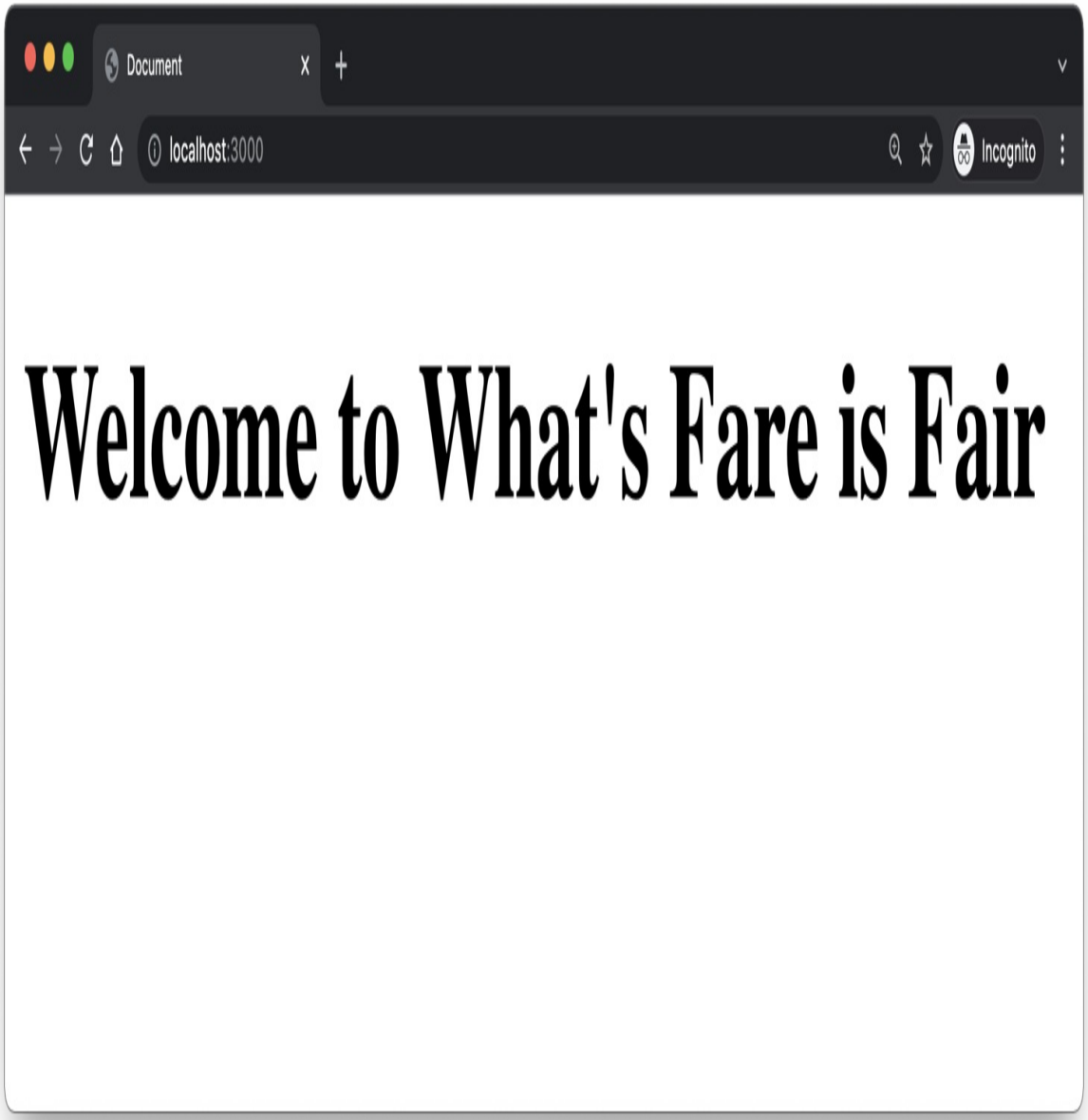
```
html
<!DOCTYPE html> #1
<html lang="en">

<head>
  <meta charset="UTF-8">
  <title>Restaurant</title>
</head>

<body>
  <h1>Welcome to <%= name %></h1> #2
</body>

</html>
```

 **Note**

EJS uses `<%= %>` to display content within in the HTML. In listing 2.10 you use this syntax to display the business name. If you want to run JavaScript on the page without printing anything you leave out the `=`.

When you restart your project and visit http://localhost:3000 your browser should look like figure 2.8.

Figure 2.8. Browser rendering of `index.ejs`



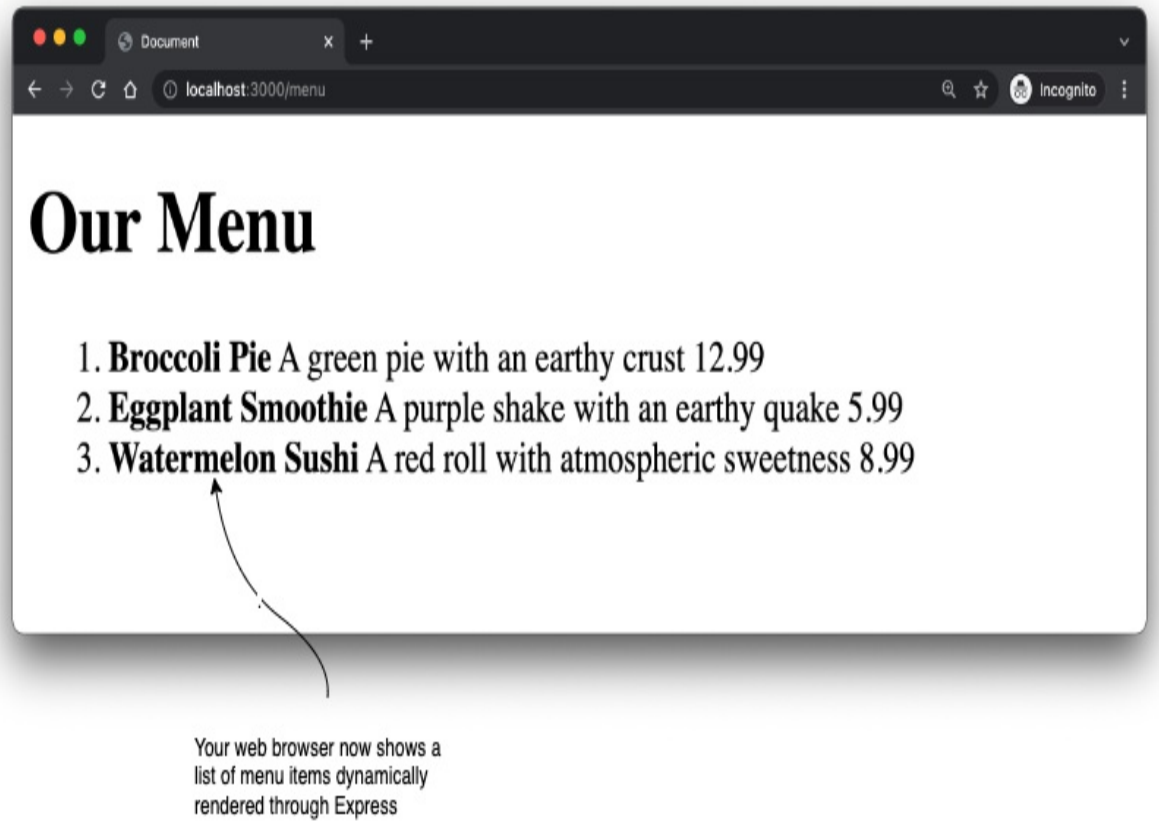
You go on to add the same HTML structure to `menu.ejs` and `hours.ejs`. Modifying only the body tags of each, your `menu.ejs` will contain a for loop iterating over each menu item. From there you display the name, description and cost of each item (listing 2.10).

Listing 2.10. Menu page content in `menu.ejs`

```
<h1>Our Menu</h1>
<ol>
  <% for(let item of menuItems) { %> #1
    <li>
      <strong><%= item.name %></strong> #2
      <%= item.description %> #3
      <%= item.cost %>
    </li>
  <% }%>
</ol>
```

With this code in place, restart your node server and navigate to http://localhost:3000/menu in your web browser to see a page that looks like figure 2.9.

Figure 2.9. Browser rendering of `menu.ejs`



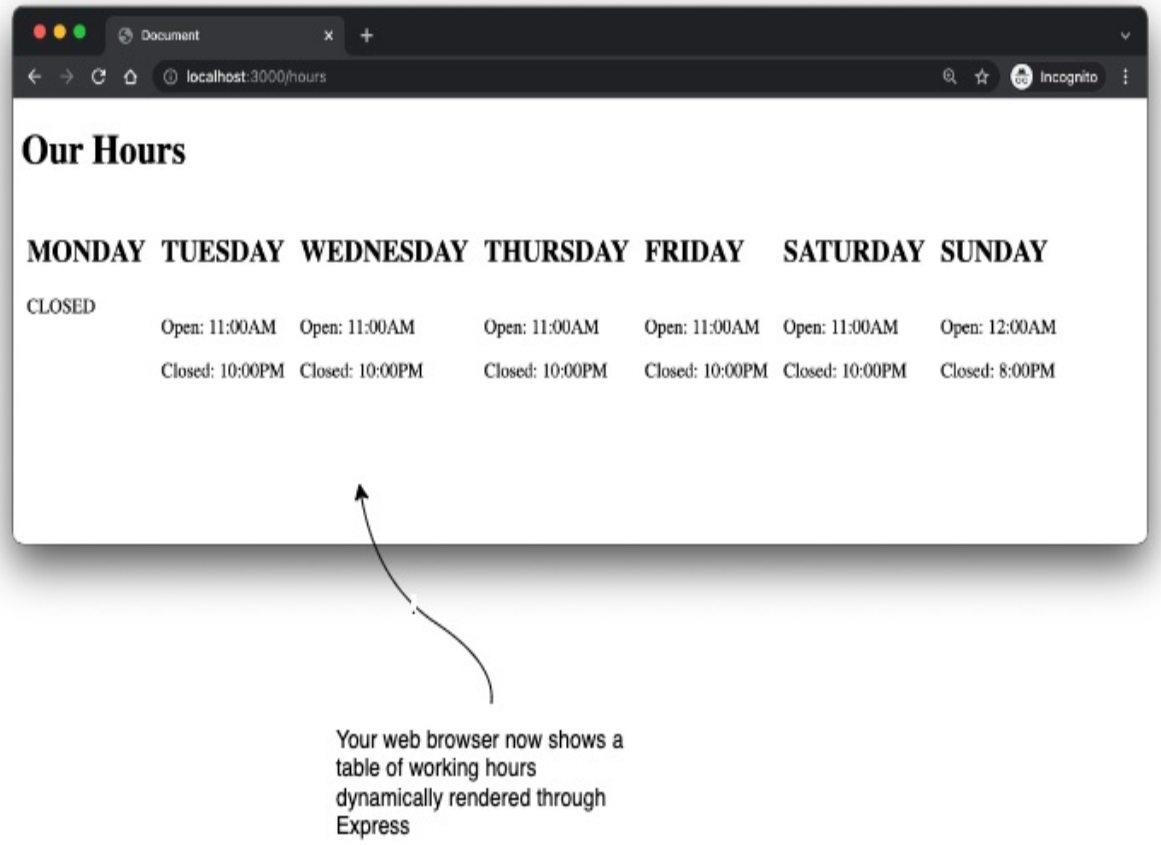
Similarly, in `hours.ejs`, you iterate over each day in your `days` array. From there, you determine whether you have data for that day to display, or simply use the default values previously defined. You use an `if-else` condition here in EJS to separate the UI for days that have hours to display and Monday, when the restaurant is closed (listing 2.11).

Listing 2.11. Hours page content in `hours.ejs`

```
html
<h1>Our Hours</h1> #1
<% for(let day of days) { %> #2
 <% const hoursObj = workingHours[day] || workingHours['default']
 <section style="display: inline-flex; flex-direction: column; pa
    <h2><%= day.toUpperCase() %></h2> #4
    <div>
      <% if (hoursObj.open) {%>  #5
        <p>Open: <%= hoursObj.open %></p> #6
        <p>Closed: <%= hoursObj.closed %></p>
      <% } else {%>
        CLOSED #7
      <% } %>
    </div>
 </section>
<% }%>
```

With this last page complete, you restart your node server and navigate to http://localhost:3000/hours in your web browser to see a page that looks like figure 2.10.

Figure 2.10. Browser rendering of `hours.ejs`



Your web browser now shows a table of working hours dynamically rendered through Express

Admittedly, these pages aren't pretty, even though they show the information required by the restaurant. It's at this stage that you may consider improving the UI with HTML, CSS, and client-side JavaScript. These additions are out of scope for a server-side Node application. Though, if you'd like to learn more about how to create your own stylesheets or import CSS within Express, visit https://expressjs.com/en/starter/static-files.html. Next, you explore how adding some basic styling can impact the look and feel of your web application.

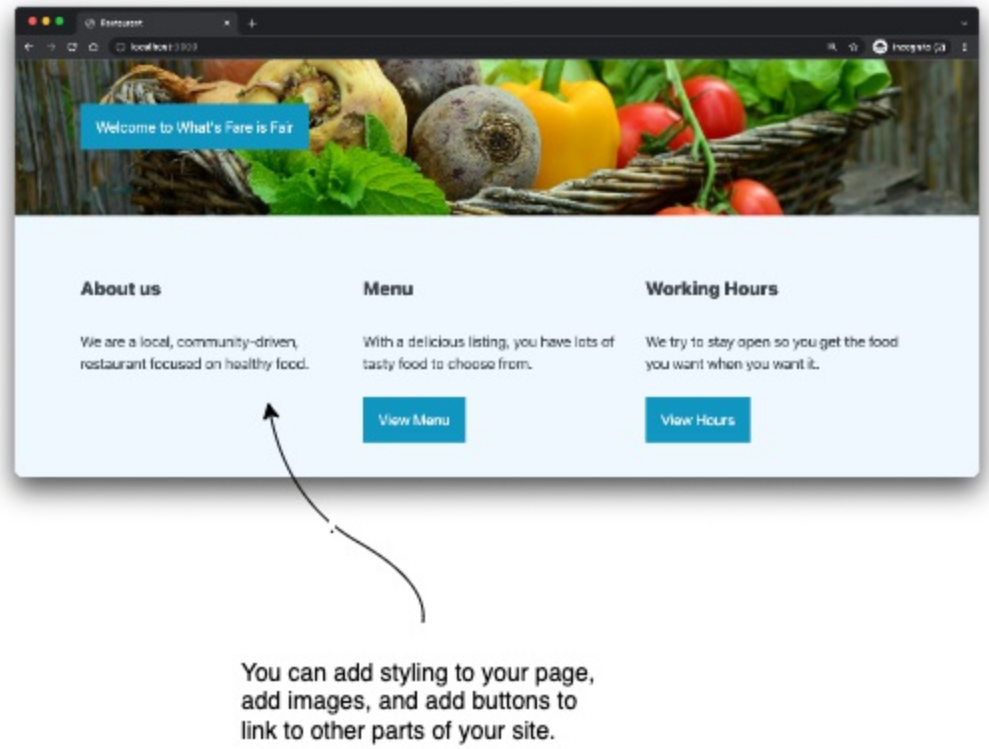
2.5 Sprucing up the UI

Developing a fullstack application generally implies that there's work to be completed on both the backend and frontend. As a Node engineer, you're spending most of your time on the backend. As such, there's no hard requirement to be fluent in HTML or CSS, or their relative libraries and frameworks. It's best to commit time to building out the UI, or to allocate the

work for someone with frontend experience.

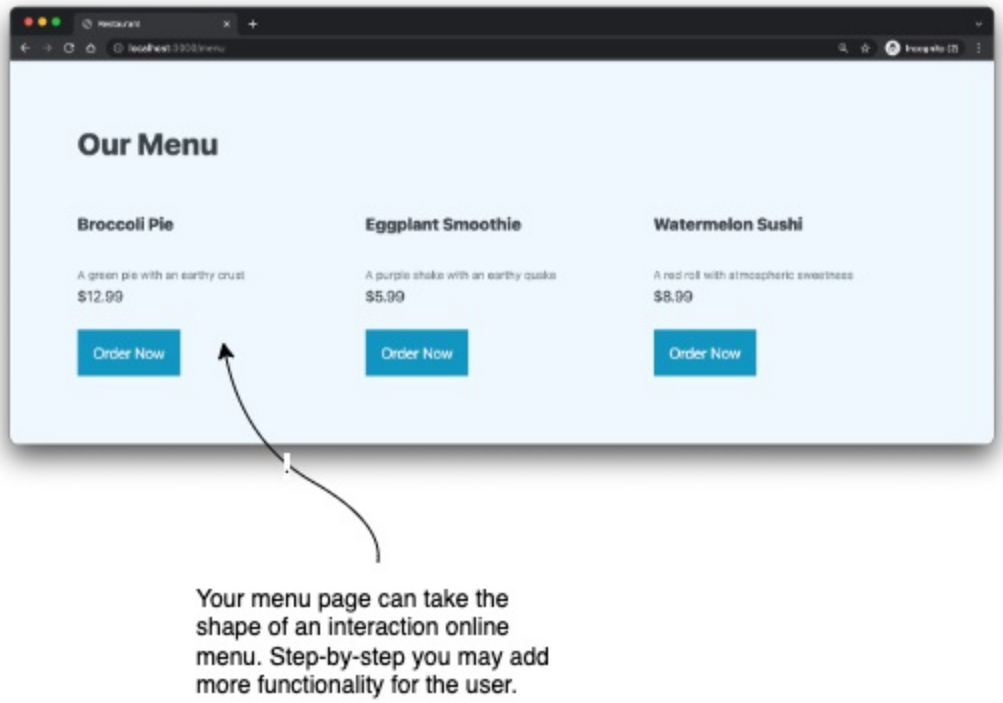
To quickly add a CSS library to Express. Add a `public` folder to the root level of your project. Then add `app.use(express.static("public"))` to your `index.js` file. Then you can add any file with a `.css` extension, images, or other static content to your public folder and access those resources from within your `.ejs` files. Look at figure 2.11, figure 2.12, and figure 2.13 to see how the addition of stylesheets can improve the aesthetics of the pages you built.

Figure 2.11. Browser rendering of styled `index.ejs`



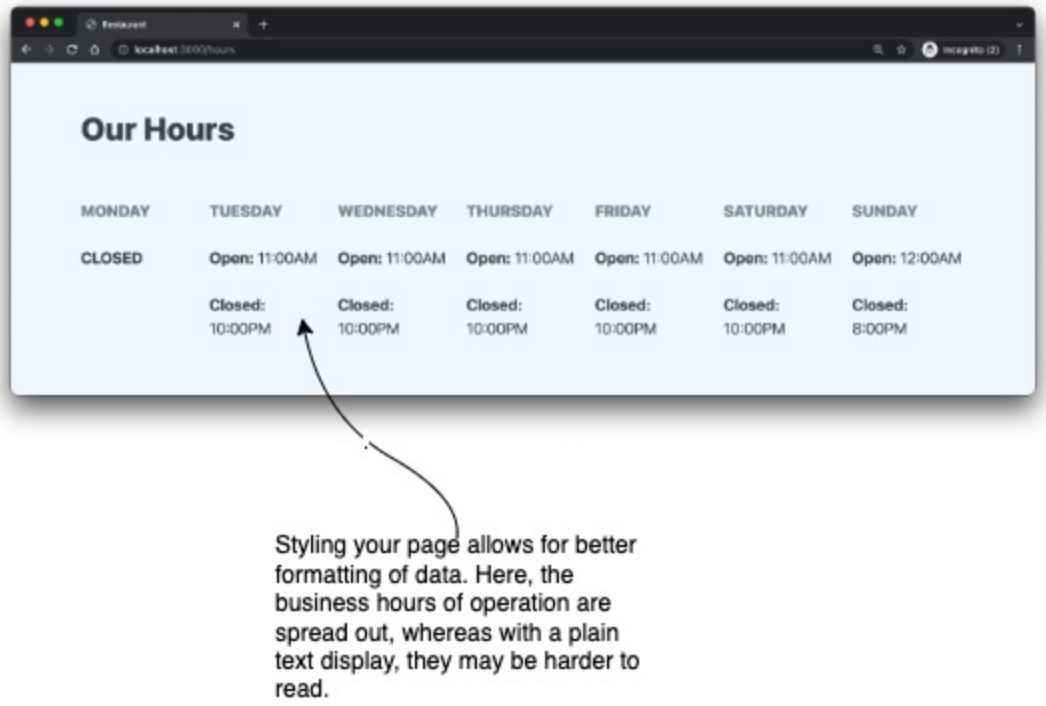
The landing page may have any layout of your choosing. Express has many tools to support custom layouts and other templating engines besides EJS. For more information on templating engines with Express visit https://expressjs.com/en/guide/using-template-engines.html.

Figure 2.12. Browser rendering of styled `menu.ejs`



The menu UI immediately feels more familiar with a visual that indicates the customer may place an online order. Online purchasing can be added to an Node app like this. For now, it’s simply a visual that may encourage the restaurant to invest more time to build a more robust application.

Figure 2.13. Browser rendering of styled `hours.ejs`



The hours of operation may be displayed in a variety of ways. In figure 2.13, a flex-box style is added to ensure all days and times are spaced out without overlap.

If you have the time to dedicate to improving the UI of a Node web app or if you can work alongside a frontend developer, you may find the end result more pleasing to your customer.

2.6 Summary

In this chapter you

- Learned about how the Node event loop handles web requests
- Built a web application with Express
- Served HTML pages using separate data modules

3 Password Manager

This chapter covers

- Data encryption concepts
- Working with Bcrypt
- Saving data to a Mongodb collection

Building applications on the server provides immediate benefits over building on the client. One of those benefits is enhanced control over data security.

The server engineer is typically responsible for protecting data in the database, determining what data the client can view, and who can see it. For this reason, there are a multitude of encryption packages on the npm registry to use with Node to hide sensitive data from all other than the data's original owner.

In this chapter, you will build a password manager using the `bcrypt` encryption package and `mongodb` for persistent storage. You'll start by understanding what happens under the hood with encryption and how you can use this mechanism to build an effective productivity tool. Later, you'll introduce document storage with MongoDB to stay your encrypted data for future access.

Tools & applications used in this chapter

Before you get started, you'll need to install and configure the following tools and applications that are used in this project. Detailed instructions are provided for you in the specified appendix. When you've finished, return here and continue.

- Appendix A.1.2 Installing VS Code
- Appendix A.1.3 Installing Node
- Appendix A.1.4 Installing MongoDB

3.1 Your prompt

Your blockchain startup, Crypto Spies, is growing. With each new service your company uses, you're finding it harder to keep track of your passwords, and you don't yet trust external companies to manage them for you. You decide to set a few hours aside and build your own password manager. This way, you can quickly access passwords from your homemade manager running on your computer.

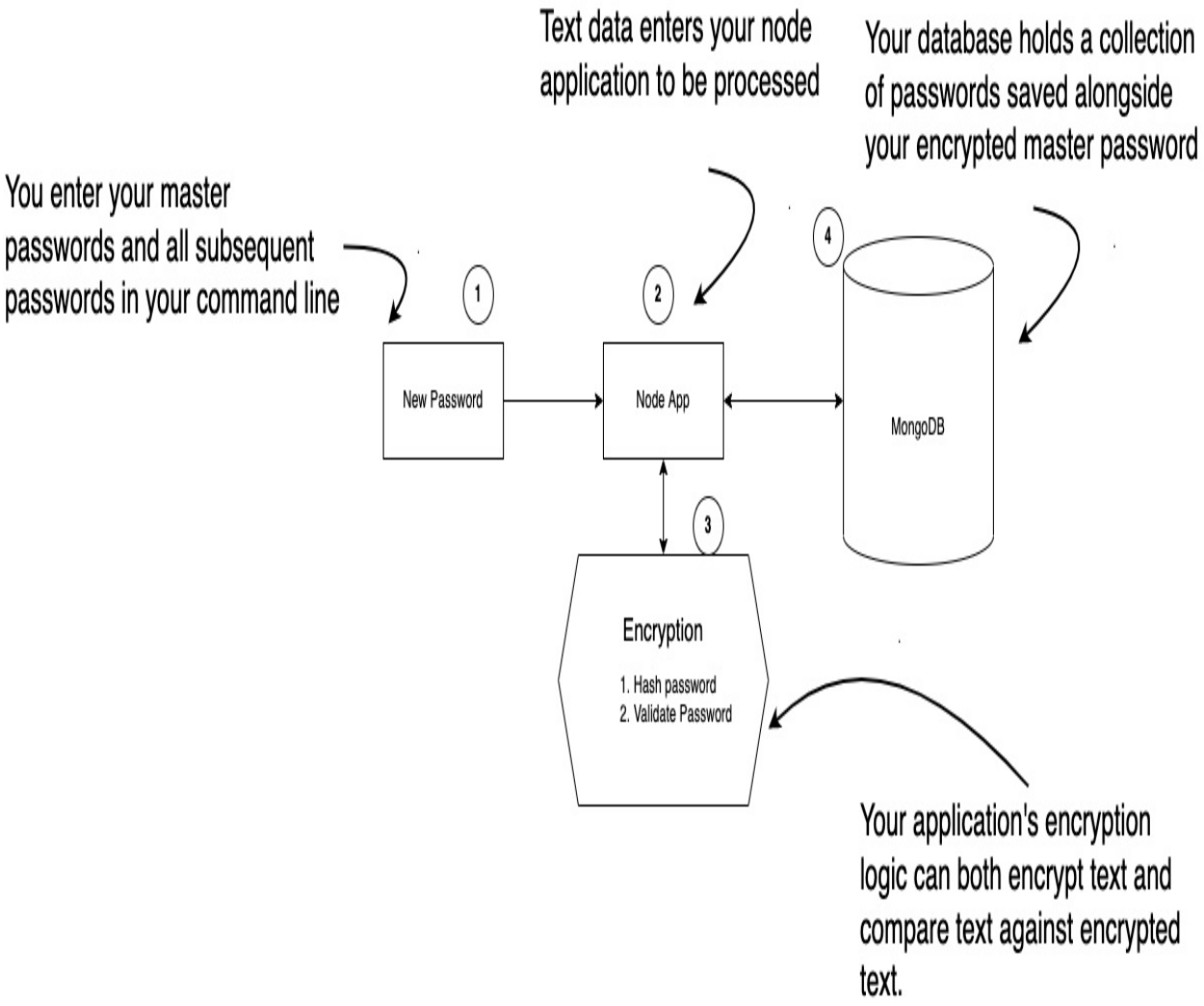
3.1.1 Get planning

You've decided that you want this Node application to run on your own machine, only allow access via an encrypted password, and save your personal passwords in a database. To get this application working in only a short time, you choose an existing encryption library to hash your passwords, and MongoDB to store those passwords. Before you start programming, you diagram the requirements of the project and your result.

Figure 3.1 shows the flow of information for your completed application. Your application will store both an encrypted master password and plain text passwords. The following steps detail how the application should work:

1. To start, a master password will be typed into your command line and sent to your Node application.
2. Your application logic encrypts that password and saves it to your database.
3. The next time you access your application, you type your master password, which will be validated against your encrypted password.
4. If the typed password matches your master password, you may choose to save personal passwords or view a list of saved passwords.

Figure 3.1. Project blueprint for flow of data in password manager app



As you type new passwords to save to your database, the password text enters into your Node app. From there, application logic encrypts your password and saves it to your database. To retrieve that list of passwords, you must re-type a master password that only you know.

Now, it’s time to start coding.

3.2 Building a local command-line manager

It’s best to build a simple version of your application and incrementally add capabilities. For your first version, you plan to build a Node application with

the logic to encrypt your main password and store your list of other passwords in memory (this means the list gets deleted whenever you close your application process).

 **Note**

Before you incorporate a database to save your data long-term, your computer has the ability to temporarily save the data in memory. This means that the data in your application is only preserved for as long as the application is running and the computer is turned on.

Start by creating a new project folder called `password_manager`—This folder may be created where you plan to save Node projects on your computer. Navigate to this folder on your command line and run `npm init` to initialize the Node project. Your initialization prompt should resemble listing 3.1.

Listing 3.1. Prompt for `npm init`

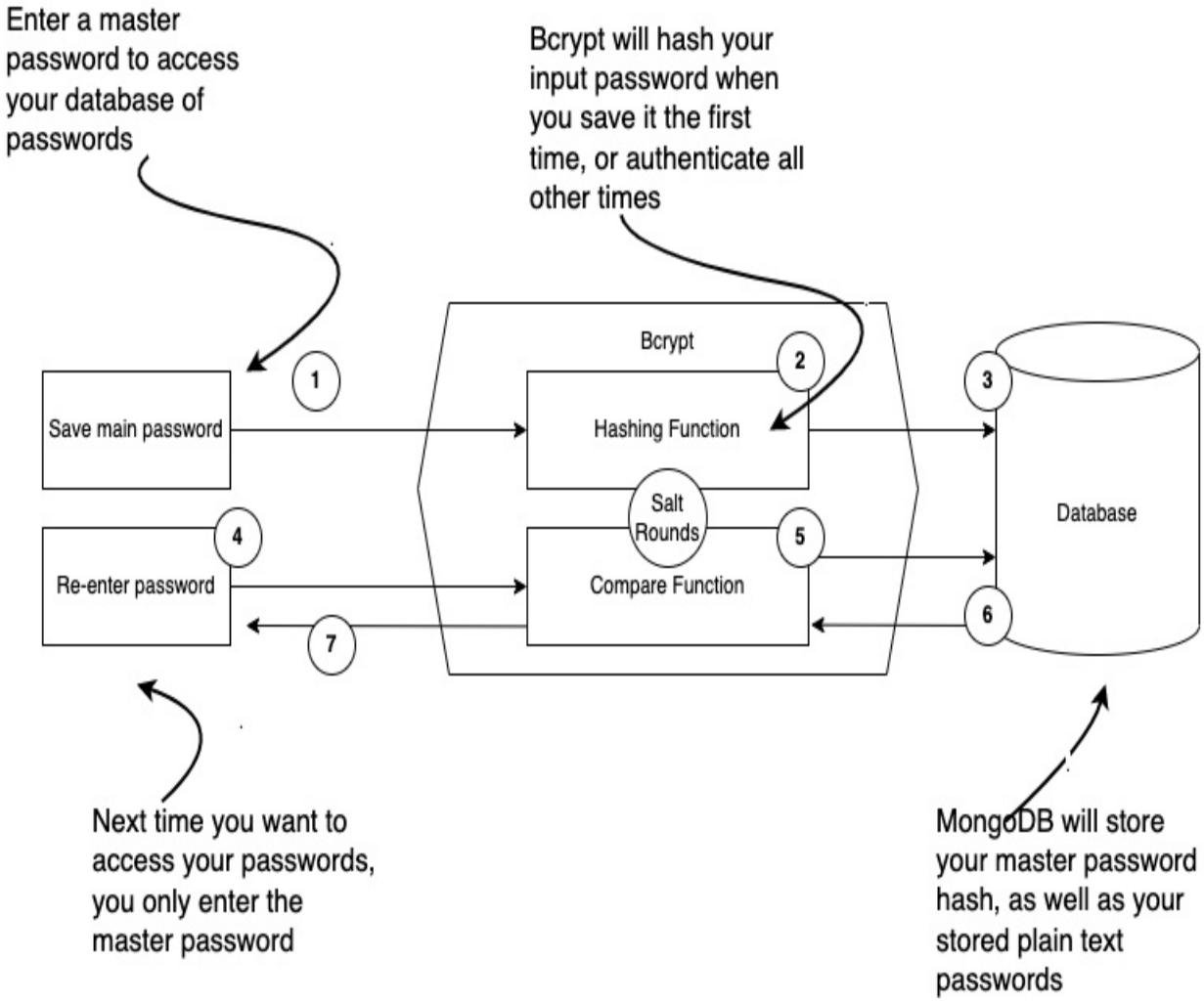
```
bash
package name: (password_manager) #1
version: (1.0.0)
description: A Node app for storing passwords
entry point: (index.js) #2
test command:
git repository:
keywords:
author: Jon Wexler
license: (ISC)
```

Next, run `npm i bcrypt` to install the `bcrypt` package. The `bcrypt.hashSync` function is one of many you can use to hash your password. The process of encrypting your password involves two steps: hashing your password and validating a plain text password against your hashed password. Figure 3.2 shows how the hashing function is a one-way procedure. In this way, it is very difficult to reverse engineer the original password from the hashed value.

First, you type the main password that you'll use to access your other passwords. Bcrypt's `hash` function uses a salt (randomly generated text) to

jumble your password text a number of times equal to your salt rounds value. The resulting hashed password is then stored in your database. Later, when you type your password again to access your manager, your input text is again encrypted and compared to the stored password hash. Bcrypt's `compare` function will use the same salt rounds to evaluate your plain text against the hashed password. If your password matches the hashed password in your database, you are authorized. In this way, Bcrypt does not reverse a hashed password, but instead re-hashes a re-typed password and compares the result with the password hash in the database.

Figure 3.2. Encryption process with Bcrypt



Now, create your `index.js` file at the root level of your project directory. This is where most of your application logic will live. You can test some of

bcrypt's functions by adding the code in listing 3.2 to `index.js`.

Listing 3.2. Testing `bcrypt` in `index.js`

```
js
import bcrypt from "bcrypt"; #1
const password = "test1234"; #2
const hash = bcrypt.hashSync(password, 10); #3
console.log(`My hashed password is: ${hash}`); #4
```

With this code in place you can run `node index` at the root level of your project in your command line window. Your resulting output should look like `My hashed password is: $2b$10$/mLyLstSX54RgR9nQwO.3etHggaCP53.eG1.tsFYmyb8OXVfre84C` (with a different hash value, of course).

 **Note**

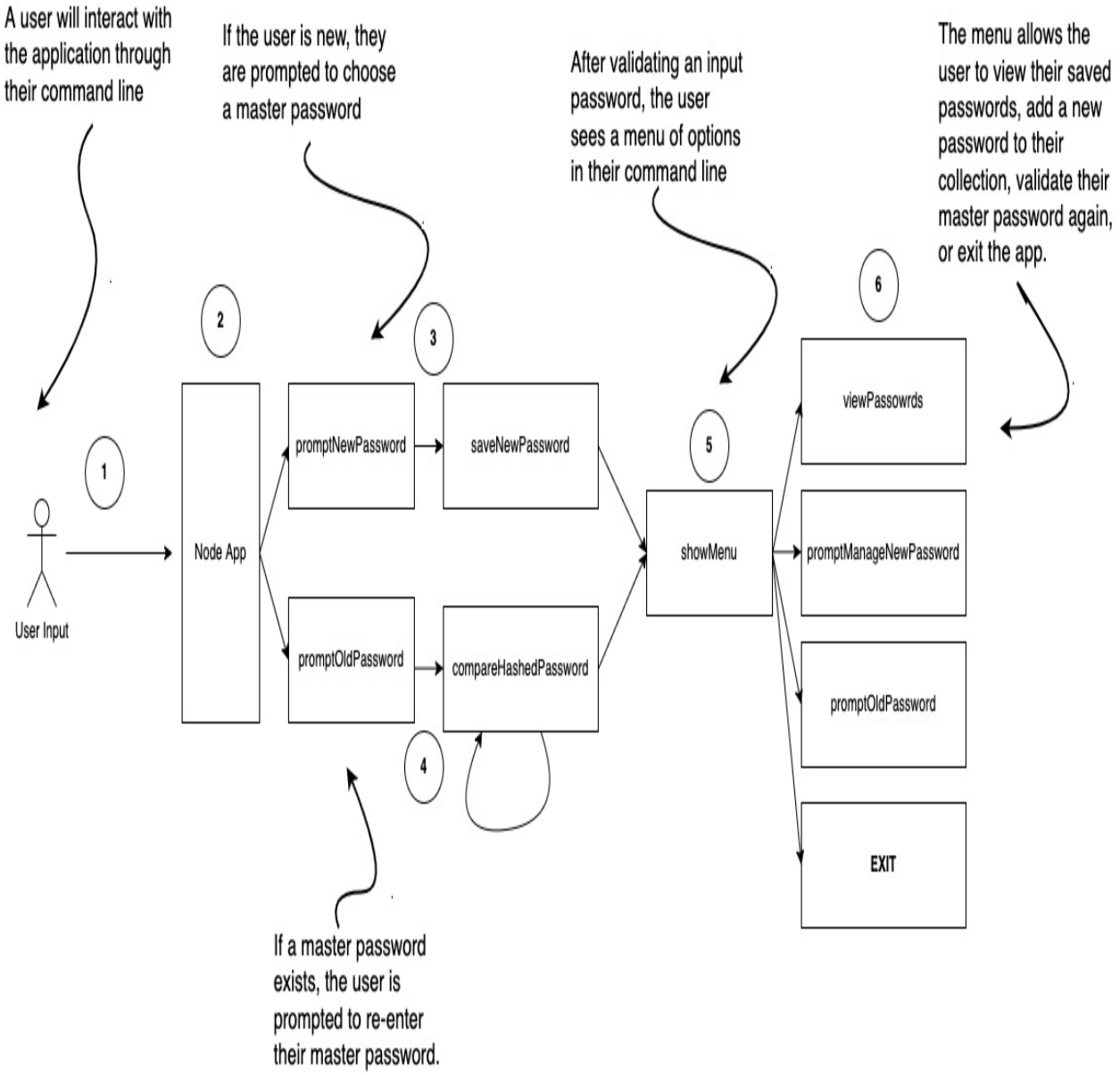
If you don't see a logged statement in your command line window, check to make sure your `index.js` file was saved in the same directory from which you're running the application.

With your test case working, you build out the functions needed to facilitate saving a new encrypted password. Figure 3.3 demonstrates the flow of logic according to the function names you'll use. To start, your application runs a `prompt` function to enable user interaction on the command line. Then, you check whether there is already a master password hash stored. If a master password hash exists, you run `promptOldPassword` to prompt the user to re-type their password. Otherwise, you run `promptNewPassword` to prompt the user to type a new master password for the first time. When the user types their new password, the `saveNewPassword` function will save the resulting hash to the database.

If the user types their existing master password, you compare their input to the stored password hash through `compareHashedPassword`. If their password is validated you display a menu of items to choose from through the `showMenu` function. Within this menu, the user may choose to view their list of passwords (`viewPasswords`), add a new password to your list

(`promptManageNewPassword`), re-verify your hashed password, or exit the app.

Figure 3.3. Code logic-flow diagram



The code for this logic can be written one function at a time. First, install the `prompt-sync` package by running `npm i prompt-sync` at the root level of your project in your command line. Then, add the `bcrypt` and `prompt-sync` imports to your `index.js` file. Also, add a JavaScript object with a `passwords` key mapped to an empty object to represent your database. As you

add new passwords to save, this object will get populated (listing 3.3).

 **Note**

In other chapters the `prompt` package is used, which provides a different syntax for prompting the user in async functions than `prompt-sync`. Here, you are blocking further interactions with your app until prompts are responded to, due to their synchronous nature.

Listing 3.3. Add module imports and mock db to the top of `index.js`

```
js
import bcrypt from "bcrypt"; #1
import promptModule from "prompt-sync";
const prompt = promptModule(); #2
const mockDB = { passwords: {} }; #3
...
```

With your imports in place you can create your first function, `saveNewPassword` which takes a plain text password, `password`, as an argument and makes use of the Bcrypt `hashSync` function to convert the text to a hashed value. That resulting value is then set in the mock database, `mockDB`. You let the user know the password is saved with a log message, and then call the `showMenu` function, which you'll soon write (listing 3.4).

Listing 3.4. Add the `saveNewPassword` function in `index.js`

```
js
...
const saveNewPassword = (password) => {
  const hash = bcrypt.hashSync(password, 10); #1
  mockDB.hash = hash; #2
  console.log("Password has been saved!");
  showMenu(); #3
};
...
```

After the `saveNewPassword` is added, you'll create a function called `compareHashedPassword`, as shown in listing 3.5. In this function, you accept a plain text `password` argument, which is compared to the stored password

hash in `mockDB`. The resulting value is either `true` or `false`;

Listing 3.5. Add the `compareHashedPassword` function in `index.js`

```
js
...
const compareHashedPassword = async (password) => {
  const { hash } = mockDB; #1
  return await bcrypt.compare(password, hash);  #2
};
...
```

The next two functions will prompt the user to type a new password or re-type an old password (listing 3.6). `promptNewPassword` logs a message to the command line console for the user to type their `main` master password. The typed password is subsequently saved in your `saveNewPassword` function. Meanwhile, `promptOldPassword` prompts the user to re-type their old master password. The input text is validated, determining whether the user can view the menu by running `showMenu`, or if the user must re-type their master password again, by re-running `promptOldPassword`.

Listing 3.6. Prompting the user to type passwords in `index.js`

```
js
...
const promptNewPassword = () => {
  const response = prompt("Enter a main password: ");  #1
  saveNewPassword(response);  #2
};

const promptOldPassword = async () => {
  const response = prompt("Enter your password: "); #3
  const result = await compareHashedPassword(response);
  if (result) { #4
    console.log("Password verified.");
    showMenu(); #5
  } else {
    console.log("Password incorrect.");
    promptOldPassword(); #6
  }
};
...
```

So far, you've added functions to facilitate the user's initial interactions and authentication. Listing 3.7 adds code to show a menu of options to choose from once authenticated. `showMenu` logs 4 options for the user to select. The first option runs `viewPasswords` to show them all their saved passwords. Option 2 runs `promptManageNewPassword` to allow the user to save a new password to their database. The third option reruns `promptOldPassword`, allowing the user to re-validate their master password. Finally, the user may quit the application, `exit`, by selecting option 4. If none of the four options are chosen, the user will be notified and prompted to select again.

Listing 3.7. Building the `showMenu` function in `index.js`

```
js
...
const showMenu = () => {
  console.log(`
    1. View passwords
    2. Manage new password
    3. Verify password
    4. Exit`); #1
  const response = prompt(">");

  if (response === "1") viewPasswords(); #2
  else if (response === "2") promptManageNewPassword();
  else if (response === "3") promptOldPassword();
  else if (response === "4") process.exit();
  else { #3
    console.log(`That's an invalid response.`);
    showMenu();
  }
};
...
```

With the menu ready to display, you only need to add the functions to view stored passwords and save new passwords to store. Add the code in listing 3.8 where `viewPasswords` destructs your `passwords` from the `mockDB`. With your passwords as a key/value pair, you log both to your console for each stored password. Then, you show the menu again, which prompts the user to make another selection. `promptManageNewPassword` is the function that prompts the user to type the source for their password; effectively an application or website name for which they are storing their password. Then the user is prompted for a password they want to save. The source and

password pair are saved to your `mockDB` and, again, you run `showMenu` to prompt the menu items.

Listing 3.8. Adding the `viewPasswords` and `promptManageNewPassword` functions in `index.js`

```
js
...
const viewPasswords = () => {
  const { passwords } = mockDB; #1
  Object.entries(passwords).forEach(([key, value], index) => {
    console.log(`${index + 1}. ${key} => ${value}`);
  }); #2
  showMenu(); #3
};

const promptManageNewPassword = () => {
  const source = prompt("Enter name for password: "); #4
  const password = prompt("Enter password to save: ");

  mockDB.passwords[source] = password; #5
  console.log(`Password for ${source} has been saved!`);
  showMenu(); #6
};
...
```

Your application is ready to run. The last piece to add is the code in listing 3.9. Here, `mockDB` is checked for an existing `hash` value. If one does not exist the user is prompted to create one through `promptNewPassword`. Otherwise, the user is prompted to re-type their master password through `promptOldPassword`.

Listing 3.9. Determine the entry point for your application in `index.js`

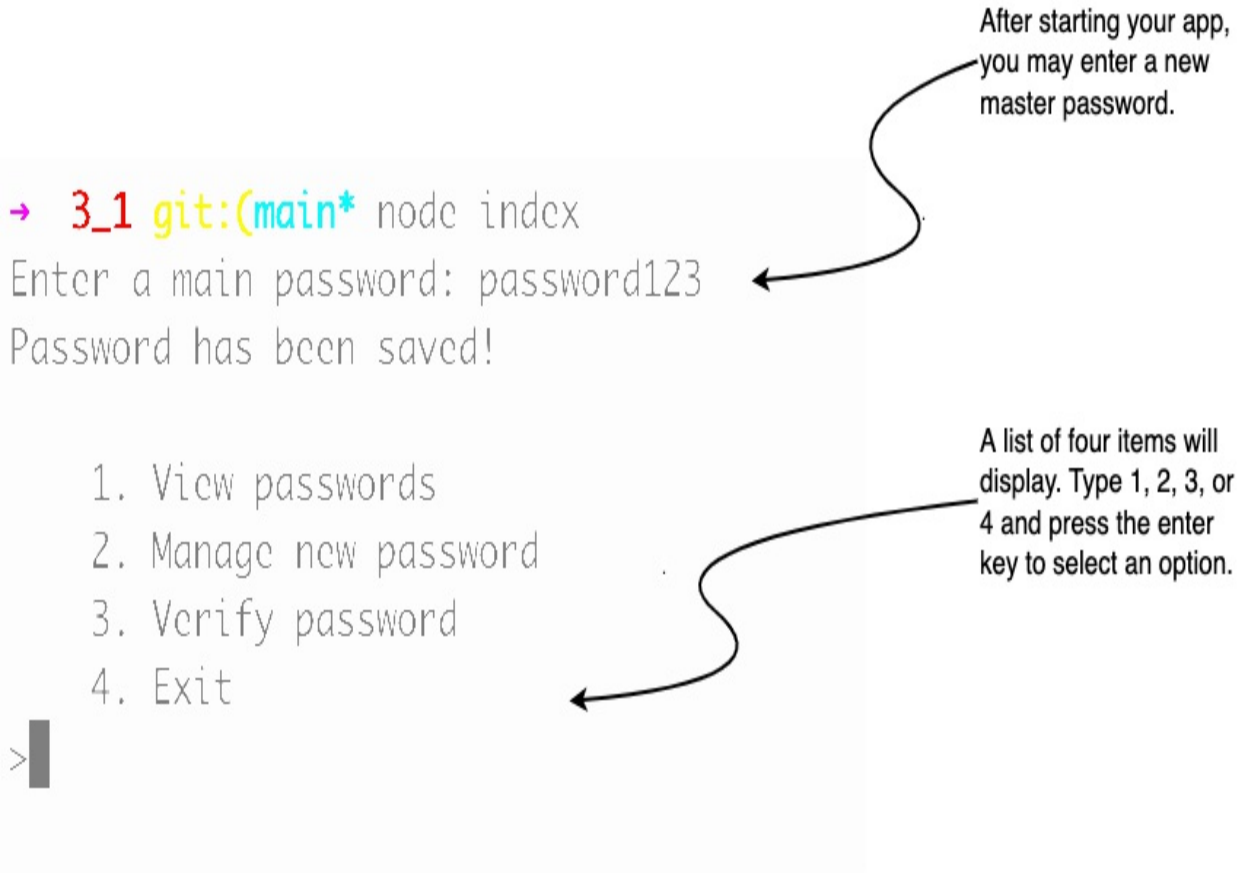
```
js
...
if (!mockDB.hash) promptNewPassword(); #1
else promptOldPassword();
```

With this code in place you have most of the logic you need to run the password manager. The only downside is the local database temporarily stores your managed passwords while the application is running. Because the local database is only an in-memory object, it will get deleted each time you

start your app.

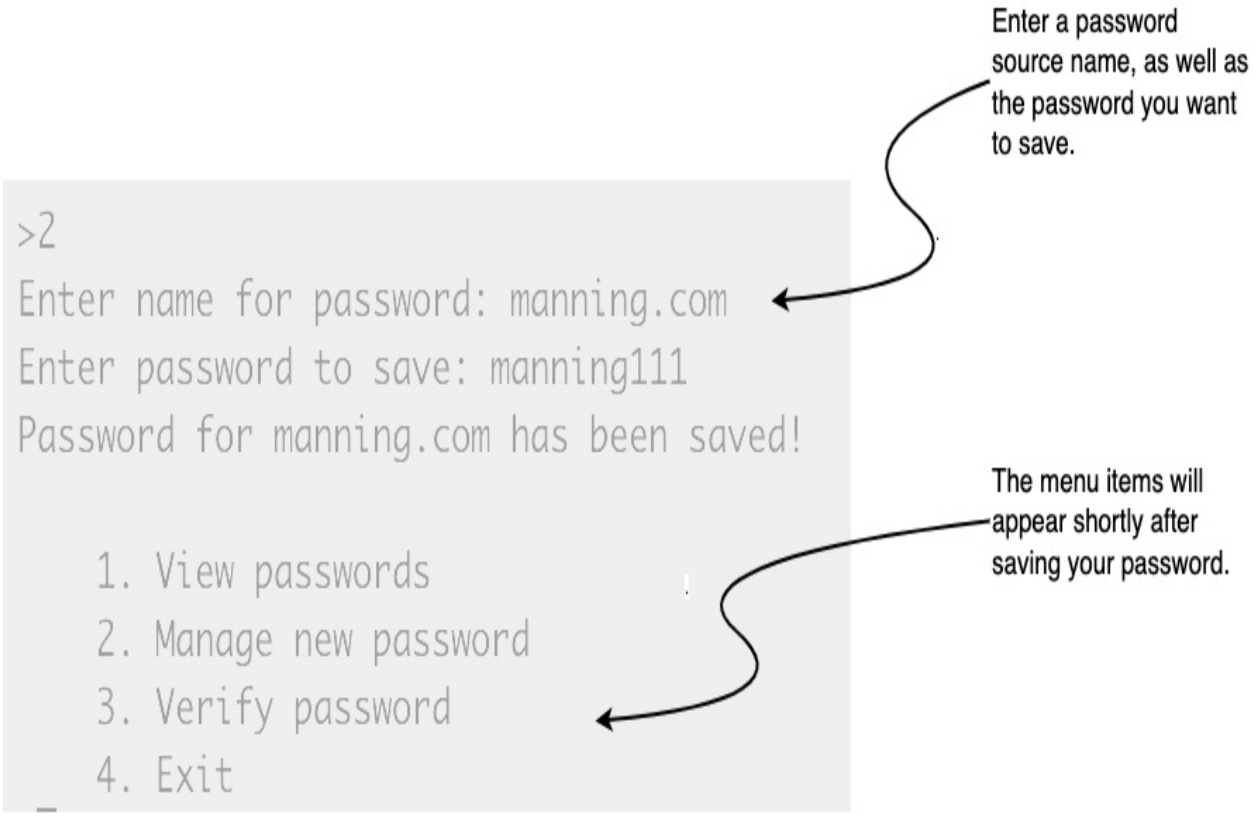
To test this, go to the root level of your project folder in your command line and run `node index`. You should be prompted to type a new password like in figure 3.4. After typing your password, it will be hashed by `bcrypt` and you'll see a menu of items to choose from.

Figure 3.4. Typing your main password to access your application menu



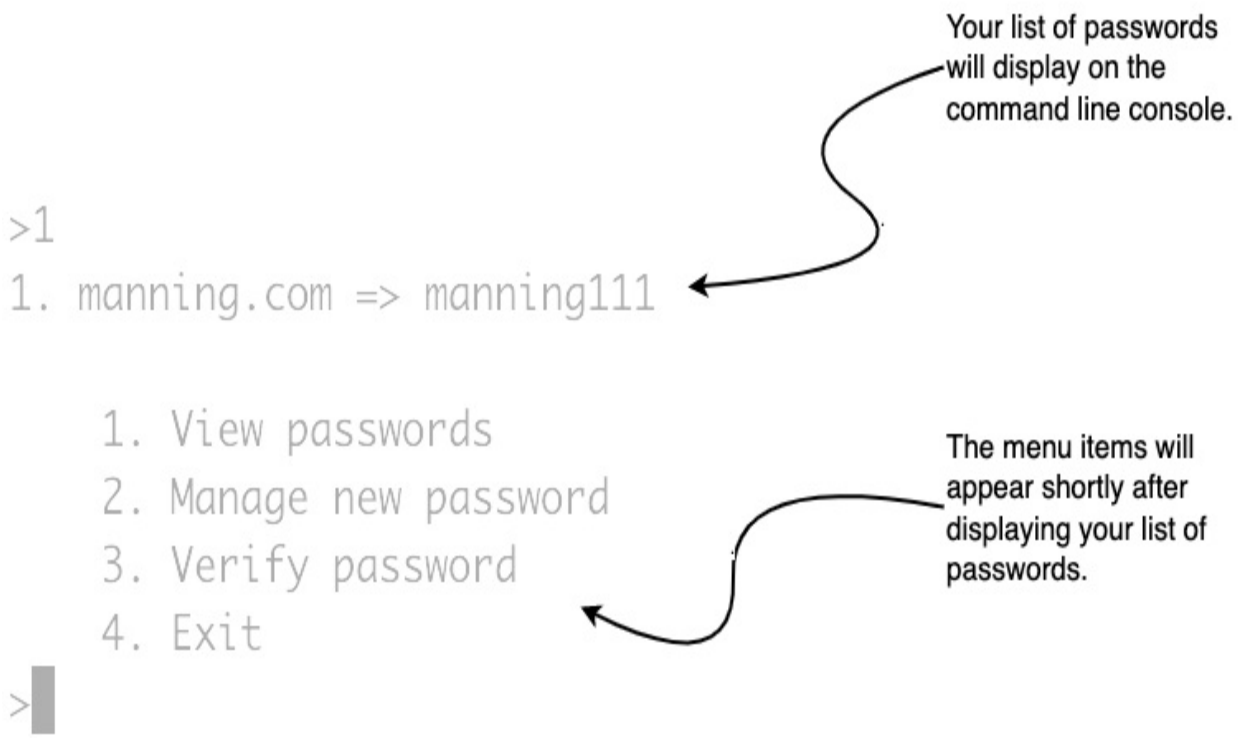
From here you can select `2` and press **Enter** to add a new password to manage. Try typing a source like `manning.com` and a password as seen in figure 3.5.

Figure 3.5. Saving a new password to manage



After pressing **Enter**, this password is saved to your in-memory object. You should then see the original menu items appear. Select `1` and press **Enter** to see the list of passwords now containing your `manning.com` password (figure 3.6).

Figure 3.6. Selecting to view all managed passwords



You can also test your main password (the first password you typed when you started the app) by selecting `3` and pressing **Enter**. If you type in the wrong original password you'll see a log statement letting you know the password is incorrect. Otherwise you'll be prompted that the password matches the hashed password and returned to the menu.

Now you can safely exit the application by typing `4` and pressing **Enter**. This step safely kills the Node process and exits your command line app. The next step is to add a persistent database so you don't have your passwords deleted every time you run your app.

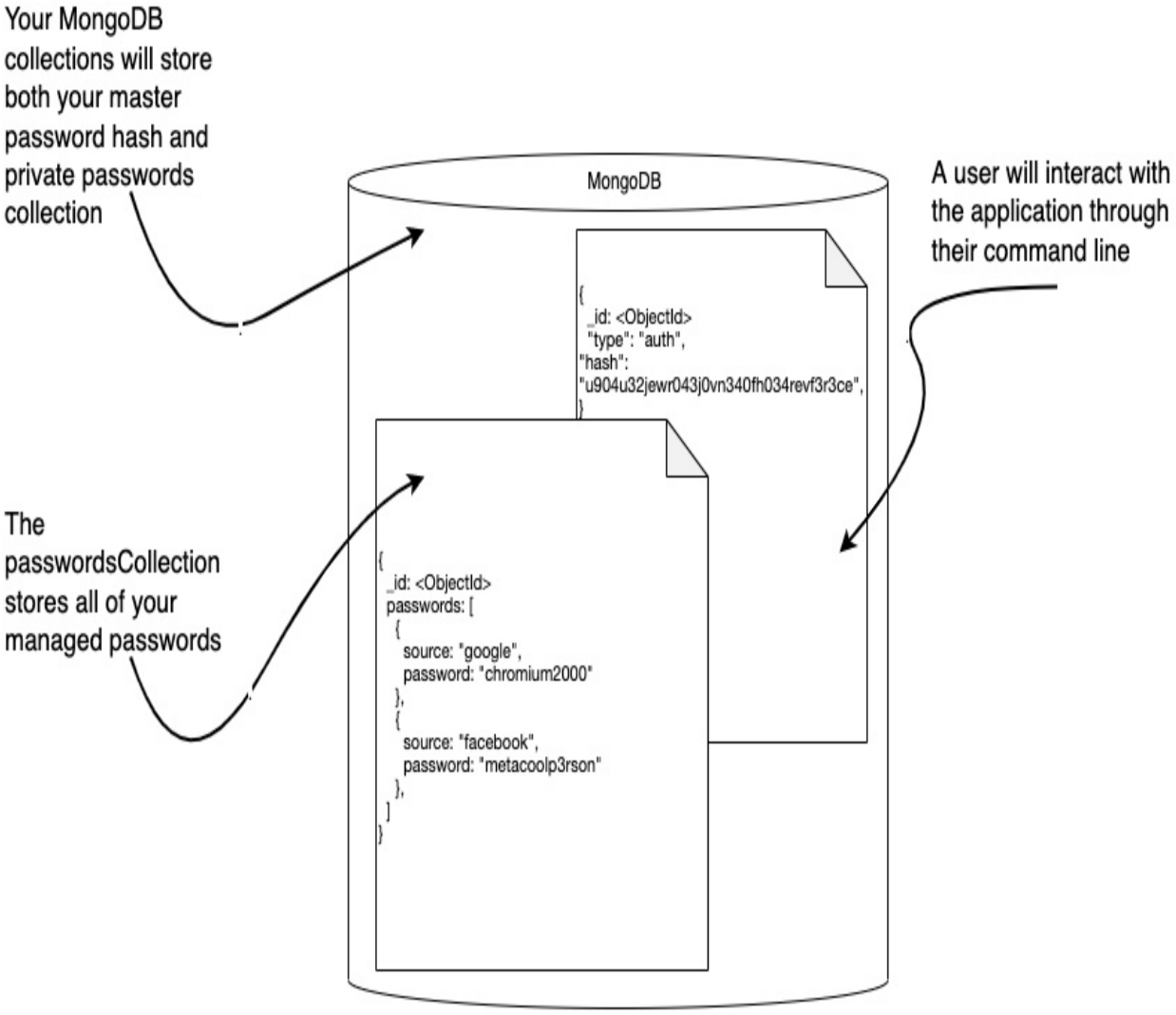
3.3 Saving passwords with MongoDB

After completing most of the logic of your Node app, the next step is to introduce a way to save application data when the application is no longer running. MongoDB is one database you can use to store this information. MongoDB is a document-oriented database manager, meaning it manages NoSQL non-relational databases. Because your application is intended to

store your own collection of passwords, using MongoDB collections is appropriate for this project.

In figure 3.7 you see a diagram with an example of how your data could be stored. This structure is similar to JavaScript Object Notation (JSON), making it easier to continue to work with JavaScript on the backend. Notice in this figure you store the `password_hash` as an encrypted value for your main password. Then you have a list of `passwords` that map a `source` name to a plain text `password`. Additionally, MongoDB will assign an ObjectId to new data items within a collection.

Figure 3.7. Diagram of saving passwords to MongoDB



Note

For this section, you'll need to ensure MongoDB is properly installed. Visit appendix A.1.4 for installation steps.

Go to your project's root level at the command prompt and run `npm i mongodb`.

Once installed, the `mongodb` package will provide your Node application the tools it needs to connect to your database and start adding data. For this reason, you no longer need your temporary in-memory storage, `mockDB`, from section 3.2. Instead, you use the `MongoClient` to set up a new connection to you local MongoDB server. Your development server should be running at `mongodb://localhost:27017` on your computer. Last, you set up a database name, `passwordManager`, to connect to.

In `index.js`, add the code in listing 3.10.

Listing 3.10. import `mongodb`

```
js
import { MongoClient } from "mongodb"; #1
let hasPasswords = false; #2
const client = new MongoClient("mongodb://localhost:27017"); #3
const dbName = "passwordManager"; #4
```

Next, create an `async` function to establish your app's connection to the database. In listing 3.11 you'll find the code you need to add to connect to the database. The `client.connect` function will attempt to initiate a connection with your local MongoDB server. Then, `client.db(dbName)` will connect to a database by the name name assigned to `dbName`. In your case, you'll have two MongoDB collections in the database: `authCollection` to handle storing your password hash, and `passwordsCollection` to store the list of passwords. Once connected, this function will search for an existing password hash by running `authCollection.findOne({ "type": "auth"})`. `hasPasswords = !!hashedPassword` converts the result from your search into a boolean value. In the end, you return `passwordsCollection` and `authCollection` in an array.

 **Note**

The reason you need an async function is because connecting to the MongoDB database is considered a blocking I/O operation. Typically when you connect to a database and run a command against it you don't know how long it may take to complete. Async-await allows us to effectively run this code synchronously until you get a response from the database.

Listing 3.11. `main` function to initialize the database

```
js
...
const main = async () => { #1
  await client.connect(); #2
  console.log("Connected successfully to server");
  const db = client.db(dbName); #3
  const authCollection = db.collection("auth"); #4
  const passwordsCollection = db.collection("passwords");
  const hashedPassword = await authCollection.findOne({ type: "au
  hashedPasswords = !!hashedPassword; #6
  return [passwordsCollection, authCollection]; #7
};
```

At the bottom of `index.js` add the code in listing 3.12 to call the `main` function and begin processing your app.

Listing 3.12. Call `main` to set up MongoDB collections

```
js
...
const [passwordsCollection, authCollection] = await main(); #1
if (!hasPasswords) promptNewPassword(); #2
else promptOldPassword();
```

Now you can restart your Node application by exiting any running application and typing `node index`. If your application successfully connected to the database you should see `"Connected successfully to server"` logged to your command line.

 **Note**

After saving passwords, if you want to delete the database of passwords and start from scratch, you can always add `await passwordsCollection.deleteMany({})` or `await authCollection.deleteMany({})` to delete your passwords or main hashed password, respectively.

With your database connected, you need to modify some of your application logic to handle reading and writing to your MongoDB collections. Change `saveNewPassword` to become an `async` function. Within that function change `mockDB.hash = hash` to `await authCollection.insertOne({ "type": "auth", hash })`. This will save the hashed password `hash` to the `authCollection` in your database.

Next, change the `compareHashedPassword` function to `async` and modify the first line to `const { hash } = await authCollection.findOne({ "type": "auth"})`. This line will search your `authCollection` for a hashed password and send that to the `bcrypt` compare function.

The last three functions to change are in listing 3.13. Here, `viewPasswords` is modified to pull all passwords (by source and password value) from your `passwordsCollection`. `showMenu` will remain the same, but like the other functions will become `async`. In this function you add `await` before each function call, as they are now performing I/O operations. Last, `promptManageNewPassword` uses the `findOneAndUpdate` MongoDB function to add a new password entry if it doesn't exist, or override and update a password entry if an old value exists. The options `returnNewDocument` and `upsert` tell the function to override the changed value and return a copy of the modified value when the save operation is complete.

Listing 3.13. Adding database calls to functions in `index.js`

```
js
const viewPasswords = async () => {
  const passwords = await passwordsCollection.find({}).toArray();
  Object.entries(passwords).forEach(([key, { source, password }],
    console.log(`${index + 1}. ${source} => ${password}`);
  });
  showMenu();
};
```

```
const showMenu = async () => {
  console.log(`
     1. View passwords
     2. Manage new password
     3. Verify password
     4. Exit`);
  const response = prompt(">");

  if (response === "1") await viewPasswords(); #3
  else if (response === "2") await promptManageNewPassword();
  else if (response === "3") await promptOldPassword();
  else if (response === "4") process.exit();
  else {
    console.log(`That's an invalid response.`);
    showMenu();
  }
};

const promptManageNewPassword = async () => {
  const source = prompt("Enter name for password: ");
  const password = prompt("Enter password to save: ");
  await passwordsCollection.findOneAndUpdate(
    { source },
    { $set: { password } },
    {
      returnNewDocument: true,
      upsert: true,
    }
  ); #4
  console.log(`Password for ${source} has been saved!`);
  showMenu();
};
```

With this code in place, you have a fully functional database to support your password manager application. Quit any previously running Node application and restart the application by running `node index`. Nothing should change about the prompts you see in the command line. Only this time the values you type will persist even when you quit the application.

With this application complete, you can always run the application locally and add or retrieve passwords secured behind your hashed main password. Some next steps you could take would be to add a client with a UI to help with visualizing your password data or setting up your database in the cloud, so that your passwords persistent from computer to computer.

3.4 Summary

In this chapter you

- Built your own password manager application. * Designed encryption logic utilizing the `bcrypt` package
- Set up a MongoDB database collection for passwords
- Configured Node to use the `mongodb` package with user input

4 RSS Feed

Dashboards of data are the way of the future. If there are metrics worth tracking, there's a dashboard to visualize it. One of the original dashboards designed to save everyone time is the really simple syndication (RSS) feed. RSS is not a dashboard by nature, but an organization of content, usually XML files, to distribute snippets of text, headlines, and latest news from various sources. Instead of you having to manually visit multiple websites and read their articles, an RSS feed can collect the top stories and present them to you in your custom RSS reader. For this to work, you need a server to act as the RSS aggregator, which tracks what content you've seen so you can always get new and up-to-date results. Although RSS feeds are dwindling in popularity, their innate architecture is still useful to understand and apply in more creative and useful ways.

In this chapter, you'll build your own RSS feed reader to access and process XML. Then you'll create your own RSS aggregator, which will read from multiple feeds and provide up-to-date and relevant content on demand. By the end, you'll have a fully functioning RSS feed aggregator that can list data on your command-line or web client.

Tools & applications used in this chapter

Before you get started, you'll need to install and configure the following tools and applications that are used in this project. Detailed instructions are provided for you in the specified appendix. When you've finished, return here and continue.

- Appendix A.1.2 Installing VS Code
- Appendix A.1.3 Installing Node

4.1 Your prompt

Your coworkers at the office love to talk with one another about food, but with so many articles to read these days it has been difficult to keep up with

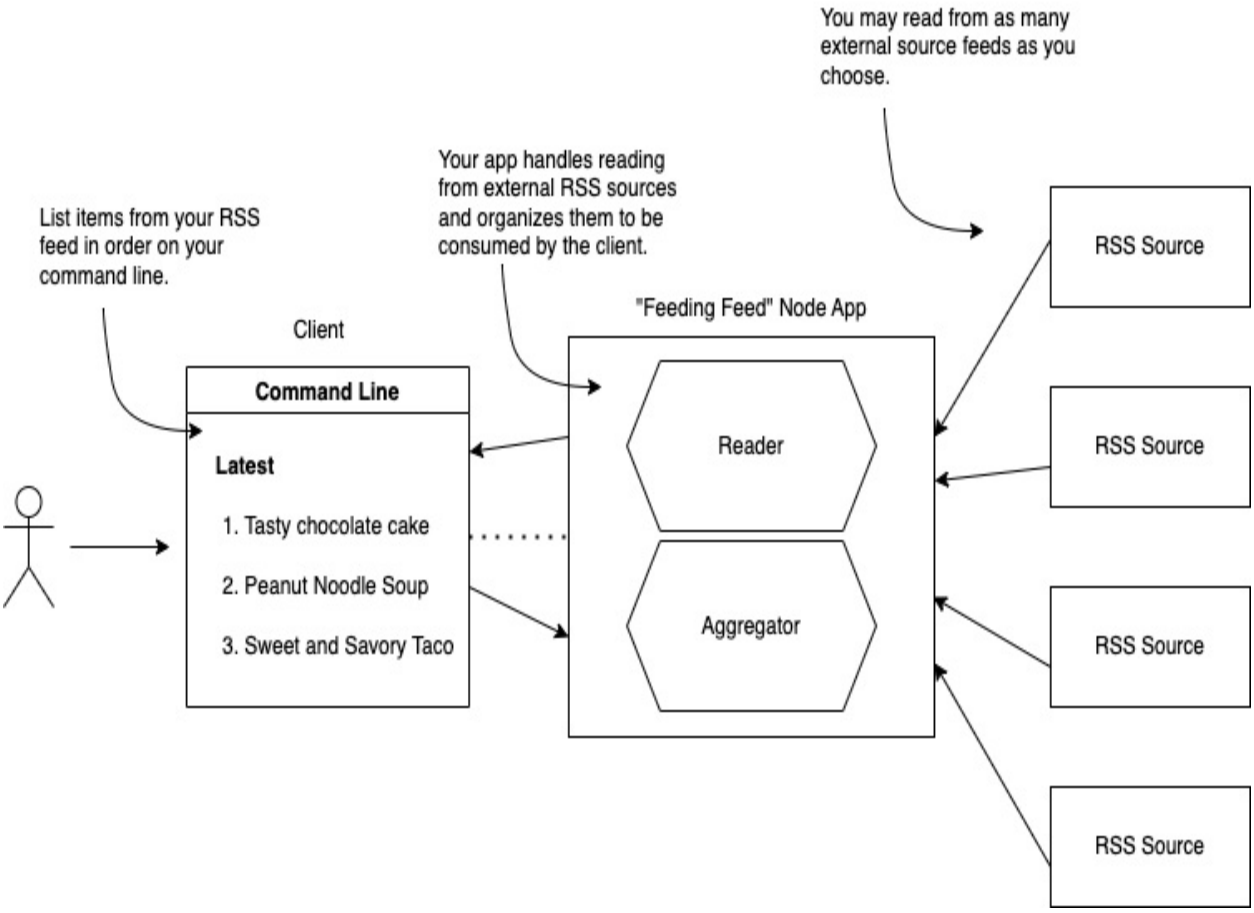
all the food trends. Text chat has been a mildly effective way to share interesting links, but you have a better idea. You'll build an RSS aggregator for your company to collect new food-related content. If there's an article on upcoming plant-based steak, both you and your coworkers can hear about it. For that reason, you decide to build an RSS "Feeding" feed.

4.1.1 Get planning

The goal of this project is to build an app that aggregates new and relevant content for a large group of people. While popular RSS apps exist, you want to use that same architecture to design your own Node app. You begin by experimenting with existing RSS packages on npm. From there you'll follow the design requirements of the project as shown in figure 4.1.

In this diagram, you see the layout of logic and flow of information. Starting from the client (which may be any computer or device with a network connection), a request is made to collect the top feed results from your Node app. From there, your app processes data from multiple RSS sources and parses the data to return a summary list of results. This diagram demonstrates how a list may appear on your command line client.

Figure 4.1. Project blueprint for an application RSS reader and aggregator

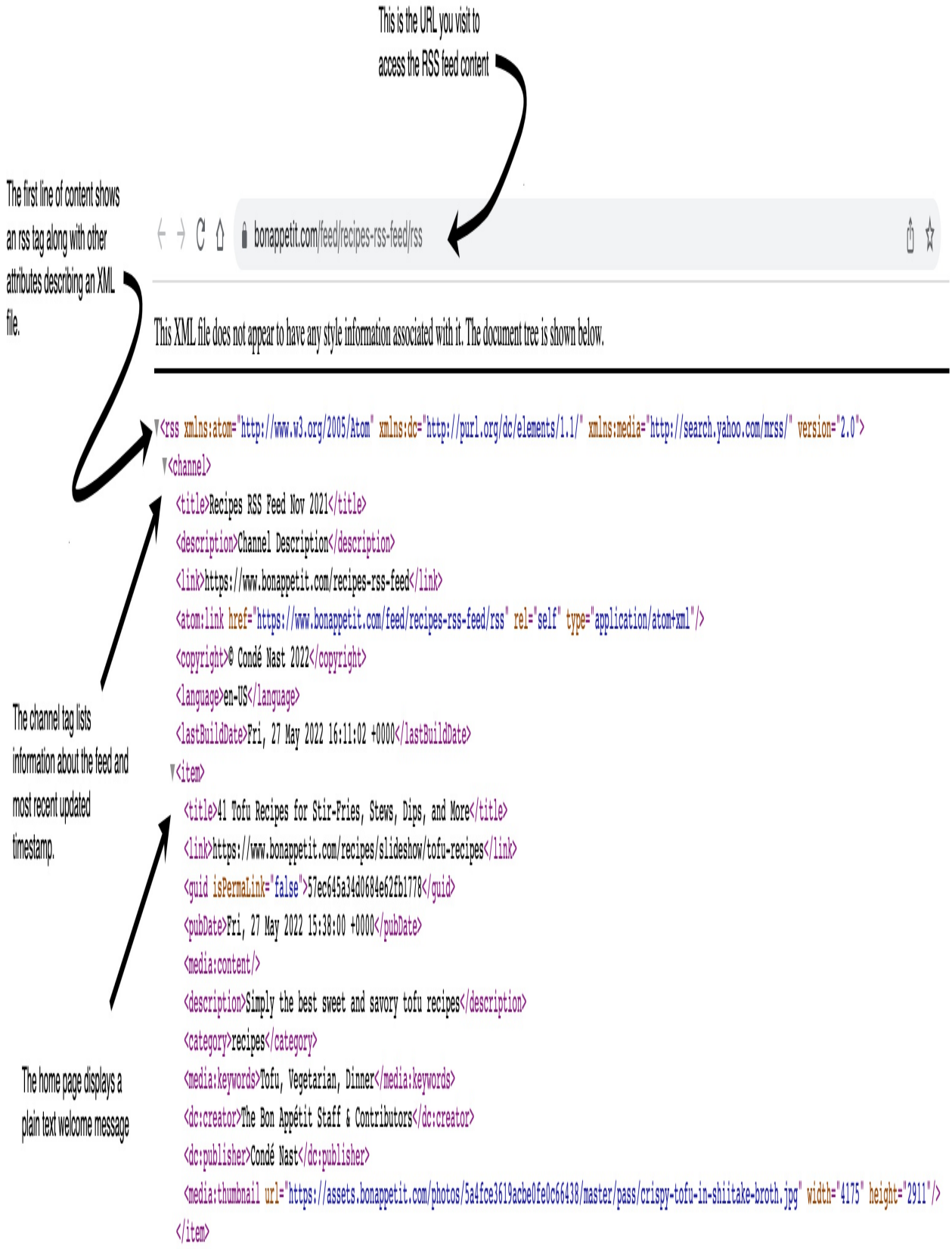


Before you get coding, it may help to review how RSS works and what type of data to expect. RSS works largely because there are sources for content across the web. RSS content typically comes from news sites or blogs that want allow to their viewers quick and easy access to the top headlines of the day. For this to work, a news site must offer an API to access the RSS feed. For example, you may use the Bon Appetit recipes RSS feed for app, which is accessible at https://www.bonappetit.com/feed/recipes-rss-feed/rss. You can click on this link to view the feed contents in your web browser, as shown in figure 4.2.

XML (Extensible Markup Language) is just one of many data formats you can use across the web. The XML structure uses tags similar to those used in HTML web pages. The nested tags help you understand which pieces of data belong to which sections. RSS-feed XML files typically start with an `rss` tag that describes the type of XML being used. After that, there's a tag labeled `channel` to detail information about the source of data. In the case of Bon

Appetit, the `channel` tag lists the name, link, and written language used for their channel's feed. Within the channel tag is where you'll find the main content. Each content listing is wrapped in an `item` tag. Normally, you'll mind many item tags, and it's your job to extract these items and present them in your own way to your app's clients.

Figure 4.2. RSS feed results for Bon Appetit recipes in the browser


This is the URL you visit to access the RSS feed content
bonappetit.com/feed/recipes-rss-feed/rss
The first line of content shows an rss tag along with other attributes describing an XML file.
This XML file does not appear to have any style information associated with it. The document tree is shown below.
<rss xmlns:atom="http://www.w3.org/2005/Atom" xmlns:dc="http://purl.org/dc/elements/1.1/" xmlns:media="http://search.yahoo.com/mrss/" version="2.0">
<channel>
<title>Recipes RSS Feed Nov 2021</title>
<description>Channel Description</description>
<link>https://www.bonappetit.com/recipes-rss-feed</link>
<atom:link href="https://www.bonappetit.com/feed/recipes-rss-feed/rss" rel="self" type="application/atom+xml"/>
<copyright>© Condé Nast 2022</copyright>
<language>en-US</language>
<lastBuildDate>Fri, 27 May 2022 16:11:02 +0000</lastBuildDate>
The channel tag lists information about the feed and most recent updated timestamp.
<item>
<title>41 Tofu Recipes for Stir-Fries, Stews, Dips, and More</title>
<link>https://www.bonappetit.com/recipes/slideshow/tofu-recipes</link>
<guid isPermaLink="false">57ec645a34d0684e62fb1778</guid>
<pubDate>Fri, 27 May 2022 15:38:00 +0000</pubDate>
<media:content/>
<description>Simply the best sweet and savory tofu recipes</description>
<category>recipes</category>
<media:keywords>Tofu, Vegetarian, Dinner</media:keywords>
<dc:creator>The Bon Appétit Staff & Contributors</dc:creator>
<dc:publisher>Condé Nast</dc:publisher>
<media:thumbnail url="https://assets.bonappetit.com/photos/5a4fce3619acbe0fe0c66438/master/pass/crispy-tofu-in-shiitake-broth.jpg" width="4175" height="2911"/>
</item>
The home page displays a plain text welcome message

The first step in building an application that can use XML data is to call the RSS feed URL directly from within your Node app. Get started by creating a `food_feeds_rss_app` folder and navigating to the project folder in your command line. From here run `npm init` to initialize the Node app with the default configurations, as shown in listing 4.1.

Listing 4.1. NPM init configurations for your app

```
javascript
{
  "name": "food_feeds_rss_app", #1
  "version": "1.0.0",
  "description": "",
  "main": "index.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "author": "Jon Wexler",
  "license": "ISC"
}
```

 **Note**

As of Node v18.2.0 you may use the `fetch` api without the need to install an external package. If you are using an earlier version of Node, you'll need to run `npm i node-fetch` to be able to use `fetch` in your app.

More on the Fetch API

Javascript provides many ways to access content across HTTP. There's `XMLHttpRequest` which is the foundation of most browser AJAX requests. Server-side Javascript also offers the `http` library, which comes pre-packaged with Node. Though, many external packages have used the `http` module to offer more comprehensive external packages.

The Fetch API was introduced to Javascript to support a more sophisticated interface for fetching resources across the web. Its main advantage over alternative interfaces is its `Request` and `Response` objects, which encapsulate

many of the functions you need to request and process data asynchronously. Fetch requests also return a Promise, which makes the API flexible with async-await syntax. Due to it's popularity, Fetch was introduced to Node for use on the server-side in 2022. Learn more at https://developer.mozilla.org/en-US/docs/Web/API/Fetch_API.

Listing 4.2. NPM init configurations for your app

```
js
const main = async () => { #1
  const url = "https://www.bonappetit.com/feed/recipes-rss-feed/r
  const response = await fetch(url); #2
  console.log(await response.text()); #3
}
main(); #4
```

4.2 Reading and parsing a feed

Your "Feeding feed" app is designed to both collect data and output it to your users in a meaningful way. Raw XML isn't particularly easy or interesting to read. The tech community recognizes that and sure enough, there are a multitude of external libraries to install that can help you. One of those libraries is in the `rss-parser` package. This package encompasses both the fetching of a feed and the parsing of its XML contents. You can install this package by going to your project's root level on your command line and running the command `npm i rss-parser`.

Once the package is installed, you'll notice that your `package.json` file added a new dependency, and a folder called `node_modules` was created at your project's root level. Next, import the `rss-parser` package into your app by adding `import Parser from 'rss-parser';` to the top of your `index.js` file. On the following line, you instantiate the `Parser` class by adding `const parser = new Parser();`. Now you have a `parser` object you can use in place of your Fetch API code. Replace the contents of your `main` function with the code in listing 4.3. This code implements the `parser.parseURL` function, by fetching the contents of your RSS feed url and preparing them in a structured format. You'll then have access to the feed title and items. In the end, you only log what you want to show from that feed. In this case, it's the

item title and link.

Listing 4.3. NPM init configurations for your app

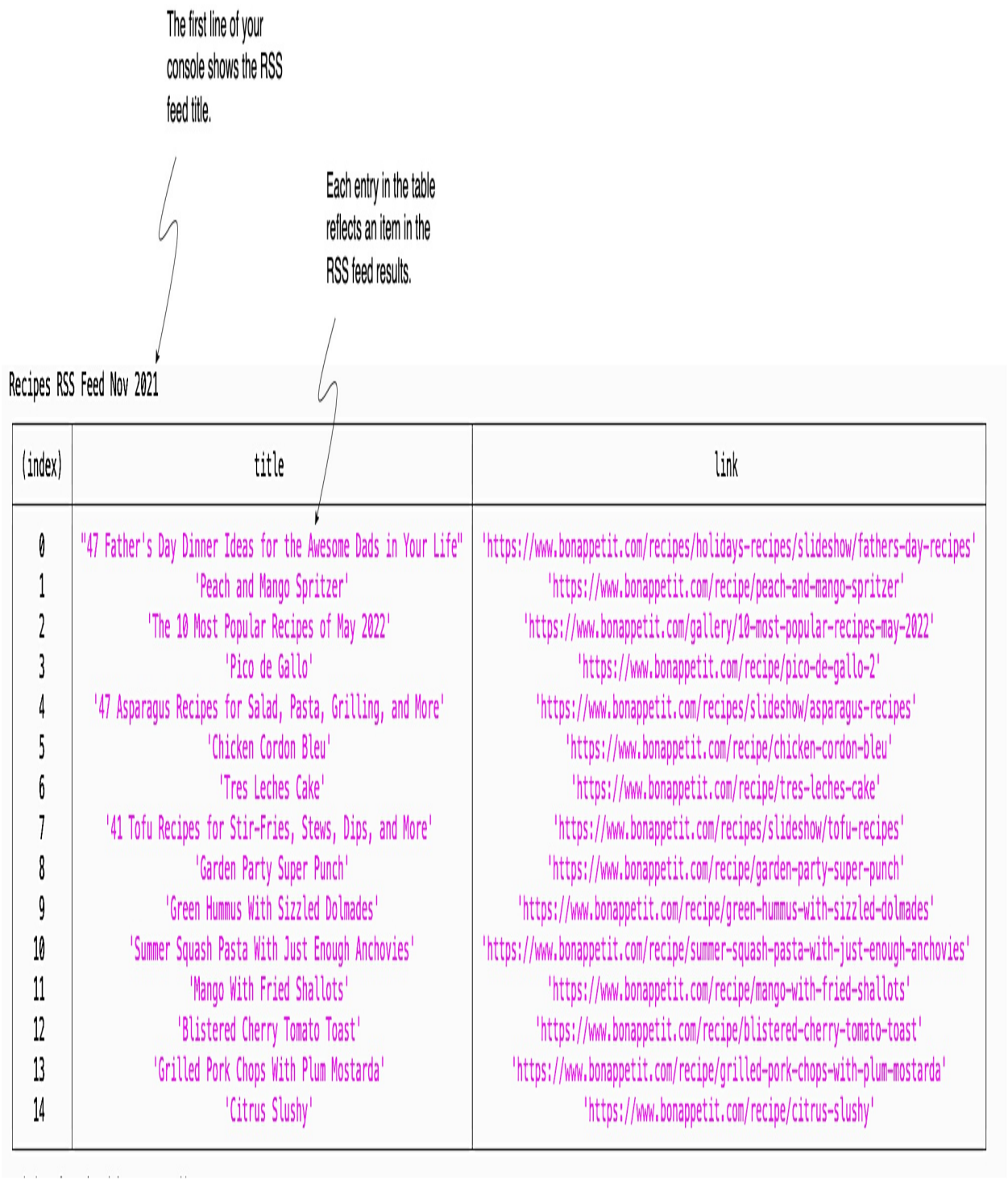
```
js
...
  const url = "https://www.bonappetit.com/feed/recipes-rss-feed/r
  const {title, items} = await parser.parseURL(url); #1
  console.log(title); #2
  const results = items.map(({title, link}) => ({title, link}));
  console.table(results); #4
...
```

 **Tip**

`console.log` is by far the most used logging and debugging function. However, there are other logging types you may use like `console.table`, which prints your content in a format that's easier to read than the former function. Learn more about `console.table` at https://developer.mozilla.org/en-US/docs/Web/API/console/table.

After adding the `rss-parser` code, save your file, navigate to your project's root level on your command line and run the command `node index`. Your output should look similar to that in figure 4.3.

Figure 4.3. Console output for a table of RSS feed items



This output shows you recipe titles and their corresponding URLs. This is a great way to summarize the contents of the Bon Appetit recipes RSS feed, though, this list is static and processes content only the moment you run your app. Because this feed receives updates, it would be ideal for your Node app

to reflect those updates in real time. To fetch new updates every two seconds, change your call to `main()` at the end of `index.js` to `setInterval(main, 2000)`. `setInterval` will keep your Node process running indefinitely, processing a new URL request, parsing, and logging every two seconds (two thousand milliseconds). To make this more apparent in your console, add `console.clear();` in `index.js` right above the `console.table` line to clear your console with each interval. Also, add `console.log('Last updated ', (new Date()).toUTCString() );` right below the table log to print an updated timestamp. Now, when you run your app, while you may not see the feed contents change immediately, you'll notice the updated timestamp changes with each interval.

This command line RSS reader is a great way to have the latest updates from your favorite RSS feed endpoints running on your computer. In the next section, you'll add more external feeds and build your own aggregator to show only the most relevant content.

4.3 Building an aggregator

With your Node app successfully printing RSS feed content to your console, you may be wondering how you can expand the tool to be more practical. After all, your goal is to collect particular recipes that align with the dietary preferences of your coworkers. The good news is your app is designed to handle more content. With the logic in place to fetch one rss feed, you can add more feed URLs to call.

To test fetching from multiple URLs, you can use the Specialty Food lunch feed and the Reddit /r/Recipes subreddit feed. Both of these feeds offer varying content at different times, making it more of a challenge to parse. To incorporate these additional feeds, you add https://www.specialtyfood.com/rss/featured-articles/category/lunch/ and https://www.reddit.com/r/recipes/.rss to the list of URLs to explore at the top of `index.js` (listing 4.4). The `urls` constant will later be used to cycle through each URL and collect its corresponding XML response.

Listing 4.4. Defining the list of URLs to read from in `index.js`

```
js
const urls = [ #1
  "https://www.bonappetit.com/feed/recipes-rss-feed/rss",
  "https://www.specialtyfood.com/rss/featured-articles/category/l
  "https://www.reddit.com/r/recipes/.rss"
];
...
```

With this list in place, you may now modify the `main` function by iterating through each URL to fetch feed content (listing 4.5). First, assign a constant `feedItems` to an empty array: this is where your eventual feed items will be stored. Next, iterate through the `urls` array using the `map` function, which will visit each URL and run the `parser.parseURL` function to return a Promise in its place. In the following line, you use `Promise.all` which waits for all the requests to external URLs to return with responses before completing. Each response will be stored in a `responses` array. Last, you use a custom `aggregate` and `print` function to sift through the responses and log your desired output, respectively.

Listing 4.5. Defining the list of URLs to read from in `index.js`

```
js
const main = async () => {
  const feedItems = []; #1
  const awaitableRequests = urls.map(url => parser.parseURL(url))
  const responses = await Promise.all(awaitableRequests); #3
  aggregate(responses, feedItems); #4
  print(feedItems); #5
}
```

Before re-running your application, you need to define the `aggregate` and `print` functions. Add the code in listing 4.6 below your `main` function. In the `aggregate` function, you collect all the feed data from each external source and, for this project, only retain the items that contain recipes with vegetables. First, loop through the array of responses and examine only the items within each XML response. Then, an inner loop visits each item destructs the `title` and `link` only, because these are the only pieces of data you care about in this project. With access to each item's title, you check if the title `includes` the string `veg`. If that condition passes, you add an object with the `title` and `link` to your `feedItems` array.

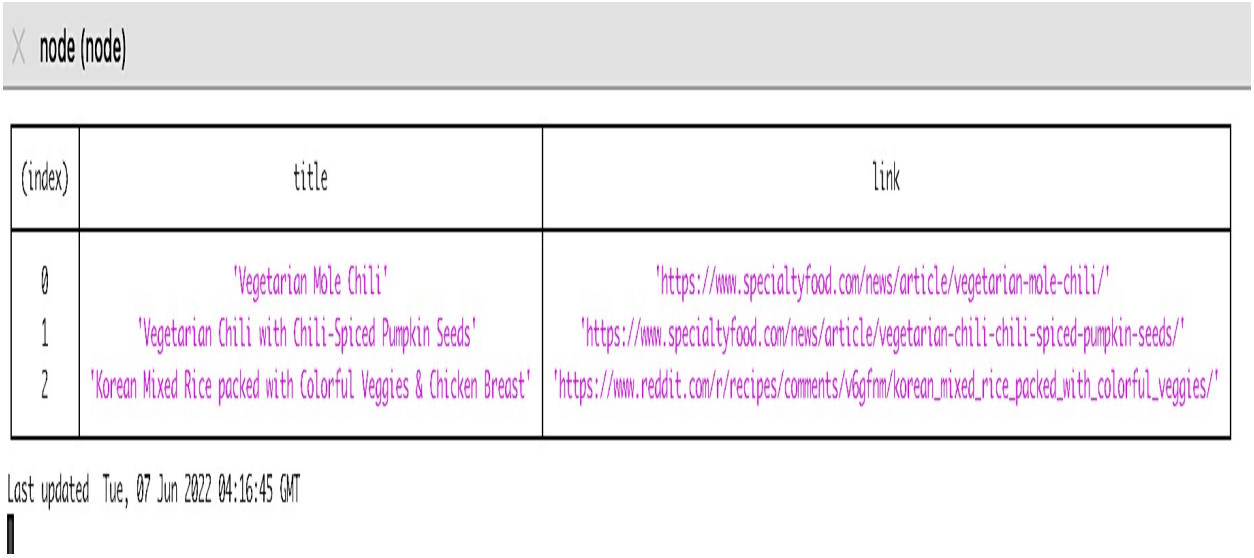
In your `print` function, you accept `feedItems` as an argument. Next, you clear the console of previous logs using `console.clear`. Print your `feedItems` to your console using `console.table` and then log your `Last updated` time by generating a new `Date` object and converting it to a human-readable string.

Listing 4.6. Defining the `aggregate`and `print` functions in `index.js`

```
js
...
const aggregate = (responses, feedItems) => { #1
  for (let {items} of responses) { #2
    for (let {title, link} of items) { #3
      if (title.toLowerCase().includes('veg')) { #4
        feedItems.push({title, link});
      }
    }
  }
  return feedItems; #5
}

const print = feedItems => {
  console.clear(); #6
  console.table(feedItems); #7
  console.log('Last updated ', (new Date()).toUTCString()); #8
}
```

Now, restart your application. You'll notice this time there are fewer results logged to your console (figure 4.4), but the items shown are from varying sources—all with titles indicating some vegetable or vegetarian recipe. You can modify the aggregate condition to your liking by focusing on other key words, or even examining data other than the `title` and `link` used in this example.

Figure 4.4. Console output for an aggregated table of RSS feed items



Ultimately, when you share this aggregator with your colleagues, they can add any additional RSS source URLs to increase the quantity of meaningful results. Before you wrap this project up, you decide to add one more feature: adding custom items to the feed.

4.4 Adding custom items to your aggregator

Most RSS aggregators collect only the results from external feeds and aggregate the results according to some defined rules. It’s not very often you have the opportunity to modify the resulting aggregated list with content not found anywhere else. The client for this project has been your command line console. However, it’s possible to convert this app into a web-accessible tool with the help of a web framework. With a web framework, you can publish the aggregator and allow anyone to read the resulting feed items on their own browser. Though, you do not need to rely on the web, or a browser, to design a standalone Node app.

To collect user typed input, install the `prompt-sync` package by running `npm i prompt-sync` at the root level of your project in your command line. Then, in `index.js` add `import promptModule from 'prompt-sync';` to the top of your file, followed by `const prompt = promptModule({sigint: true});` to instantiate the `prompt` function with a `sigint` config that allows you exit your app. Last, add `const customItems = [];` to define an array for your custom feed items. Next, you modify the `print` function by adding the code in listing

4.7 to the top of that function. The `prompt` function will show `Add item: ` on your console and wait for a typed response. When the Enter key is pressed, the input is saved to a `res` constant. User input should be of the format: `title + , + link`. Then, the `title` and `link` are extracted by splitting the resulting input string. The new custom item object is added to your `customItems` global constant array.

Listing 4.7. Modify the `print` function to accept user input in `index.js`

```
js
 const res = prompt('Add item: '); #1
 const [title, link] = res.split(','); #2
 if (![title, link].includes(undefined)) customItems.push({title,
 ...
```

Finally, modify your log statement to include your `customItems` array by replacing that line with `console.table(feedItems.concat(customItems));`. Now, when you restart your app, you should see a prompt for a custom feed item. If you have nothing to add, simply press the enter key. Otherwise, you may add a custom recipe like `Jon's famous veggie dish,http://jonwexler.com/recipes` and press the enter key to add it to your aggregated items feed. Your result should look like figure 4.5.

Figure 4.5. Console output for an aggregated table with custom feed items

```
node (node)
```

(index)	title	link
0	'Vegetarian Mole Chili'	'https://www.specialtyfood.com/news/article/vegetarian-mole-chili/'
1	'Vegetarian Chili with Chili-Spiced Pumpkin Seeds'	'https://www.specialtyfood.com/news/article/vegetarian-chili-chili-spiced-pumpkin-seeds/'
2	'Korean Mixed Rice packed with Colorful Veggies & Chicken Breast'	'https://www.reddit.com/r/recipes/comments/v6gfnm/korean_mixed_rice_packed_with_colorful_veggies/'
3	"Jon's famous veggie dish"	'http://jonwexler.com/recipes'

```
Last updated  Tue, 07 Jun 2022 05:17:59 GMT
Add item:
```

Now you have an app you can share with others in your office. You can use the aggregator to collect relevant recipes every two seconds (or at an interval of your choice). You can also add custom links not found in your external sources. When users of your app publicize their results, everyone can benefit from your new aggregated collection of quick-access recipe links. You may continue developing the application to make it accessible across a shared network, or build a web framework with a database into your app to allow web clients to access the feed data.

4.5 Summary

In this chapter you

- Learned how to build an RSS feed aggregator
- Ran API calls to retrieve data from external endpoints
- Built an API endpoint to add content to your own RSS feed

5 Library API

This chapter covers

- Constructing an API with Express.js
- Building REST endpoints
- Connecting to a relational database

Although most people are familiar with the internet by way of their web browser, most activity and data transfer happens behind the scenes. Some data, like real-time train schedules, is made available not as just a standalone web app, but as a resource others can use and implement into their own apps. This resource is called an Application Programming Interface (API), and it allows its users to view all or some available data belonging to a restricted environment (like the Railroad authority). Some APIs allow the addition, modification, and deletion of data, especially if it's your own data or if you are the authority over that resource itself.

In this chapter, you'll build a RESTful API, meaning it will support access and modification of data through a standard protocol. You'll also connect the API to a database—to which you can add new information and from which you can access older records. In the end, you'll have the translatable skills needed to build an API for just about any type of data.

Tools & applications used in this chapter

Before you get started, you'll need to install and configure the following tools and applications that are used in this project. Detailed instructions are provided for you in the specified appendix. When you've finished, return here and continue.

- Appendix A.1.2 Installing VS Code
- Appendix A.1.3 Installing Node
- Appendix A.1.5 Installing Postgres
- Appendix A.1.6 Installing Postman

5.1 Your prompt

Your township is awarding grants to individuals who can modernize the public library. At the top of their list is building a system allowing the public to request books not yet provided or in low supply at the library. Grants have already been handed out to groups designing the mobile and web clients, but there's still a need for someone to build the backend API. You are somewhat familiar with Node and Express and decide to take on the challenge.

5.2 Get planning

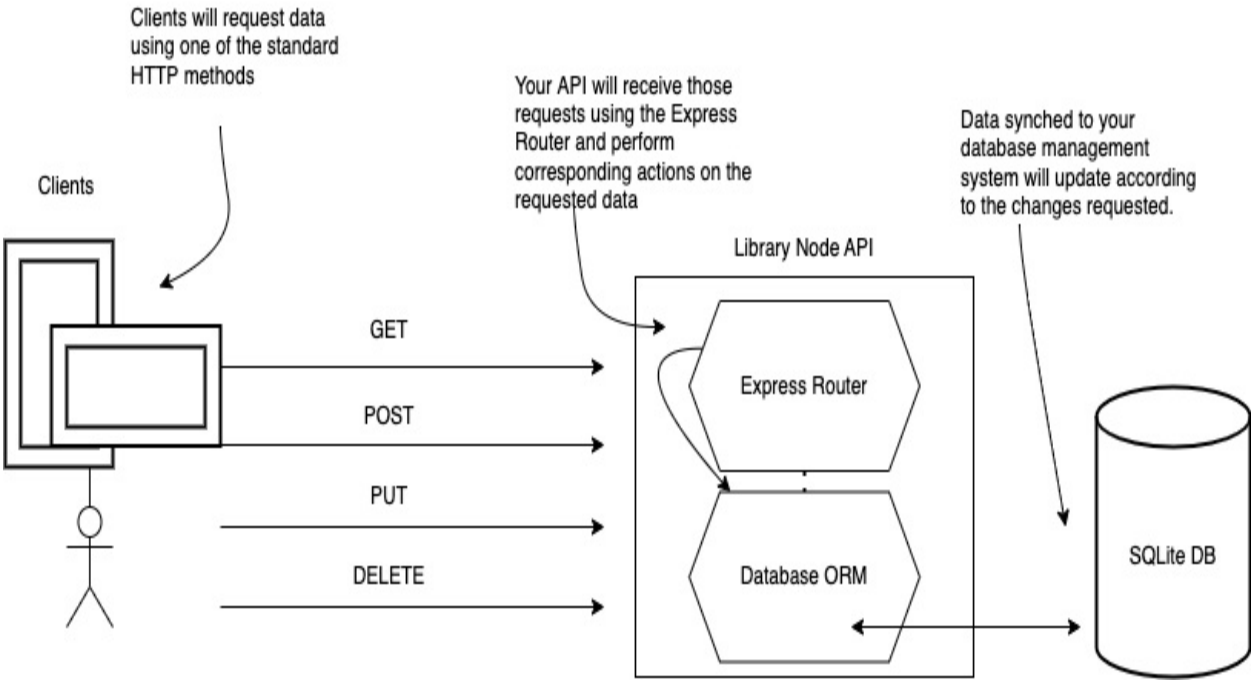
The goal of this project is to build an API that mobile and web apps can connect to in order to view, add, update, or delete book records. Just like on a website's URLs, you'll create an API that provides endpoints (URIs) accessible by other clients. To start, you'll create a Node app with the Express framework and gradually add pieces to support interactions with the library data and persist that data in a database. You follow the design requirements of the project as shown in figure 5.1.

 **Note**

Uniform Resource Identifiers (`URI`) are a superset of endpoints used to access resources or data. In this case, a `URI` is used to fetch data from an API. Uniform Resource Locators (`URL`) are a subset of `URI`, typically associated with accessing a web address.

This diagram shows four behaviors corresponding to HTTP actions (GET, PUT, POST, and DELETE), which you'll need to implement in the API. When a request is made to the API, your app must distinguish between these four request types and perform the relevant app logic on the requested data. In other words, if someone wants to submit a book recommendation, don't accidentally delete a different book from your database. At the end of development, you'll be able to test your API on your web browser, command line, or third-party application like Postman.

Figure 5.1. Project blueprint for a Node API using four HTTP methods



HTTP Methods

Hypertext Transfer Protocol (HTTP) is an internet protocol used by websites and applications to transmit data across the web. HTTP offers an architecture and semantics for sending packets of data from one IP address to another. Part of this architecture includes defined methods for requesting data.

There are nearly 40 HTTP methods that can be used, though only a handful make up the majority of requests made across the internet. The following HTTP methods are ones that you'll use in this book:

- `GET`—This request method is used whenever someone visits a landing page or web page with static content. It constitutes the majority of requests on the internet and is the simplest one to implement in your API.
- `POST`—This request method is used whenever data is sent to a server; for example, if a user logs into their account. These requests are necessary for most data to eventually enter a database and allow for information to persist across web sessions.
- `PUT`—This request method is similar to `POST` in that it sends data in the body of the request. However, this method is used in practice to update

or modify existing data on the server. These requests are distinguished from `POST` data, typically, by an ID that matches a record on the server's database.

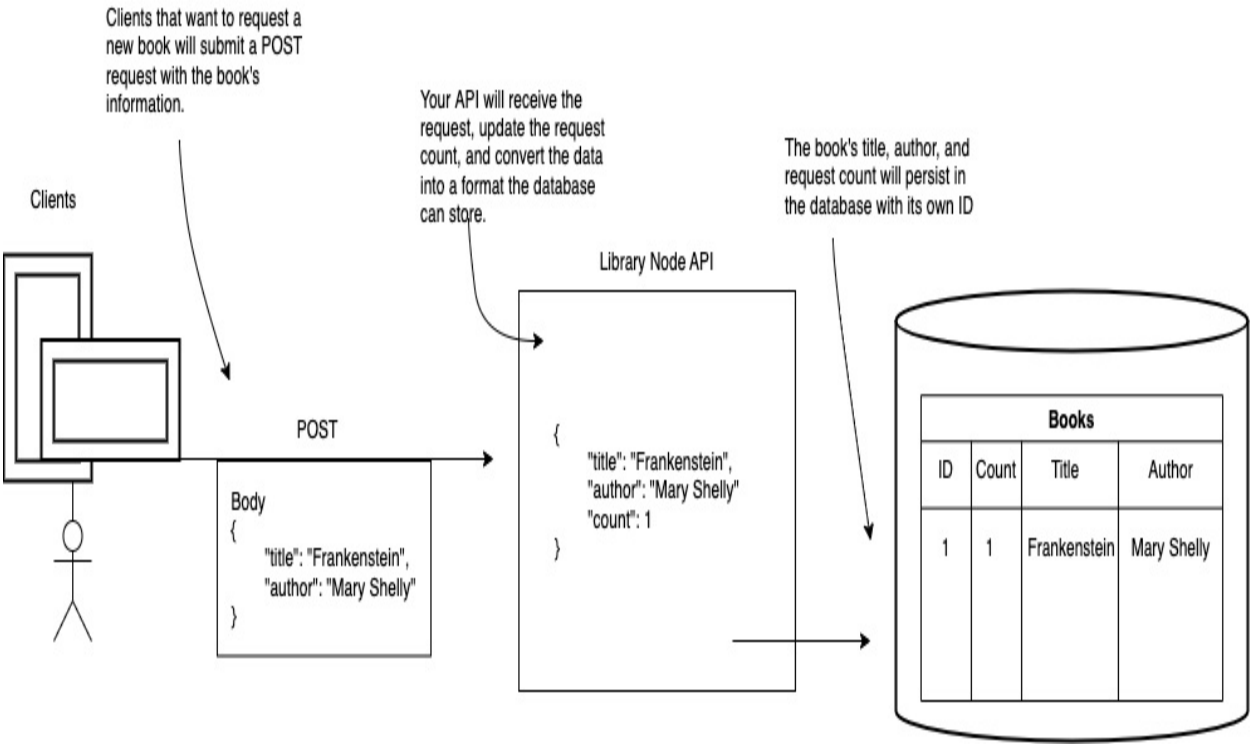
- `DELETE`—This request method effectively sends a primary key to the server, indicating a record that should be deleted. The nature of a `DELETE` request looks similar to a `GET`, but unlike the latter, it expects to change the state of data on the server.

While there are many other request methods used across HTTP, these four are enough to get started with developing an API. For more information on HTTP methods, visit https://developer.mozilla.org/en-US/docs/Web/HTTP/Methods.

Because the library wants your API to bring visibility to popular and sought-after books, the data served by your API should have enough information to identify those books and the level of interest. By the time your database is set up, you'll want to store the title and author of the book, as well as a count of how many requests that book received. In this way, mobile and web clients that use your API can notify the library and its patrons of the most popular requests.

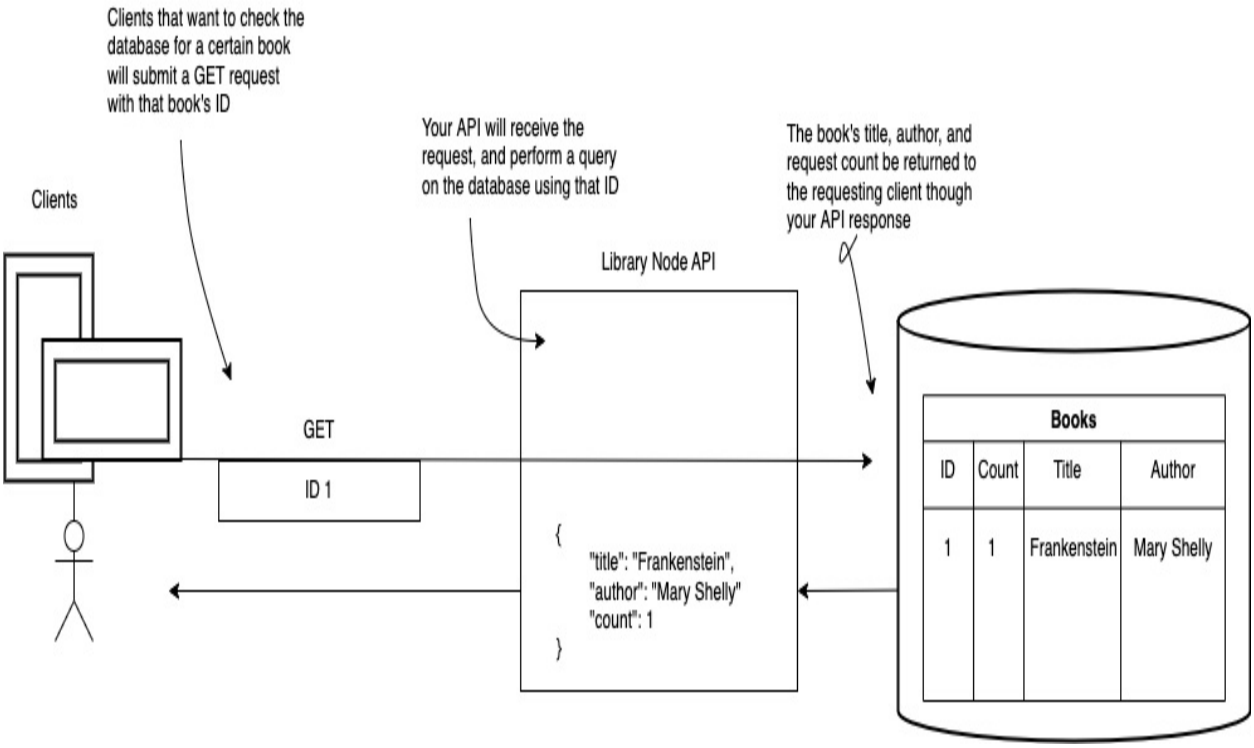
Figure 5.2 shows the flow of data during a `POST` request for a new book. That request contains the title and author of the book. If the book already exists in the database, it's request count increases. Otherwise, that book's record is added to the database for the first time with it's own serial ID.

Figure 5.2. Flow of data during a `POST` request



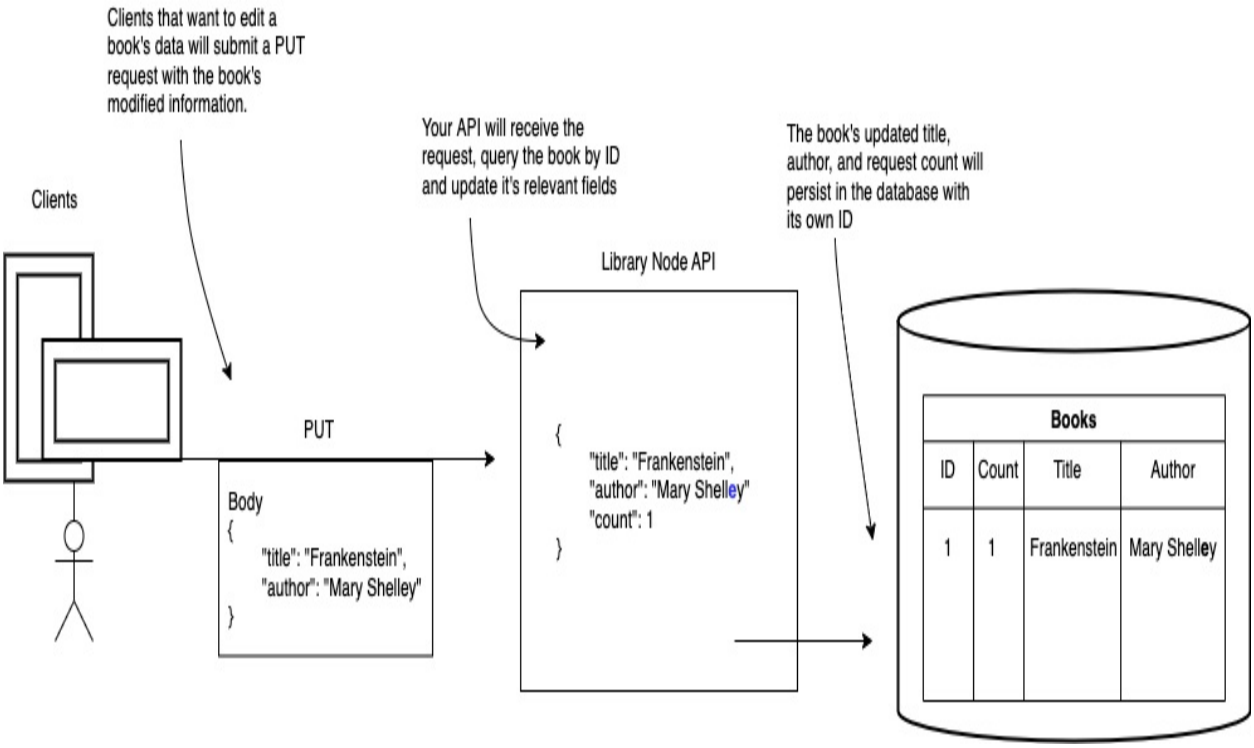
Once there is some persisted (stored) data, library clients are able to send a `GET` request using that book's persistent ID (figure 5.3). The `GET` request needs to send only an ID parameter, but in turn receives all of the book's data.

Figure 5.3. Flow of data during a `GET` request



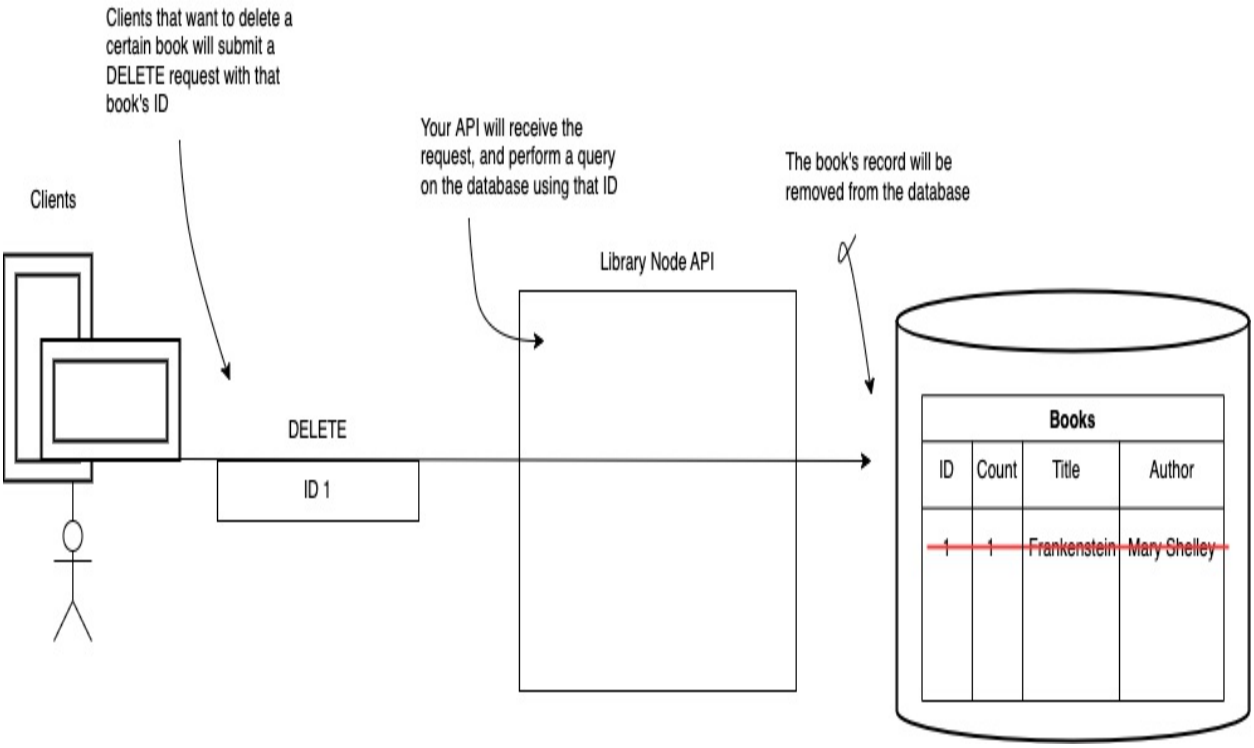
If a library admin notices a mistake in the book's data, they can modify the title or author and submit a `PUT` request. In figure 5.3, the author's name was missing a letter, and a `PUT` is sent along with the book's ID and updated information. That new data is processed within your API and saved to the database under the same record ID.

Figure 5.4. Flow of data during a `PUT` request



Last, if the library obtains enough copies of a certain book to satisfy its patrons, they can submit a `DELETE` request to remove that book from the database. Figure 5.3 shows that a `DELETE` request only needs the book's ID as a parameter. From there, the API can perform a delete query on the database.

Figure 5.5. Flow of data during a `DELETE` request



With the architecture and data structures defined, you're ready to start building your API with Node and Express. In the next section, you layout the building blocks for a typical Node app.

5.3 Get Programming with an API layout

To start developing your app, create a new folder named `library_api`. Navigate to your project folder on your command line and run `npm init`. This command initializes your Node app. You can press the **Enter** key throughout the initialization steps. The result of these steps is the creation of a file called `package.json`. This file instructs your Node app of any configurations or scripts needed to operate.

Next, create a file called `index.js` within your project folder. This file acts as the entry point for your app.

Additional steps to enhance your app

This book encourages you to build each project from scratch. For that reason, some of the packages, configurations, and installation steps that are required

are listed in a separate appendix. To continue building your app, complete the steps in the following document:

- Appendix B Setting up Node app essentials

To set up your app as an API, you install Express by navigating to your project folder on your command line and running the command, `npm i express`. This command installs Express and adds it as a package dependency in your `package.json` file. With Express installed, you import its module and initialize a new Express app instance (*instantiate*), called `app`, as shown in listing 5.1. While developing the app, you define the API `PORT` as `3000` and use the `app.listen` function to set your API to listen for requests on that port number.

Listing 5.1. Instantiating your Express app and start listening for requests in `index.js`

```
js
import express from 'express'; #1
const app = express(); #2
const PORT= `3000`; #3

app.listen(PORT, () => {  #4
  console.log(`Listening at http://localhost:${PORT}`);
});
```

Your app is now ready to accept any HTTP requests. For now, you can test this by running `npm start`. You'll notice the command line console prints a statement, `Listening at http://localhost:3000`, to indicate that your app is running.

 **Note**

With the `nodemon` package installed, you need to run `npm start` only once while developing your app. Every change you make to the project thereafter automatically restarts your Node process and reflects the changes immediately. Please reference appendix B for more information on installing `nodemon`.

Your app is ready to process requests. Though, to support an API, you need

to instruct Express on the type of data your app anticipates receiving. The `app.use` function allows you to pass middleware functions, which process incoming requests before you get to see what's inside.

Middleware functions, as the name implies, sit between the request being received by your app, and the request being processed by custom app logic. Here, you add the `express.json()` middleware function, which parses incoming requests with JSON data. Because this API expects to receive and serve JSON data, it's necessary to include this parsing function.

Similarly the `express.urlencoded()` middleware allows the Express API to parse and understand URL-encoded requests. This helps with deciphering strings that were modifying and encoded for better efficiency over the web. Setting the `extended` field to `true` enabled your app to parse more complicated nested data structures that were serialized in their internet packet.

Add the code in listing 5.2 below your `PORT` definition in `index.js`.

Listing 5.2. Adding JSON and URL-encoded parsing middleware to your Express app in `index.js`

```
js
...
app.use(express.json()); #1
app.use( #2
  express.urlencoded({
    extended: true,
  })
);
...
```

To complete the first stage of building your API, add the code in listing 5.3 right below your middleware functions. In this block, `app.get` defines a route which listens for `GET` requests only. The `"/"` indicates that your app is listening for requests made to the default URI endpoint: http://localhost:3000/. That means if the HTTP request uses the `GET` method and targets the default URI, the provided callback function is executed. Express provides your callback function with a request (`req`) and response (`res`) object as parameters. The `req` object is used to examine the contents of

the request, while the `res` object lets you assign values and package data in the response to the client.

 **Note**

By some conventions, variables that are not used in a function, but still defined, have an underscore applied to their name. This helps the next engineer know not to expect that variable to have any behavior in the function. For that reason, the request argument in this example is called `_req`.

Once a request is processed, you use the `res.json` function to reply to the client with structured JSON containing a key called `message` and value of `"ok"`. This response indicates to the client that the API server is functioning correctly.

In the following block of code, `app.use` sets up a function to handle all other requests that are not handled by your `GET` route. This is called a *catch-all* error-handling middleware. In this function, the error object, `e`, is the first argument passed in. In this way, if an error occurs, your app won't just crash but it will log the error message and return a status code of 500 (internal server error) instead.

Listing 5.3. Adding a `GET` route in `index.js`

```
js
...
app.get("/", (_req, res) => { #1
  res.json({ message: "ok" }); #2
});

app.use((e, _req, res) => { #3
  const {message, stack, statusCode } = e; #4
  console.error(message, stack); #5
  res.status(statusCode || 500).json({ message }); #6
});
...
```

Save your changes and navigate to http://localhost:3000 in your web browser. You should see `{ message: "ok" }` printed on the screen.

 **Note**

If you don’t see a message appear in your browser, it’s possible that the URL or port number was entered incorrectly. Also, make sure that your server is running. In case of a server error, it’s possible that the server will exit its process and wait for your fix before restarting.

In the next section you’ll add the other necessary routes to support your library API.

5.4 Adding routes and actions to your app

Your app is running, but lacks the ability to differentiate between request types. Moreover, you’ll eventually need the ability to create, read, update, and delete (CRUD) your data. These four actions map to the four main HTTP methods you’ll support with Express routes. From this point forward, you’ll refer to the callback functions within your Express routes as your CRUD actions.

Before you add more code to `index.js`, it’s important that your project structure is maintained and organized. So far, you have one JavaScript file in your project. By the end of this section your project directory structure should look like listing 5.4. Now you’ll introduce a new folder to separate your routing logic from the rest of your app logic.

 **Note**

The `node_modules` was automatically generated and appears within your project folder whenever you install a new external package.

Listing 5.4. Routing directory structure layout

```
bash
library-api #1
  |
  - index.js
  - routes #2
```

```
    |
    - index.js
    - booksRouter.js
  - package.json
  - node_modules
```

Navigate to your project folder in your command line and create a new folder called `routes`. Within that folder create two new files, `index.js` and `booksRouter.js`. Within the `booksRouter.js` file add the code from listing 5.5.

This code introduces the Express Router. The Router class contains Express' framework logic for handling all types of internet requests. You've already used the Express app instance to create a `GET` route in `index.js`. Now, you're going to more explicitly define your routes with an instance of the Router class you call `booksRouter`.

This custom router is named `booksRouter` because it is used only to define routes that have to do with creating, reading, updating, or deleting book records. The first of those routes is the `GET` route, which is registered using `booksRouter.get()`. Within this function, `"/:id"` indicates that a value can be passed into the endpoint, and Express should translate that parameter as a variable by the name `id`. This way, when a client asks for the book record by ID 42, you can use the integer 42 to query your database for a book with a matching primary key. Next, you destructure the `id` value from the request's `params` object.

Because there is no database set up yet, you return the ID to the client in a JSON structure. You wrap the response in a try-catch block in case anything goes wrong, you'll be able to log your errors to the API server. If something doesn't work as expected, the `next()` function passes your error to the next middleware or error handling function in your Express app.

Listing 5.5. Adding a `GET` route for books in `booksRouter.js`

```
js
import {Router} from 'express'; #1
const booksRouter = Router(); #2

booksRouter.get("/:id", async (req, res, next) => { #3
```

```
  const { id } = req.params; #4
  try {
    const book = { id }; #5
    res.json(book); #6
  } catch (e) {
    console.error("Error occurred: ", e.message); #7
    next(e); #8
  }
});
```

To test this, add `export default booksRouter` to the bottom of your `booksRouter.js` file to allow this module to be accessed elsewhere in your application. Next, open the `index.js` file within your project's `routes` folder and add the code in 5.6. Similar to the code in `booksRouter.js`, this `index.js` imports and instantiates the Express Router. However, this file does not define new routes, but simply organizes the existing routes under a namespace. `mainRouter.use('/books', booksRouter)` instructs the router to handle all requests with a `/books` URI path to be handled by the `booksRouter` routes. After this change, you'd expect a `GET` request to be sent to `http://localhost:3000/books/42` for a book with ID 42.

Listing 5.6. Exporting your app routes in `routes/index.js`

```
js
import {Router} from 'express'; #1
import booksRouter from "./booksRouter.js"; #2

const mainRouter = Router(); #3

mainRouter.use('/books', booksRouter); #4

export default mainRouter; #5
```

With the `mainRouter` set up, you import this router into `index.js` at your project's root level by adding `import router from './routes/index.js';`. Then, you have your app use this new router by adding `app.use("/api", router);` right above the error handling block in `index.js`. This new code defines an additional namespace called `/api`. This will be the final namespace change and will allow you to reach make `GET` requests to the `/api/books/:id` route path.

RESTful Routes

This project uses routing to navigate incoming requests through your app. A *route* is simply a way to get from a specified URI endpoint to your app logic. You can create any types of routes you choose, with whatever names you'd like, and as many dynamic parameters. However, the way you design your routing structure has side-effects and consequences for those using your API.

For that reason, this project uses Representational State Transfer (REST) as a convention for structuring your routes. REST provides a standard URI endpoint arrangement that let's its users know what type of resource they should except to get in return. For example, If you are looking for a particular book in the database. Your endpoint could be: `/Frankenstein/database_books/return_a_book/`. While, this route path includes most of the information I need to get the book's information, it may not follow the same structure for all the resources offered by my API.

A RESTful API empowers its users to quickly and easily understand which part of the route path refers to the resource name and which parts include the necessary data for a database query. In this way, a route like `/books` can be used for both a `GET` and `POST` request, with the server understanding that different logic handles each request for the same resource: books. Furthermore, a route like `/books/:id` adds to the resource name, but providing a dynamic parameter: id. This standard structure makes using and designing an API straightforward and convenient for everyone involved. For more information on RESTful routing visit https://developer.mozilla.org/en-US/docs/Glossary/REST.

Now, restart your app if it's not already running and navigate to `http://localhost:300/api/books/42` in your web browser. You should see `{ "id": "42" }` printing in your window. With your `GET` route working, it's time to add the routes for `POST`, `PUT`, and `DELETE`. Conveniently, your `PUT` and `DELETE` routes look identical to your `GET` route. All three require an `id` param. Duplicate the `GET` route twice, but change one of the duplicates' route function to `booksRouter.put` and the other to `booksRouter.delete`. This addition should be enough to test those routes. For the `POST` route, add the code in 5.7 right above your `export` line at the bottom of `booksRouter.js`.

In this route, `booksRouter.post` is used to have Express listen for `POST` requests, specifically. Within the action, you destructure the `title` and `author` from the request body and return them to the client in JSON format. The body of a request is typically where you'll find request data when posting to create or change information on the server.

Listing 5.7. .booksRouter.js

```
js
...
booksRouter.post("/", async (req, res, next) => { #1
  const {title, author} = req.body; #2
  try {
    const book = {title, author}; #3
    res.json(book);
  } catch (e) {
    console.error("Error occurred: ", e.message);
    next(e);
  }
});
...
```

With these last changes, its time to test your other non-`GET` routes. To test these routes, you open a new command line window and run a cURL command against your API server.

 **Note**

Client URL (cURL) is a command line tool for transferring data across the network. Because you are no longer only requesting to see data, you can use this approach to send data to your server directly from your command line.

1. Test the `GET` route again by running `curl http://localhost:3000/api/books/42`. The resulting text on your command line console should read `{"id":"42"}`.
2. Submit a `POST` request by entering `curl -X POST -d 'title=Frankenstein&author=Mary Shelley' http://localhost:3000/api/books/`. This command uses `-X` to specify a `POST` method and `-d` to send request body data. The result of this

command should look like `{"title":"Frankenstein","author":"Mary Shelley"}`.

3. Next, try a `PUT` request by running `curl -X PUT -d 'title=Frankenstein&author=Mary Shelley' http://localhost:3000/api/books/42`. This request contains both an ID and data in the body. Your response should show `{"id":"42"}`.
4. Last, run `curl -X DELETE http://localhost:3000/api/books/42` to see the same `{"id":"42"}` response for a `DELETE` request.

Now that all four routes are accessible, it's time for the final piece of the puzzle: persistent storage in a database.

5.5 Connecting a database to your app

Data storage can work simply or balloon into a complex problem depending on how you architect your app. Due to Node's popularity, just about any type of data store and database management system can be used with JavaScript. For this project, you can choose whether to use a NoSQL database like MongoDB, or a SQL or relational, database.

Despite not yet introducing other types of data other than book titles and authors, you choose to save your data in relational database tables. You figure that eventually this project might incorporate the massive amounts of information elsewhere in the library system, and so a relational database may be appropriate.

 **Note**

There is no wrong choice when it comes to database selection. At this stage in the project's development, all popular database options will work fine.

Although you've narrowed your decision to a SQL database, there are many different database management systems to choose from. You decide to compare using a SQLite DB and a PostgreSQL DB.

Comparing SQLite and PostgreSQL

SQLite and PostgreSQL are two of the most widely used Relational Database Management Systems (RDBMS). While SQLite is capable of handling data in most cases, as its name implies, this RDBMS is lightweight and requires significantly less setup than other systems. SQLite is considered an *embedded* database because it requires no additional server to connect to the database. In this way, SQLite is the preferred choice for quickly developing an app for demonstration, or even for long term use in an app with minimal traffic.

PostgreSQL, like SQLite, is open-source and supports a relational structure between data elements. PostgreSQL takes adds another layer by supporting Object-relational mapping, which supports persistent storage of more data types. PostgreSQL runs on a separate server which adds more overhead to the overall development process.

Learn more at https://hevodata.com/learn/sqlite-vs-postgresql/.

Overall, although both databases are sufficient for this project, you find that SQLite will get your app up and running the fastest.

You start to incorporate SQLite by installing its most recent npm package and running the command, `npm i sqlite3`. Now that you have an RDBMS installed, you could just connect to the database and start running SQL queries to search, save, modify, and destroy data. But, what's the fun in developing a Node API if you couldn't do it all purely in JavaScript.

Install another package called `sequelize` by running `npm i sequelize` in your command line. `sequelize` is an object relational mapper (ORM) between JavaScript objects and SQL databases. With this package installed, you can define JavaScript classes with `sequelize` and have them automatically map functions to their corresponding SQL queries. So, no SQL knowledge is needed for this project (or really any in this book).

Before you continue, take a look at your project directory. In this last section you'll add two more sub-folders as shown in listing 5.8. Create the first folder, `models`, and within it create a file called `book.js`. This file contains the code needed by Sequelize to map your book data to the database. Next, create the `db` folder. Within this folder create a file called `config.js`, which will contain all the configurations needed to set up your database. After

adding all the required changes, your database will live within the application folder in a file called `database.sqlite`.

Listing 5.8. Project directory structure with a database

```
bash
library-api #1
  |
  - index.js
  - routes
    |
    - index.js
    - booksRouter.js
  - models #2
    |
    - book.js
  -db #3
    |
    - config.js
    - database.sqlite
  - package.json
  - node_modules
```

Open your `config.js` file and add the code in listing 5.9. In this code, you import Sequelize and instantiate a new database connection using SQLite, defining the storage location within the `db` folder of your project. You then authenticate the connection to the database through `db.authenticate()`. If the connection is successful you'll get a logged statement indicating so. Otherwise you'll log the error that occurred while trying to connect. Luckily, there is no additional server to run with SQLite, so there should not be many issues to troubleshoot at this step. At the end of the file you export both the `Sequelize` class and `db` instance.

Listing 5.9. Setting up the database configuration in `config.js`

```
javascript
import { Sequelize } from "sequelize"; #1

const db = new Sequelize({ #2
  dialect: "sqlite",
  storage: "./db/database.sqlite",
});
```

```
try {
  db.authenticate(); #3
  console.log("Connection has been established successfully.");
} catch (error) {
  console.error("Unable to connect to the database:", error); #4
}

export default { #5
  Sequelize,
  db,
};
```

The database is almost ready to get fired up, but first it needs some data to map in your app. Add the code from listing 5.10 to `book.js`. In this file you import the database configs and destructure the `Sequelize` and `db` values. Then, you use the `db.define` function to create a Sequelize model called `Book`. This model name later maps in the SQLite database to create a corresponding table of the same name. The fields of this model reflect the data your library wants you to store:

- a title as a string(`title`)
- author name as a string (`author`)
- number of requests made for the book as an integer (`count`)

These fields are all that's needed to save countless book records in your database (though you will be counting). `Book.sync` will initiate a sync with the database and set up a table called `Books`. At the end of the file, you export the model for use back in your `booksRouter.js` file.

 **Tip**

Passing the option `{force: true}` to `Book.sync` ensures that with each startup of the app, the sync function attempts to create a fresh table if any changes occurred since the last run. This is helpful in development if you don’t want to fill your database with too many test records.

Listing 5.10. Defining a Book model in `book.js`

```
javascript
import config from '../db/config.js'; #1
```

```
const {Sequelize, db} = config; #2

const Book = db.define('Book', { #3
  title: { #4
    type: Sequelize.STRING,
    unique: true
  },
  author: { #5
    type: Sequelize.STRING
  },
  count: { #6
    type: Sequelize.INTEGER,
    default: 0
  },
}, {});

Book.sync(); #7

export default Book; #8
```

With your Sequelize model set up, you'll need to revisit the CRUD actions you previously built in `booksRouter.js`. These actions currently return the data they receive. Now that you have access to a database, you can add the logic needed to support actual data processing in your API.

First, import your Book model into `booksRouter.js` by adding `import Book from '../models/book.js';` to the top of the file. This gives access to the `Book` ORM object and allows you to create, read, update, and delete Book data. Change the values assigned to the `book` and `books` variables in each route to use the result of your database queries (listing 5.11).

The first change makes a call to `Book.findByPk`, where the `id` from your request params is passed in as a primary key to search within the database for a matching book record. You use the `await` keyword as you're making a blocking call to the database, waiting for a response before you continue executing subsequent logic. The next change is to the `POST` request, `booksRouter.post`, where you use the `Book.create` function and pass in the `title` and `author` you retrieved earlier from the request body. You wait for the `create` function to complete and return the resulting created record to the client. Similarly, the `Book.update` also takes in the `title` and `author` as parameters, but this time they reflect the changed `title` and `author` values. A second parameter in this `PUT` request uses a `where` key to identify the

record to update by its primary key: `id`. Last, `Book.destroy` uses the `where` key to search for a record by the specified `id` ` in the `DELETE` request and remove that matching record from the database.

For more information about model query types and the sequelize API, visit https://sequelize.org/docs/v6/.

Listing 5.11. Updating routes with the Book model in `booksRouter.js`

```
javascript
...
// GET /books/:id
const book = await Book.findByPk(id); #1
...
// POST /books/
const book = await Book.create({title, author}); #2
...
// PUT /books/:id
 const book = await Book.update({title, author}, { #3
    where: { id }
  });
...
// DELETE /books/:id
const book = await Book.destroy({ #4
  where: { id }
});
...
```

Now test your changes, only this time, there are different outcomes because each command results in a database action. Notice the `id` field that is returned in some of the responses. Also notice the `updatedAt` and `createdAt` fields Sequelize adds automatically to keep track of when data has entered the database or changed. Return to your command line, open a new window, and run the following cURL commands:

1. Submit a `POST` request by entering `curl -X POST -d 'title=Frankenstein&author=Mary Shelly' http://localhost:3000/api/books/`. This command uses `-X` to specify a `POST` method and `-d` to send request body data. The result of this command should look like `{"id":1,"title":"Frankenstein","author":"Mary`

`Shelly","updatedAt":"2022-07-13T21:37:53.372Z","createdAt":"2022-07-13T21:37:53.372Z"}`.

2. Test the `GET` route again by running `curl http://localhost:3000/api/books/1`. The resulting text on your command line console should read `{"id":1,"title":"Frankenstein","author":"Mary Shelly","requests":null,"createdAt":"2022-07-13T21:37:53.372Z","updatedAt":"2022-07-13T21:37:53.372Z"}`. This record was saved to the database in the last request. Now it is retrievable by ID.
3. Next, try a `PUT` request by running `curl -X PUT -d 'title=Frankenstein&author=Mary Shelley' http://localhost:3000/api/books/42`. This request contains both an ID and data in the body. Your response should show `[1]` to indicate that 1 record was changed. You may run the `GET` request again to see the updated value.
4. Last, run `curl -X DELETE http://localhost:3000/api/books/1` to see `1` as your response for a `DELETE` request to indicate that 1 record was deleted. Running the `GET` request again should return `null`, as the record no longer exists.

 **Note**

If you run the `POST` request a second time with the same data you get a `Validation error` in your server's console. This is expected by design because your Book model has a validation criteria that new books should have unique titles.

Your API is not set up to handle new incoming requests to create, read, update, and delete Book records. If you choose to expand your API, you can add new models or modify the logic in your existing actions.

5.6 Summary

In this chapter, you built a fully functional API with Node and Express. Moreover, you added a SQL database and used the Sequelize library to

persist data processed in your API logic. With the skills you've learned from this chapter, you may now:

- Design custom APIs
- Build an API with any SQL database
- Connect RESTful endpoints for new resources in the API

Try this out

Your Book model can support updating the request count for books already in the database. What can you change in the `POST` request to update the count value before saving the new Book record?

Answer: You can search for a Book record by the same title (which should be unique): `const book = await Book.findOne({where: { title }});`. If that Book exists, you can get the book's existing count `book.count` and add 1 to it and save it to the database:

```
javascript
book.count += 1;
book.save();
```

Try this out

You may want to search for all the books in the database, not just the ones you know by ID. What might a route for an index of all books look like?

Answer: The route would look the same as the `GET` route for a book by `id`, but you would leave the `:id` param out of the route path and then search for all book records instead of just one by its primary key. Here is an example of what that would look like:

```
javascript
booksRouter.get("/", async (req, res, next) => {
  try {
    const books = await Book.findAll();
    res.json(books);
  } catch (e) {
    console.error("Error occurred: ", e.message);
    next(e);
```

```
  }
});
```

Appendix A. Getting set up with installations

In this appendix you'll

- Install development tools needed to build tiny projects
- Set up your development environment with Node.js
- Install all relevant database management systems for projects in this book

This chapter will walk you through the environment setup and installation steps needed to start programming the projects in this book. You do not need to install every tool in this chapter, though you may benefit from installing the required software and libraries for the projects you plan to complete.

 **Note**

Throughout this chapter you'll have the option to install through a Graphical User Interface (GUI), a third-party tool, or the binary packages (non-compiled source code). While the binary packages are sometimes pre-compiled, there are often a few steps required to extract their contents for a successful installation. I recommend only using the binary installation steps when working on a machine without a graphical interface, like a standalone server.

A.1 Following along with code

Each chapter in this book teaches you how to build a standalone application with Node. Each corresponding application requires you to write a moderate amount of code yourself. For this reason, all of the code detailed in this book ha been made available at https://github.com/JonathanWexler/tiny-node-projects-code.

This link will take you to a public Github repository where you'll find separate folders that match the chapter order in this book. Within each chapter folder you will find sub-folders that correspond to the section numbers in each chapter. For example, if you are working on writing the code in section 2 of chapter 1, you may reference the application code by visiting https://github.com/JonathanWexler/tiny-node-projects-code/tree/master/chapter%201/1_2.

You may also choose to clone this repository to your computer. You may follow the instructions at https://github.com/git-guides/git-clone. For more information on downloading and working with Git on your computer visit https://git-scm.com/downloads.

The code in this repository will change to reflect modifications and updates in this book, as well as code security and patch updates to packages taught here.

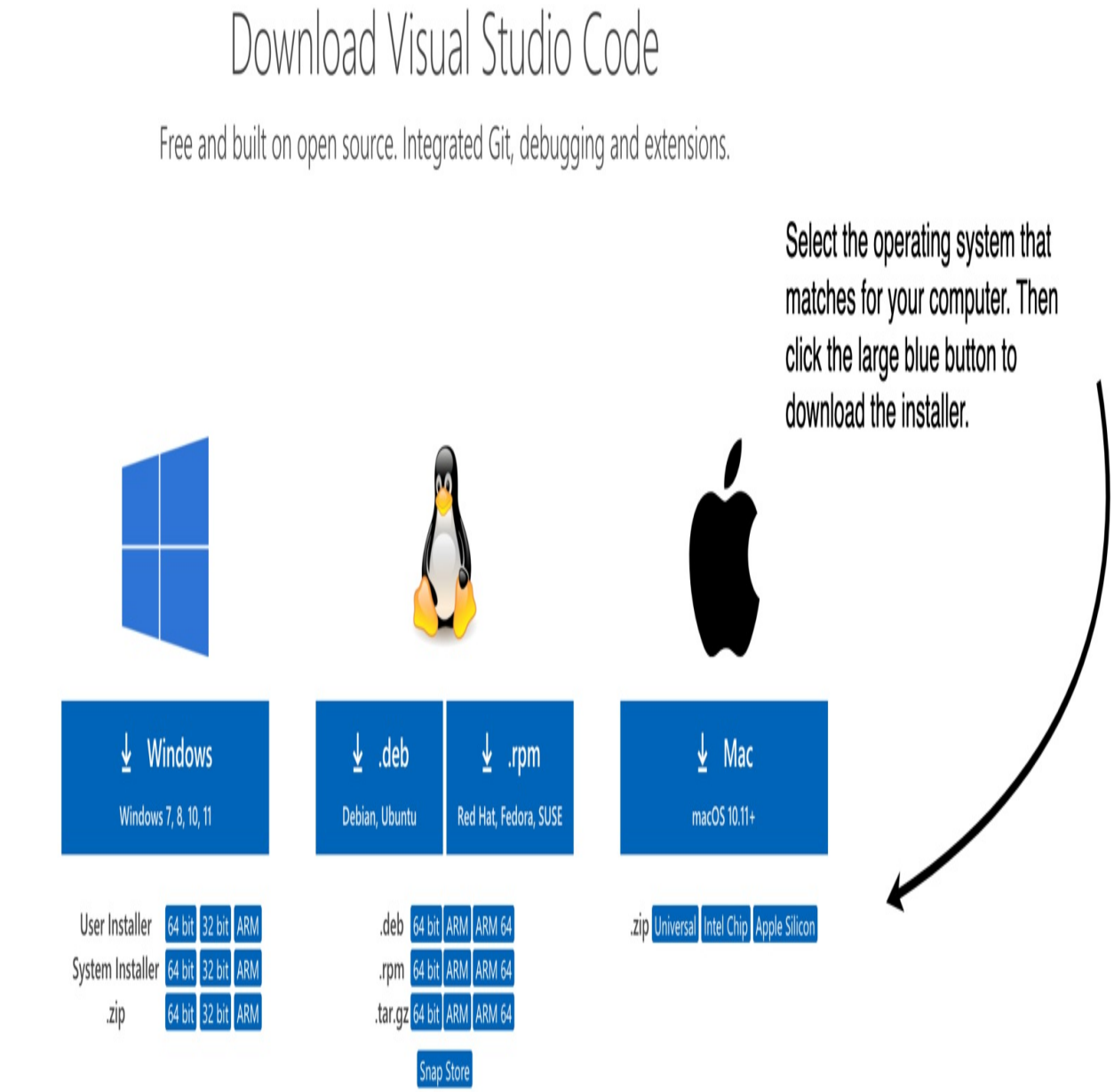
A.2 Installing VS Code

Visual Studio Code (VS Code) is a popular source-code editor designed and supported by teams at Microsoft. In recent years, it has risen as one of the most used code editors in the tech industry, both for personal and professional projects. For this reason and more, I recommend using VS Code as your primary code editor throughout the Tiny Node Projects chapters.

In this section you will find installation instructions for Mac, Windows, and Linux computers. All GUI installation instructions can also be found at https://code.visualstudio.com/.

To install VS Code through a visual interface, go to your web browser and enter the URL: https://code.visualstudio.com/download. You should see a page load with icons representing the Mac, Windows, and Linux operating systems, as shown in figure A.1. From here, you may follow the operating system-specific installation steps.

Figure A.1. VSCode installation page

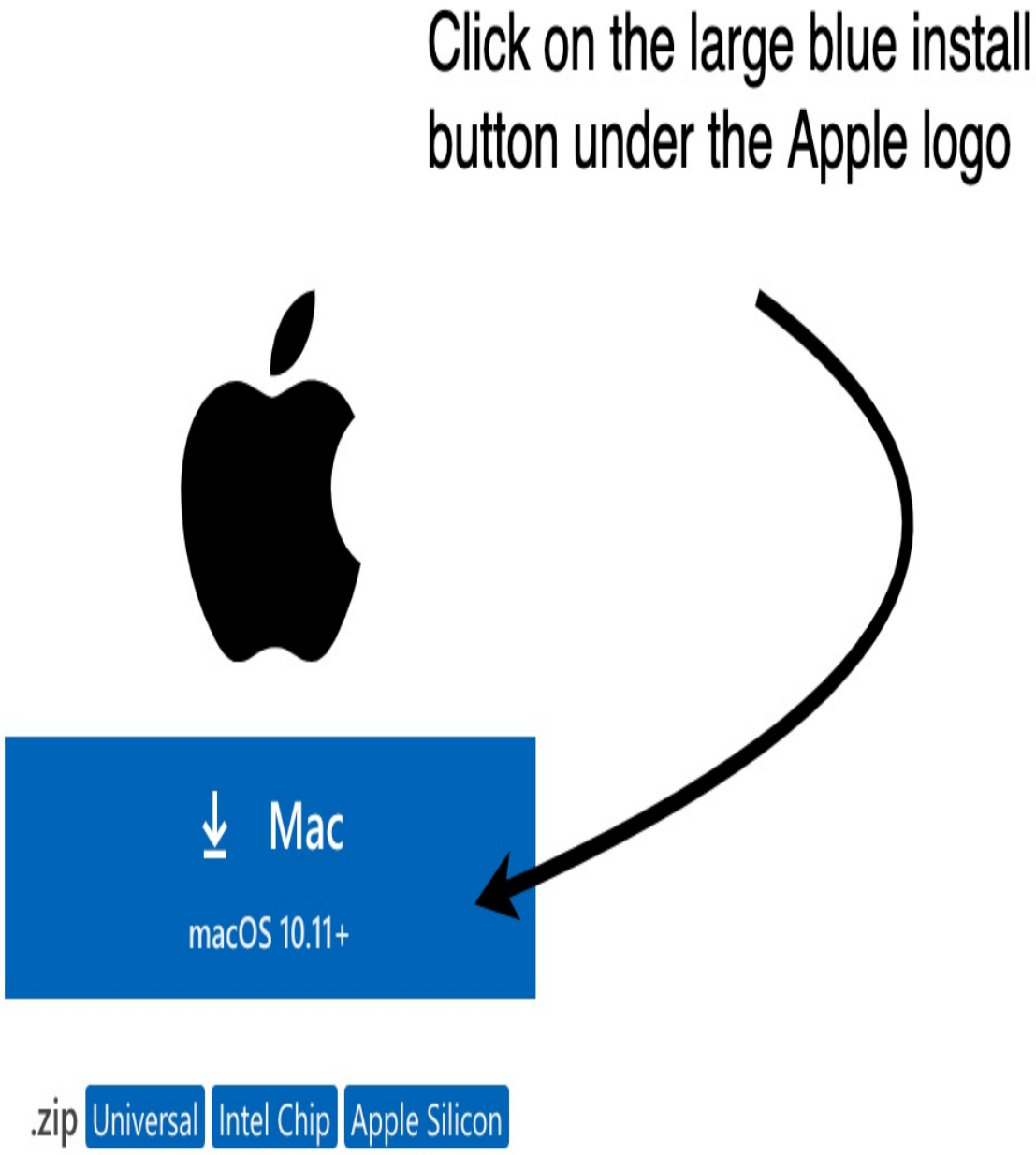


A.2.1 Mac installation

If you are using an Apple computer running macOS 10.11 or higher, you may click on the blue button under the Apple logo as shown in figure A.2.

Clicking this button in your web browser will download the VS Code installation ZIP file with everything you need to get started.

Figure A.2. VSCode installation for Mac



Once the ZIP file downloads to your web browser’s specified download location (this could be your downloads folder or desktop), double-click the ZIP file to unzip its contents. Once the contents are unzipped you will see a

folder appear with the VS Code app logo. Select the VS Code icon and drag it to your computer's **Applications** folder. Now you have VS Code installed on your computer.

 **Tip**

You will be working largely from your command line. You can set up VS Code to open from your command line window using the `code` keyword. To set this up, open VS Code, open the Command Palette by typing **Cmd**+**Shift**+**P**. You'll see an input box appear where you can type `Shell Command: Install 'code' command in PATH`. Select the matching option from the dropdown. Now you may restart your command line and simply type `code` and press **Enter** to open VS Code from your command line window.

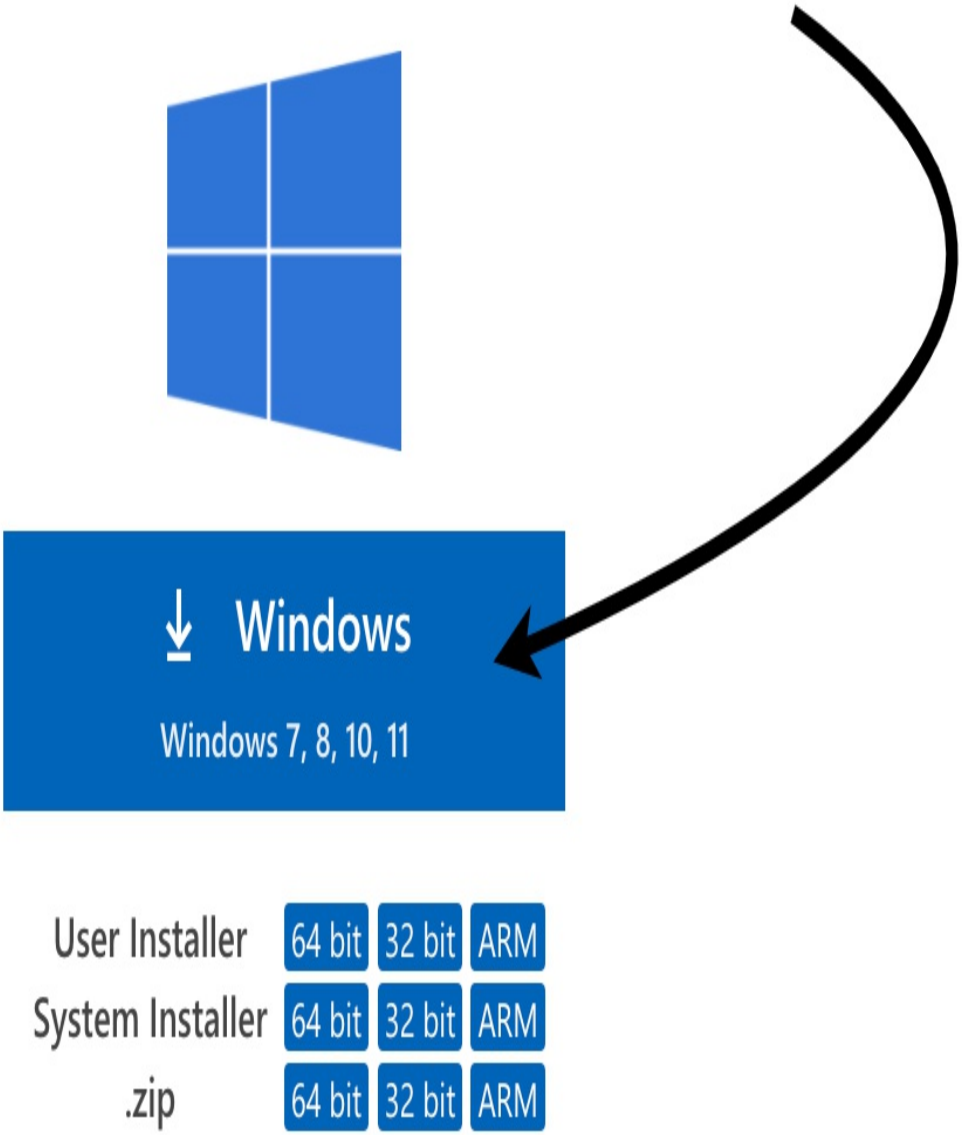
For more information on installing VSCode on Mac computers read more at https://code.visualstudio.com/docs/setup/mac.

A.2.2 Windows installation

If you are using a Windows computer running Windows 7, 8, 10, or 11, you may click on the blue button under the Windows logo as shown in figure A.3. Clicking this button in your web browser will download the VS Code installation EXE file with everything you need to get started.

Figure A.3. VSCode installation for Windows

Click on the large blue install button under the Windows logo



Once the EXE file downloads to your web browser's specified download location (this could be your downloads folder or desktop), double-click the EXE file to run the program installer. Once installed you will find VS Code in `C:\Users{Username}\AppData\Local\Programs\Microsoft VS Code`. Double-click the VS Code icon from this location to open the program.

 **Note**

You will be working largely from your command line. VS Code automatically configures your command line to open VS Code using the `code` keyword. Now you may restart your command line and simply type `code` and press **Enter** to open VS Code from your command line window.

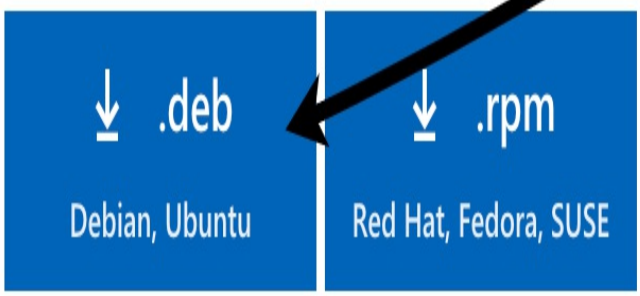
For more information on installing VSCode on Windows computers read more at https://code.visualstudio.com/docs/setup/windows.

A.2.3 Linux installation

If you are using a Linux computer running Debian or Ubuntu, you may click on the blue button under the Linux logo as shown in figure A.4. Clicking this button in your web browser will download the VS Code installation DEB file with everything you need to get started.

Figure A.4. VSCode installation for Linux

Click on the large blue install button under the Linux logo



.deb	64 bit	ARM	ARM 64
.rpm	64 bit	ARM	ARM 64
.tar.gz	64 bit	ARM	ARM 64

Snap Store

Once the DEB file downloads to your web browser's specified download location (this could be your downloads folder or desktop), double-click the DEB file to run the program installer. Once installed, double-click the VS Code icon from this location to open the program.

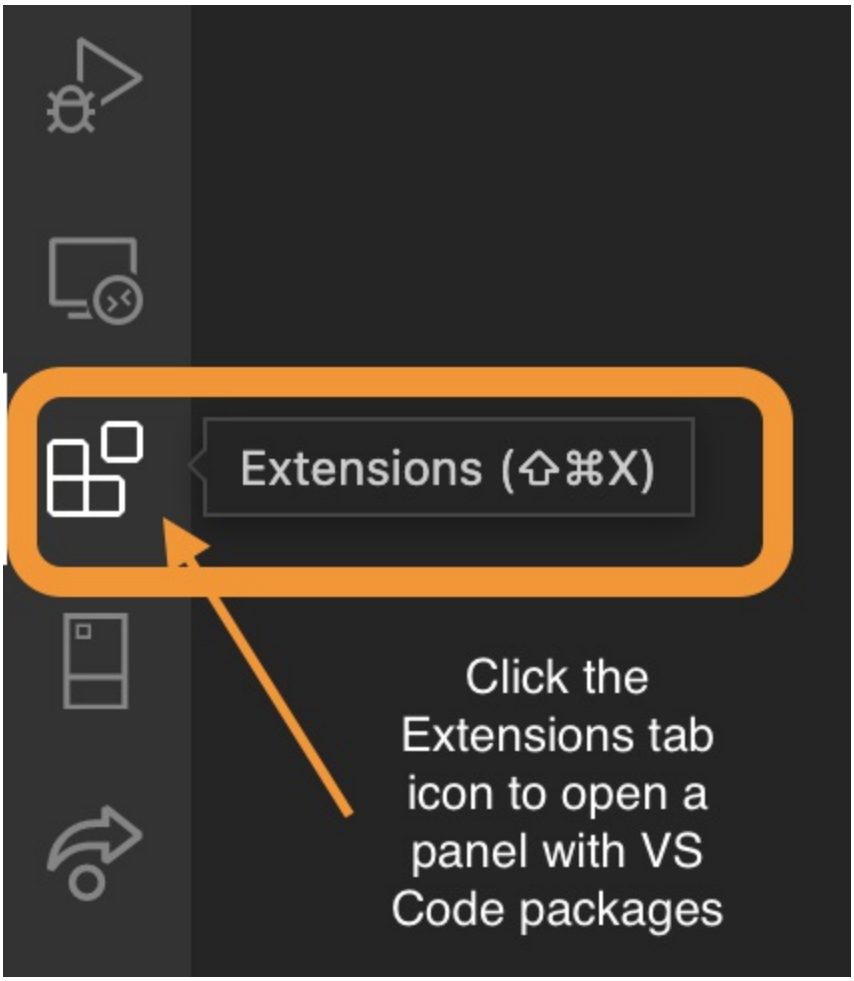
For more information on installing VSCode on Linux computers read more at https://code.visualstudio.com/docs/setup/linux.

A.2.4 More on VS Code

Throughout this book, you'll have the opportunity to develop for both the backend and frontend of your Node applications. You may choose another code editor or integrated development environment (IDE) such as Atom, IntelliJ, or Eclipse. These alternatives will certainly allow you to develop an application with Node, but you may find that they are not as supportive for web development as VS Code is in 2022.

VS Code runs on Electron (which is built with Node). The program as a whole is a lightweight editor capable of anything from basic text manipulation to running your full stack application. Unlike popular IDEs, VS Code does not fully support file linking (where you can click on a class name to display its associated file) out of the box. However, VS Code offers a community-driven library of packages (similar to Node NPM packages), called the VS Code marketplace, to install custom tooling to support your development. This marketplace is accessible within VS Code by clicking the `Extensions` navigation tab shown in figure A.5.

Figure A.5. Navigating to the extensions panel in VS Code



From here, you can search for extensions that can help make coding easier for you without adding unnecessary bloat to the editor. Within the `Extensions` panel, search for the extension name, or any keyword that may be related to your work. From the list of results, select the extension you want to use and click the install button, as seen in figure A.6.

Figure A.6. Installing extensions in VS Code

To get you started, here are some of the extensions you'll want to install and configure before you code anything:

- `JavaScript (ES6) code snippets`—This popular JavaScript extension will offer snippets of ES6 syntax code that are commonly used. You may save a lot of time wasted on retyping functions and statements. To read more on this extension visit https://marketplace.visualstudio.com/items?itemName=xabikos.JavaScriptSnippets.
- `Prettier – Code formatter`—This extension comes bundled with formatting support for anything from HTML to JavaScript classes and JSON structures. To read more on this extension visit https://marketplace.visualstudio.com/items?itemName=esbenp.prettier-

vscode.

- `npm`—This extension supports many of the NPM commands you'll mostly use on the command line. Having these commands accessible from within your code editor will help significantly. To read more on this extension visit https://marketplace.visualstudio.com/items?itemName=eg2.vscode-npm-script.
- `npm Intellisense`—This extension watches your `node_modules` folder and any global NPM packages you use in your project. From there, it offers tab-to-autocomplete for npm package names it recognizes in your application. To read more on this extension visit https://marketplace.visualstudio.com/items?itemName=christian-kohler.npm-intellisense.
- `ESLint`—This extension is a popular JavaScript linter tool. It will help keep your code formatted and bug-free. From semicolons to indentation, ESLint rules will help you understand why your code isn't working as expected. To read more on this extension visit https://marketplace.visualstudio.com/items?itemName=dbaeumer.vscode-eslint.

VS Code is constantly being improved to offer more support for developers. I recommend keeping your version of VS Code up to date, and to take advantage of some of the next generation extensions that are bound to be released. For more information about the VS Code Marketplace visit https://marketplace.visualstudio.com/vscode.

A.3 Installing Node.js

The focus of this book is to build applications using Node. As such, having Node.js properly installed on your computer is a crucial step to getting started. At the time of writing, Node v16.14.2 is the recommended Long Term Support (LTS) version to install.

Tip

When in doubt, try to install Node v16.14.2 specifically. It is unlikely that future versions of Node will break compatibility with this book's projects,

but if all installation versions match those used in the book, you should have no obstacles in the way of development.

Installing Node means you are installing a stand-alone JavaScript runtime environment on your computer. By the end of the process you will be able to run JavaScript commands in your command line window and run JavaScript-based applications without the need for a web browser.

All of the installation instructions in this section can be found at https://nodejs.org/en.

 **Note**

Any version after v14 should be ok to use with modern ES6 syntax.

To install Node, visit https://nodejs.org/en/download/ in your web browser. There you will see a page similar to the one in figure A.7. This page shows you all of the recommended installation options for the latest and LTS versions of Node across Mac, Windows, and Linux operating systems.

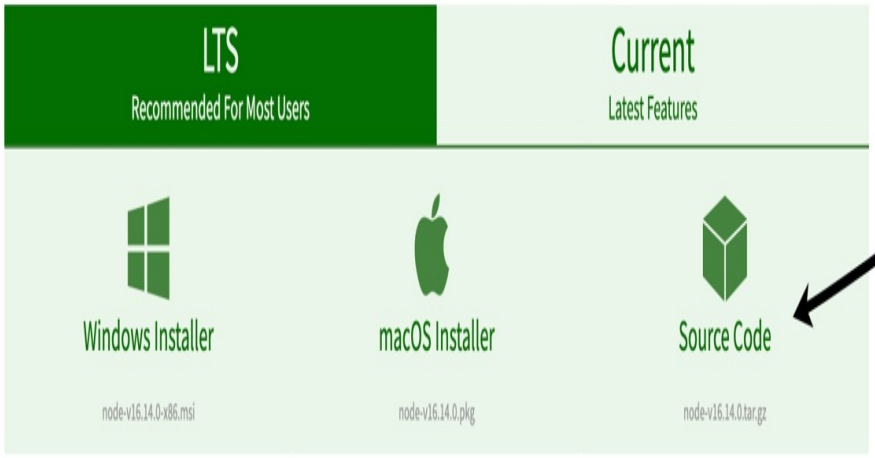
Figure A.7. Node installation page

Downloads

Latest LTS Version: **16.14.0** (includes npm 8.3.1)

Download the Node.js source code or a pre-built installer for your platform, and start developing today.

Click on your computer's corresponding Node.js installer.



Windows Installer (.msi)	32-bit	64-bit
Windows Binary (.zip)	32-bit	64-bit
macOS Installer (.pkg)	64-bit / ARM64	
macOS Binary (.tar.gz)	64-bit	ARM64
Linux Binaries (x64)	64-bit	
Linux Binaries (ARM)	ARMv7	ARMv8
Source Code	node-v16.14.0.tar.gz	

Next, you may follow your computer's operating system-specific instructions to complete the installation.

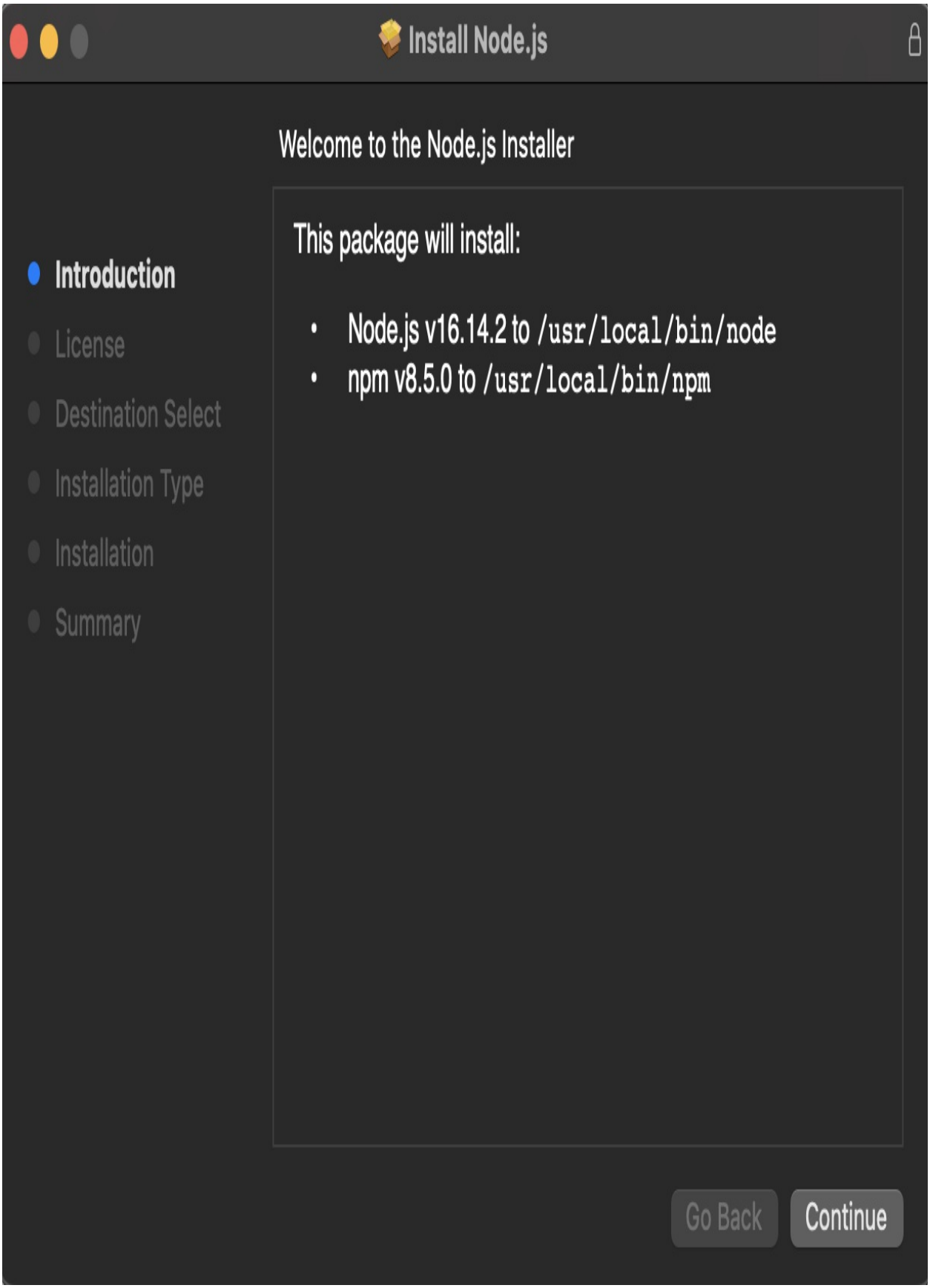
A.3.1 Mac installation

If you are using an Apple computer running macOS 10 or higher, you may click on the green button under the Apple logo as shown in figure A.8. Clicking this button in your web browser will download the VS Code installation PKG file with everything you need to get started.

Figure A.8. Node installation GUI button for Mac

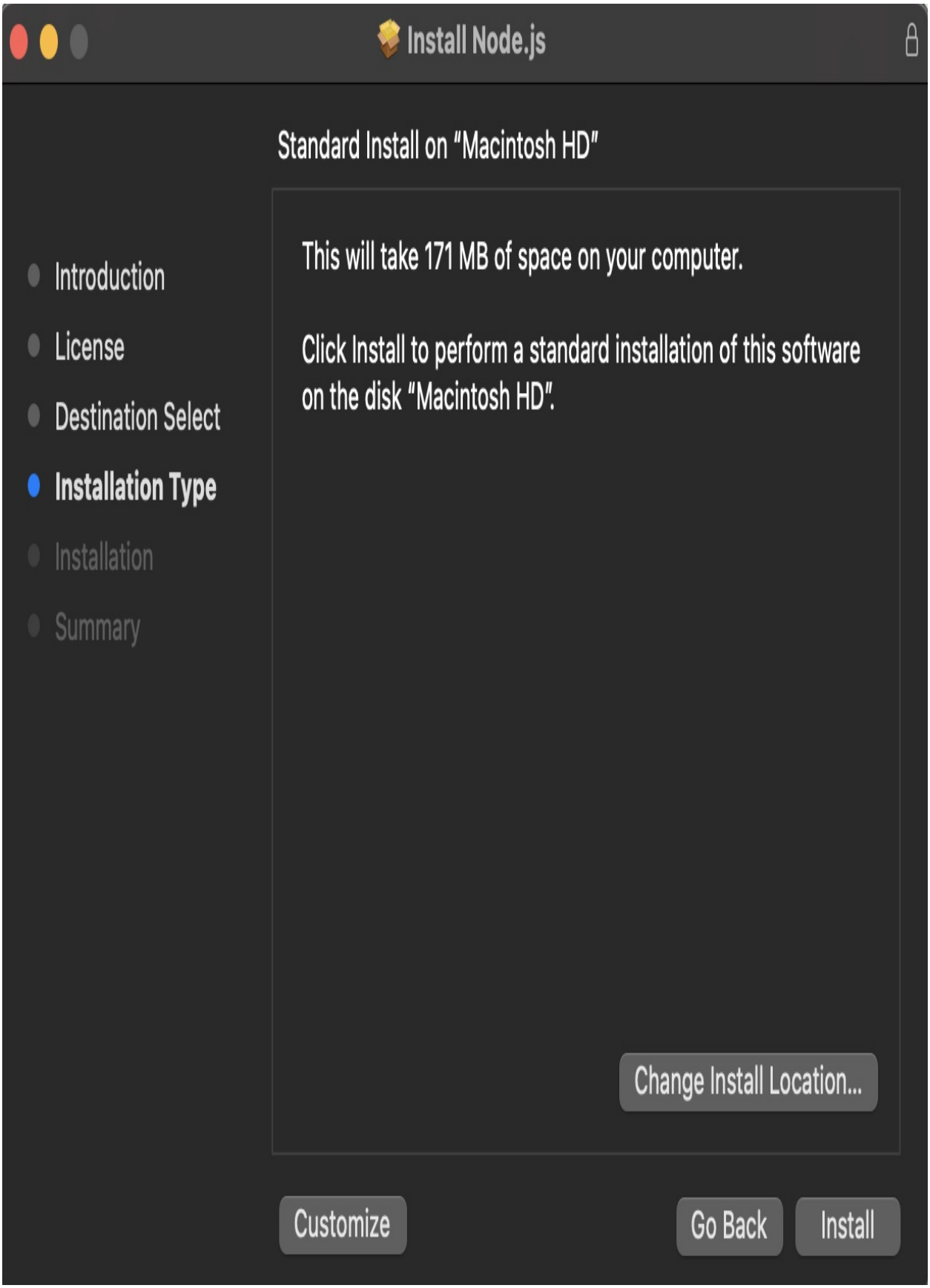
Once the PKG file downloads to your web browser's specified download location (this could be your downloads folder or desktop), double-click the PKG file to decompress its contents. Once the contents are decompressed you will see an installer window appear (figure A.9).

Figure A.9. Mac installer window show the introduction


Install Node.js
Welcome to the Node.js Installer
Introduction
License
Destination Select
Installation Type
Installation
Summary
This package will install:
• Node.js v16.14.2 to /usr/local/bin/node
• npm v8.5.0 to /usr/local/bin/npm
Go Back
Continue

Follow the steps in the installer by pressing continue and accepting the licensing agreement. You will then be instructed on where Node and NPM are going to be installed on your computer. Generally this Node will be installed at `usr/local/bin/node` as shown in figure A.10.

Figure A.10. Mac installer window show the introduction


Install Node.js
Standard Install on "Macintosh HD"
Introduction
License
Destination Select
Installation Type
Installation
Summary
This will take 171 MB of space on your computer.
Click Install to perform a standard installation of this software on the disk "Macintosh HD".
Change Install Location...
Customize
Go Back
Install

After completing these steps, Node and NPM are installed on your computer. You can test this by opening your command line and running `node`. The response should be a prompt to type in JavaScript. This is your Node REPL (Read-Eval_Print-Loop) environment. Try typing `console.log("Hello Tiny Projects!");` and pressing the enter key to run your first line of JavaScript in Node.

 **Note**

You may need to restart your computer to avoid any issues with your operating system PATH identifying Node for you.

For more information on installing Node on Mac computers read more at https://nodejs.org/en/download/package-manager.

A.3.2 Windows installation

If you are using a Windows computer running Windows 7 or higher, you may click on the blue button under the Windows logo as shown in figure A.11. Clicking this button in your web browser will download the VS Code installation MSI file with everything you need to get started.

Figure A.11. Node installation GUI button for Windows

Once the MSI file downloads to your web browser's specified download location (this could be your downloads folder or desktop), double-click the PKG file to decompress its contents. Once the contents are decompressed you will see an installer window appear.

Allow your computer to start the installation by pressing the `Run` button. You can then click `Next` through the installation wizard until you reach the main install page. Then, simply click `Install` and Node will install to your computer. When the installation completes, click `Finish` to close the installer window.

If the installation steps succeeded, Node and NPM are installed on your computer. You can test this by opening your command line and running `node`. The response should be a prompt to type in JavaScript. This is your Node REPL (Read-Eval_Print-Loop) environment. Try typing `console.log("Hello Tiny Projects!");` and pressing the enter key to run your first line of JavaScript in Node.

 **Note**

You may need to restart your computer to avoid any issues with your operating system PATH identifying Node for you.

For more information on installing Node on Windows computers read more at https://docs.microsoft.com/en-us/windows/dev-environment/javascript/nodejs-on-windows.

A.3.3 Linux installation

If you are using a Linux computer running Debian or Ubuntu, you may install Node by opening your command line and running `sudo apt install nodejs`.

After completing these steps, Node and NPM are installed on your computer. You can test this by opening your command line and running `node`. The response should be a prompt to type in JavaScript. This is your Node REPL (Read-Eval_Print-Loop) environment. Try typing `console.log("Hello Tiny`

`Projects!");` and pressing the enter key to run your first line of JavaScript in Node.

 **Note**

You may need to restart your computer to avoid any issues with your operating system PATH identifying Node for you.

For more information on installing Node on Linux computers read more at https://www.geeksforgeeks.org/installation-of-node-js-on-linux.

A.4 Installing MongoDB

For some of the chapters in this book you'll be using MongoDB (Mongo). Mongo is a NoSQL database, meaning it does not save data in a traditional tabular format. Instead, data is stored in collections, which in turn contain documents. One way to think of this storage structure is like a filing cabinet. Each cabinet is a Mongo database, each drawer is a collection, and each folder is a document. A document may contain only one paragraph of information or 50,000 pages of information.

Like a SQL, or relational database, you may still associate data. For example, you may store data for a customer and associate them with multiple purchase of several different products. The main difference with SQL databases is in how the data is stored and retrieved.

The following sections will describe how you may install Mongo on your Mac, Windows, or Linux computers.

A.4.1 Mac installation

There are multiple ways to install Mongo on your Mac. As of late, a recommended approach is to first install Homebrew (a package manager for tools and software). Next you'll find steps for installing Homebrew, and then subsequent steps to get Mongo running.

To install Hombrew, open your command line and run `/bin/bash -c "$(curl -fsSL raw.githubusercontent.com/Homebrew/install/HEAD/install.sh)"`. This will fetch Homebrew from the public Github repository and begin the installation process on your computer. This may take a few minutes to install.

Once Hombrew finishes installing you may begin installing Mongo.

 **Note**

You may also need to install Xcode tools, which don't come preinstalled with your Mac. Unless otherwise prompted, run `xcode-select --install` in any command line window to install these tools.

First, run `brew tap mongodb/brew`. This command will download the Mongo Database Tools needed to assist with your instance of Mongo on your computer. Then, run `brew install mongodb-community@5.0` which installs the actual Mongo database management system on your computer. This community version allows you to work with Mongo on all of your projects for free and without restriction.

When these installations complete you'll be able to run `mongod` to start your Mongo server.

 **Note**

In order for Mongo to save your data properly you'll need to set up a `db` directory. On Macs with Intel processors, that location is `/usr/local/var/mongodb` and for the newer M1 models the location is `/opt/homebrew/var/mongodb`. Make sure to create a `db` folder at these locations.

Alternative to the `mongod` command, you may also start Mongo through Hombrew. If you have not yet started your Mongo server, run `brew services start mongodb-community@5.0` to initiate the database server. Later, to stop this service run `brew services stop mongodb-community@5.0` in your

command line.

 **Tip**

To verify that Mongo is running on your computer with Hombrew, run `brew services list` in your command line window. You'll see a list of services currently running.

`visit `https://brew.sh/`[brew.sh/]. Here you'll find instructions for https://docs.mongodb.com/manual/tutorial/install-mongodb-on-os-x/

A.4.2 Windows installation

https://docs.mongodb.com/manual/tutorial/install-mongodb-on-windows/

A.4.3 Linux installation

Mac and Linux computers have fairly similar steps for installing Mongo. Aside from following the Homebrew instructions in the Mac section, you may also install Mongo through your command line.

Open a command line window and run `www.mongodb.com/docs/manual/tutorial/install-mongodb-on-ubuntu/` to import the MongoDB public GPG Key. When complete you should see an `OK` response. Next you need to create the list file for your version of Linux at `/etc/apt/sources.list.d/mongodb-org-5.0.list`. For Ubuntu v20.04 you would run `echo "deb [ arch=amd64,arm64 ] repo.mongodb.org/apt/ubuntu focal/mongodb-org/5.0 multiverse" | sudo tee /etc/apt/sources.list.d/mongodb-org-5.0.list`.

Next, reload your local packages by running `sudo apt-get update`. You may now install the latest stable version of Mongo by running `sudo apt-get install -y mongodb-org`.

After this last command, you should have Mongo installed on your computer. You may now start the database server by running `sudo systemctl start mongod` in your command line.

 **Note**

If you run into an error starting Mongo, try running `sudo systemctl daemon-reload`

https://docs.mongodb.com/manual/administration/install-on-linux/

Appendix B. Setting up Node app essentials

In this appendix you'll

- Set up packages useful for developing Node apps
- Install packages that can be used with any Node APIs

If you're reading this appendix, it's probably because you've begun the app development process in one of the book's chapters. By now you should have an application folder, including `package.json` and `index.js` files. With these two files in place, you can add additional configurations, package dependencies, and app scripts to your project.

Follow the recommended steps in the next section to ensure your app will run with the book's suggested Node version and JavaScript syntax. From there, you may also install the optional packages, which help with cleaning up and organizing your code and development environment.

B.1 Setting up project essentials

Now that you have a `package.json` file you'll need to add a configuration line to support ECMAScript (ES) modules. To do that, add `"type":"module",` as a key-value pair in your `package.json` file.

 **Note**

ES defines the standard for JavaScript. One of those standards is using modules to package JavaScript. Most Node programmers choose between one of two ways to import external modules. The CommonJS syntax offers the `require` keyword for importing JavaScript packages outside of a web browser (for more information visit

https://requirejs.org/docs/commonjs.html). Alternatively, Node v18+ has offered support for the standard `import` and `export` statements when working with modules. You can learn more about ES modules in Node at https://nodejs.org/api/esm.html.

Because

```
javascript
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1",
    "start": "nodemon index.js",
  },
```

Appendix C. Node under the hood

In this appendix you'll

- How the Node event loop operates
- What modules come prepackaged with Node
- Why Node is preferred for certain apps
    - When JavaScript was created
    - Life as a browser language
    - Simple operations allowed JavaScript to operate off of a single thread
    - More complex server applications made use of CPU, cores, and threads to run multiple operations at the same time
    - Explain difference in callout: https://www.guru99.com/cpu-core-multicore-thread.html#:~:text=KEY%20DIFFERENCE&text=Cores%20is%2
    - Under stand how a computer works first. A computer's fundamental software is the operating system. Most of what's processed in a computer is done so through the computer's Central Processing Unit (CPU) Core. A computer's CPU core is like the brain of the computer. While humans can multitask in theory, our brains can only process a single thought at a time. Likewise, computer CPU cores generally process a single task at a time.
    - Diagram of person thinking about taking out the trash while computer garbage collects, plays a video, and monitors a security camera -Nowadays, computers have multiple cores, effectively allowing the computer to use multiple brains at the same time to process a greater factor of tasks at the same time. Mutlicore processors support a computer's ability to more rapidly handle large tasks like video rendering, or unrelated tasks like calculating a large equation and running computationally expensive software (the ones that get your fans spinning). This is called concurrent, or parallel, programming. In fact, a CPU core can further break down a task by processing multiple parts of the task in smaller units of

execution called threads. In multithreading programming, a CPU can execute multiple parts of a process at the same time to reduce the latency before a completed state. So, if you are querying a database for billions of records, you may find it faster to break the query down into four parts that each execute on their own thread, in parallel. That's four times faster than on one thread. What could take four hours to computer, only takes one hour. So, between CPU cores, their processes, and multiple threads, modern computers have the ability to handle large tasks in exponentially shorter periods of time. So, what don't we always use multiple threads? There are tradeoffs. For one thing, the more you process in parallel, the more power is used, which can put a strain on your computer. Also, threads may share resources, which means they may all have access to the same data at the same time. A common side effect of this relationship is a condition called deadlock, where two or more threads are waiting for each other to complete a task or use a resource. The result is a blocked application where no operation can be completed. With more capable and complex hardware come complex scenarios and edge cases to account for. Enter JavaScript.

- JavaScript was designed with a single call stack that uses a single thread only to processes its tasks. This architecture is simpler to implement and avoids the downsides of a multithreaded system.
- ***** More on how we benefit from this architecture
- Side note: While JavaScript can theoretically offload some of its tasks to the computer's threads, the main JavaScript process still runs on a single thread.
- Because JavaScript runs on a single thread it's architecture allows for some functions to execute when ready and others to execute a callback eventually
- SHOW event loop diagram
- See how callback queue allows the loop to handle functions when they are ready
- This was fine until callbacks got out of hand and created callback hell
- What came next were Promises, the Promise was incorporated into the standard ECMAScript JavaScript versions, allowing eventual return ofa value from an inner function. Also allowed simpler

    syntax
    - The syntax improved even more with async await, which effectively wraps a promise response
    - So, does an azsync await call block the event loop?

- Libuv
- fs and readline modules, etc.
- fast
- Javascript on server
- Growing community

- What is Nodejs
    - The Platform
    - built in modules

- Synchronous vs Async
    - What’s a callback
    - What’s a promise

- Event Loop
    - Why a single thread?
    - https://2ality.com/2022/09/nodejs-overview.html#why-does-node.js-code-run-in-a-single-thread%3F
    - Worker threads