sitepoint

BUILD YOUR FIRST NODE APP

COMPLETE PROJECT TUTORIAL

Build Your First Node App

- **Author:** James Hibbard
- **Cover Designer:** Alex Walker

Notice of Rights

Notice of Liability

Trademark Notice

Published by SitePoint Pty. Ltd.

48 Cambridge Street Collingwood
VIC Australia 3066
Web: www.sitepoint.com
Email: books@sitepoint.com

About SitePoint

SitePoint specializes in publishing fun, practical, and easy-to-understand content for web professionals. Visit http://www.sitepoint.com/ to access our blogs, books, newsletters, articles, and community forums. You'll find a stack of information on JavaScript, PHP, Ruby, mobile development, design, and more.

Table of Contents

Preface

Conventions Used

You'll notice that we've used certain typographic and layout styles throughout this book to signify different types of information. Look out for the following items.

Code Samples

Code in this book is displayed using a fixed-width font, like so:

```
<h1>A Perfect Summer's Day</h1>
<p>It was a lovely day for a walk in the park.
The birds were singing and the kids were all back at school.</p>
```

Where existing code is required for context, rather than repeat all of it, ⋮ will be displayed:

```
function animate() {
  ⋮
new_variable = "Hello";
}
```

Some lines of code should be entered on one line, but we've had to wrap them because of page constraints. An ↪ indicates a line break that exists for formatting purposes only, and should be ignored:

```
URL.open("http://www.sitepoint.com/responsive-web-
↪design-real-user-testing/?responsive1");
```

Tips, Notes, and Warnings

Hey, You!

Tips provide helpful little pointers.

Ahem, Excuse Me ...

Notes are useful asides that are related—but not critical—to the topic at hand. Think of them as extra tidbits of information.

Make Sure You Always ...

... pay attention to these important points.

Watch Out!

Warnings highlight any gotchas that are likely to trip you up along the way.

Live Code

This example has a Live Codepen.io Demo you can play with.

Github

This example has a code repository available at Github.com.

Chapter

1

What Is Node.js?

There are plenty of definitions to be found online. Let's take a look at a couple of the more popular ones:

This is what the project's home page has to say:

> *Node.js® is a JavaScript runtime built on Chrome's V8 JavaScript engine. Node.js uses an event-driven, non-blocking I/O model that makes it lightweight and efficient.*

And this is what StackOverflow has to offer:

> *Node.js is an event-based, non-blocking, asynchronous I/O framework that uses Google's V8 JavaScript engine and libuv library.*

Hmmm, "non-blocking I/O", "event-driven", "asynchronous" — that's quite a lot to digest in one go. So let's approach this from a different angle and begin by focusing on the other detail that both descriptions mention — the V8 JavaScript engine.

Node Is Built on Google Chrome's V8 JavaScript Engine

The V8 engine is the open-source JavaScript engine that runs in the Chrome, Opera and Vivaldi browsers. It was designed with performance in mind and is responsible for compiling JavaScript directly to native machine code that your computer can execute.

However, when we say that Node is built on the V8 engine, we don't mean that Node programs are executed in a browser. They aren't. Rather, the creator of Node (Ryan Dahl) took the V8 engine and enhanced it with various features, such

as a file system API, an HTTP library, and a number of operating system–related utility methods.

This means that Node.js is a program we can use to execute JavaScript on our computers. In other words, it's a JavaScript runtime.

How Do I Install Node.js?

In this next section, we'll install Node and say hello (world). We'll also look at npm, a package manager that comes bundled with Node.

Node Binaries vs Version Manager

Many websites will recommend that you head to the official Node download page and grab the Node binaries for your system. While that works, I would suggest that you use a version manager instead. This is a program which allows you to install multiple versions of Node and switch between them at will. There are various advantages to using a version manager. For example, it negates potential permission issues which would otherwise see you installing packages with admin permissions.

If you fancy going the version manager route, please consult our quick tip: Install Multiple Versions of Node.js using nvm. Otherwise, grab the correct binaries for your system from the link above and install those.

"Hello, World!" the Node.js Way

You can check that Node is installed on your system by opening a terminal and typing `node -v`. If all has gone well, you should see something like `v8.9.4` displayed. This is the current LTS version at the time of writing.

Next, create a new file `hello.js` and copy in the following code:

```
console.log("Hello, World!");
```

This uses Node's built-in console module to display a message in a terminal window. To run the example, type the following command:

```
node hello.js
```

If Node.js is configured properly, "Hello, World!" will be displayed.

Chapter

2

What Is Node.js Used For?

Now that we know what Node and npm are and how to install them, we can turn our attention to the first of their common uses: they're used to install (npm) and run (Node) various build tools — tools designed to automate the process of developing a modern JavaScript application.

These build tools come in all shapes and sizes, and you won't get far in a modern JavaScript landscape without bumping into them. They can be used for anything from bundling your JavaScript files and dependencies into static assets, to running tests, or automatic code linting and style checking.

We have a wide range of articles covering build tooling on SitePoint. Here's a short selection of my favorites:

- A Beginner's Guide to Webpack and Module Bundling
- How to Bundle a Simple Static Site Using Webpack
- Up and Running with ESLint — the Pluggable JavaScript Linter
- An Introduction to Gulp.js
- Unit Test Your JavaScript Using Mocha and Chai

And if you want to start developing apps with any modern JavaScript framework (for example, React or Angular), you'll be expected to have a working knowledge of Node and npm (or maybe Yarn). This is not because you need a Node backend to run these frameworks. You don't. Rather, it's because these frameworks (and many, many related packages) are all available via npm and rely on Node to create a sensible development environment in which they can run.

If you're interested in finding out what role Node plays in a modern JavaScript app, read The Anatomy of a Modern JavaScript Application.

Node.js Lets Us Run JavaScript on the Server

Next we come to one of the biggest use cases for Node.js — running JavaScript on the server. This is not a new concept, and was first attempted by Netscape way back in 1994. Node.js, however, is the first implementation to gain any real

traction, and it provides some unique benefits, compared to traditional languages. Node now plays a critical role in the technology stack of many high-profile companies. Let's have a look at what those benefits are.

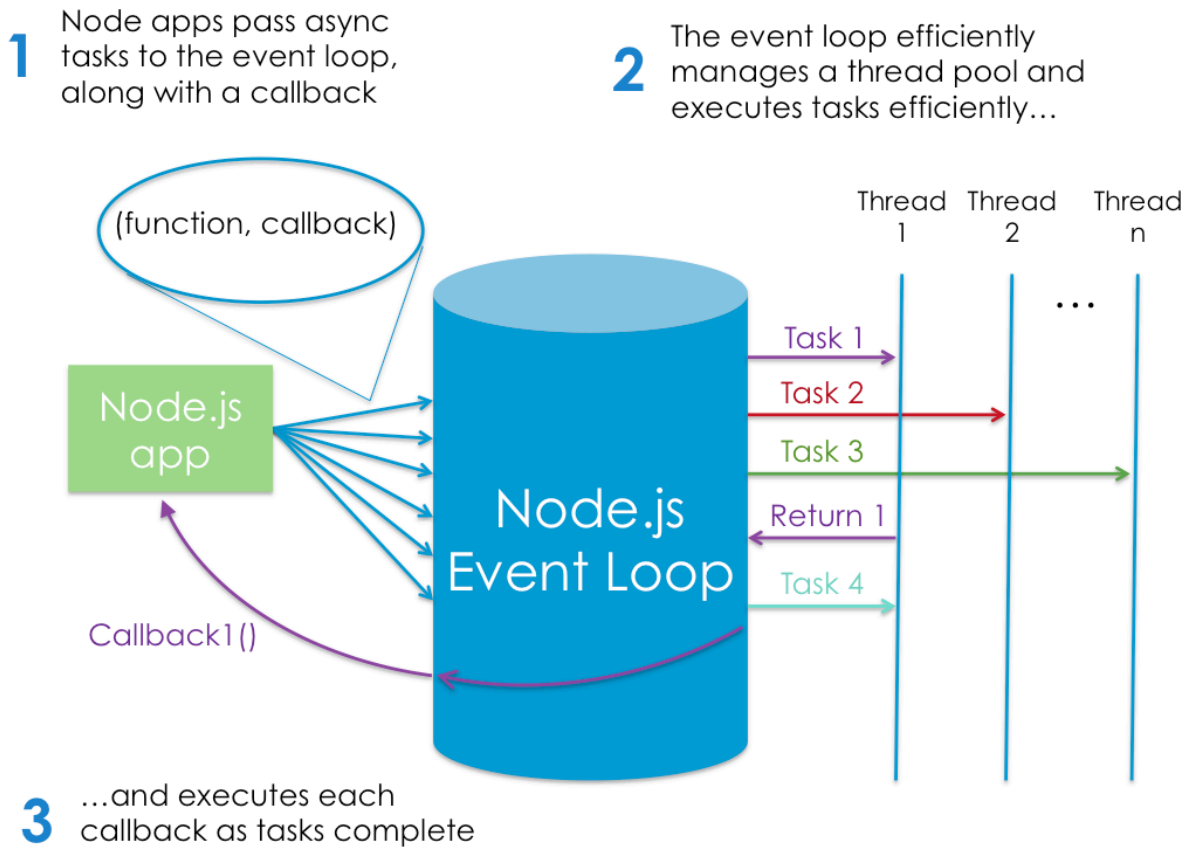
The Node.js Execution model

In very simplistic terms, when you connect to a traditional server, such as Apache, it will spawn a new thread to handle the request. In a language such as PHP or Ruby, any subsequent I/O operations (for example, interacting with a database) block the execution of your code until the operation has completed. That is, the server has to wait for the database lookup to complete before it can move on to processing the result. If new requests come in while this is happening, the server will spawn new threads to deal with them. This is potentially inefficient, as a large number of threads can cause a system to become sluggish — and, in the worse case, for the site to go down. The most common way to support more connections is to add more servers.

Node.js, however, is single-threaded. It is also event-driven, which means that everything that happens in Node is in reaction to an event. For example, when a new request comes in (one kind of event) the server will start processing it. If it then encounters a blocking I/O operation, instead of waiting for this to complete, it will register a callback before continuing to process the next event. When the I/O operation has finished (another kind of event), the server will execute the callback and continue working on the original request. Under the hood, Node uses the libuv library to implement this asynchronous (i.e. non-blocking) behavior.

Node's execution model causes the server very little overhead, and consequently it's capable of handling a large number of simultaneous connections. The traditional approach to scaling a Node app is to clone it and have the cloned instances share the workload. Node.js even has a built-in module to help you implement a cloning strategy on a single server.

The following image depicts Node's execution model:



2-1. Node's execution model

Source: Introduction To Node.js by Prof. Christian Maderazo, James Santos

Are There Any Downsides?

The fact that Node runs in a single thread does impose some limitations. For example, blocking I/O calls should be avoided, and errors should always be handled correctly for fear of crashing the entire process. Some developers also dislike the callback-based style of coding that JavaScript imposes (so much so that there is even a site dedicated to the horrors of writing asynchronous JavaScript). But with the arrival of native Promises, followed closely by async await (which is enabled by default as of Node version 7.6), this is rapidly becoming a thing of the past.

“Hello, World!” — Server Version

Let’s have a quick look at a “Hello, World!” example HTTP server.

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

http.createServer((request, response) => {
  response.writeHead(200);
  response.end('Hello, World!');
}).listen(3000);

console.log('Server running on http://localhost:3000');
```

To run this, copy the code into a file named `hello-world-server.js` and run it using `node hello-world-server.js`. Open up a browser and navigate to http://localhost:3000 to see “Hello, World!” displayed in the browser.

Now let’s have a look at the code.

We start by requiring Node’s native HTTP module. We then use its createServer method to create a new web server object, to which we pass an anonymous function. This function will be invoked for every new connection that is made to the server.

The anonymous function is called with two arguments (`request` and `response`) which contain the request from the user and the response, which we use to send back a 200 HTTP status code, along with our “Hello World!” message.

Finally, we tell the server to listen for incoming requests on port 3000, and output a message to the terminal to let us know it’s running.

Obviously, there’s lots more to creating even a simple server in Node (for example, it is important to handle errors correctly), so I’d advise you to check the documentation if you’d like to find out more.

Chapter

Laying the Foundations of Our App

3

Today, we're going to build the basic functionality of our note-taking app using the MVC architecture. As I mentioned earlier, we're going to employ the Hapi.js framework for Node.js and SQLite as a database, using Sequelize.js, plus other small utilities to speed up our development. We're going to build the views using Pug, the templating language.

What is MVC?

Model-View-Controller (or MVC) is probably one of the most popular architectures for applications. As with a lot of other cool things in computer history, the MVC model was conceived at PARC for the Smalltalk language as a solution to the problem of organizing applications with graphical user interfaces. It was created for desktop applications, but since then, the idea has been adapted to other mediums including the web.

We can describe the MVC architecture in simple words:

- **Model**: The part of our application that will deal with the database or any data-related functionality.
- **View**: Everything the user will see. Basically the pages that we're going to send to the client.
- **Controller**: The logic of our site, and the glue between models and views. Here we call our models to get the data, then we put that data on our views to be sent to the users.

Our application will allow us to publish, view, edit and delete plain-text notes. It won't have other functionality, but because we'll have a solid architecture already defined, we won't have big trouble adding things later.

Github Repository

You can check out the final application in the accompanying GitHub repository, so you get a general overview of the application structure.

Laying out the Foundation

The first step when building any Node.js application is to create a *package.json* file, which is going to contain all of our dependencies and scripts. Instead of creating this file manually, npm can do the job for us using the `init` command:

```
npm init -y
```

After the process is complete will get a *package.json* file ready to use.

npm?

If you're not familiar with these commands, check out our Beginner's Guide to npm.

We're going to proceed to install Hapi.js — the framework of choice for this tutorial. It provides a good balance between simplicity, stability and feature availability that will work well for our use case (although there are other options that would also work just fine).

```
npm install --save hapi
```

This command will download the latest version of Hapi.js and add it to our *package.json* file as a dependency.

Now we can create our entry file — the web server that will start everything. Go ahead and create a *server.js* file in your application directory and all the following code to it:

```
const Hapi = require('hapi');
const Settings = require('./settings');
const server = new Hapi.Server({ port: Settings.port });
```

```
server.route({
  method: 'GET',
  path: '/',
  handler: (request, h) => 'Hello, World!'
});

const init = async () => {
  await server.start();
  console.log(`Server running at: ${server.info.uri}`);
};

process.on('unhandledRejection', (err) => {
  console.log(err);
  process.exit(1);
});

init();
```

This is going to be the foundation of our application.

First, we include our dependencies and instantiate a new server object where we set the connection port to whatever we declare in our settings file. We will use port 3000 although this can be any number above *1023* and below *65535*.

The first route for our server will work as a test to see if everything is working, so a “Hello, world!” message is enough for us.

In each route, we have to define the HTTP method and path (URL) that it will respond to, as well as a handler, which is a function that will process the HTTP request. The handler function can take two parameters. The first, *request*, contains information about the HTTP call. The second is something called the *response toolkit*, an object containing properties and utilities for taking care of the response. This parameter is often shortened to *h*.

Next, we create an async function from within which we can start our server using the *server.start* method. Note that the latest version of Hapi (v17 at the

time of writing) has undergone something of a re-write and is now designed to be used with the new *async-await* syntax. We then log a message to the console that the server is running and include some basic error-handling code, before kicking everything off by calling the *init* method.

Storing Our Settings

It is good practice to store our configuration variables in a dedicated file. This file exports a JSON object containing our data, where each key is assigned from an environment variable — but without forgetting a fallback value.

In this file, we can also have different settings depending on our environment (e.g. development or production). For example, we can have an in-memory instance of SQLite for development purposes, but a real SQLite database file for production.

Selecting the settings depending on the current environment is quite simple. Since we also have an *env* variable in our file which will contain either *development* or *production*, we can do something like the following to get the database settings (for example):

```
const dbSettings = Settings[Settings.env].db;
```

So *dbSettings* will contain the setting of an in-memory database when the *env* variable is *development*, or will contain the path of a database file when the *env* variable is *production*.

Using a `.env` File

You can use the sqlite3 library to create a SQLite database, like so: *const db = new sqlite.Database('db/database.sqlite');*

Also, we can add support for a *.env* file, where we can store our environment

variables locally for development purposes; this is accomplished using a package like dotenv for Node.js, which will read a *.env* file from the root of our project and automatically add any values it finds to the environment. You can find a couple of examples of what a *.env* file should look like in this tutorial.

Using a `.env` File

If you decide to also use a `.env` file, make sure you install the package with `npm install -s dotenv` and add it to `.gitignore` so you don't publish any sensitive information.

Create a *settings.js* file in the project root and add the following code:

```
// Uncomment the next line to load our .env file and add the values to process.env
// require('dotenv').config({ silent: true });

module.exports = {
  port: process.env.PORT || 3000,
  env: process.env.ENV || 'development',

  // Environment-dependent settings
  development: {
    db: {
      dialect: 'sqlite',
      storage: ':memory:',
      operatorsAliases: false,
    },
  },
  production: {
    db: {
      dialect: 'sqlite',
      storage: 'db/database.sqlite',
      operatorsAliases: false,
    },
  },
};
```

Now we can start our application by executing the following command and navigating to *http://localhost:3000* in our web browser.

```
node server.js
```

You should be greeted by a "Hello, World!" message.

Check Your Version

This project was tested on the latest (at the time of writing) LTS version of Node (8.12.0). If you get any errors, ensure you have an updated installation.

Next, we'll take things further and take on the app's routes.

Chapter

Defining the Routes

4

The definition of routes gives us an overview of the functionality supported by our application. To create our additional routes, we just have to replicate the structure of the route that we already have in our *server.js* file, changing the content of each one.

Let's start by creating a new directory called *lib* in our project. Here we're going to include all the JS components. Inside *lib*, let's create a *routes.js* file and add the following content:

```
module.exports = [
  // We're going to define our routes here
];
```

In this file, we'll export an array of objects that contains each route of our application. To define the first route, add the following object to the array:

```
{
  method: 'GET',
  path: '/',
  handler: (request, h) => {
    return 'All the notes will appear here';
  },
  config: {
    description: 'Gets all the notes available'
  }
},
```

Our first route is for the home page (/) and since it will only return information, we assign it a GET method. For now, it will only give us the message *All the notes will appear here*, which we're going to change later for a controller function. The *description* field in the *config* section is only for documentation purposes.

Next, we can create the four routes for our notes under the */note/* path. Since we're building a CRUD application, we will need one route for each action with the corresponding HTTP method.

Add the following definitions below to the previous route:

```
{
  method: 'POST',
  path: '/note',
  handler: (request, h) => {
    return 'New note';
  },
  config: {
    description: 'Adds a new note'
  }
},
{
  method: 'GET',
  path: '/note/{slug}',
  handler: (request, h) => {
    return 'This is a note';
  },
  config: {
    description: 'Gets the content of a note'
  }
},
{
  method: 'PUT',
  path: '/note/{slug}',
  handler: (request, h) => {
    return 'Edit a note';
  },
  config: {
    description: 'Updates the selected note'
  }
},
{
  method: 'GET',
  path: '/note/{slug}/delete',
  handler: (request, h) => {
    return 'This note no longer exists';
  },
  config: {
    description: 'Deletes the selected note'
```

```
  }
},
```

We've done the same as in the previous route definition, but this time we've changed the method to match the action we want to execute.

The only exception is the delete route. In this case, we're going to define it with the *GET* method rather than *DELETE* and add an extra */delete* in the path. This way, we can call the delete action just by visiting the corresponding URL.

REST

If you plan to implement a strict REST interface, then you would have to use the `DELETE` method and remove the `/delete` part of the path.

We can name parameters in the path by surrounding the word in brackets (**{** ... **}**). Since we're going to identify notes by a slug, we add *{slug}* to each path, with the exception of the PUT route. We don't need it there, because we're not going to interact with a specific note, but to create one.

You can read more about Hapi.js routes in the official documentation.

Now, we have to add our new routes to the *server.js* file. Let's import the routes file at the top of the file:

```
const Routes = require('./lib/routes');
```

and replace our current test route with the following:

```
server.route(Routes);
```

To check that things are working, restart the server, then try visiting a couple of the new GET routes: http://localhost:3000/, http://localhost:3000/note/1, or

http://localhost:3000/note/1/delete

You should see the corresponding messages from the routes file we just defined.

Next, we'll build the app's models, which allow us to define the structure of the data and all the functions that work within it.

Chapter

Building the Models

5

Models allow us to define the structure of the data and all the functions to work with it.

In this example, we're going to use the SQLite database with Sequelize.js which is going to provide us with a better interface using the ORM (Object-Relational Mapping) technique. It will also provide us a database-independent interface.

Setting up the database

For this section, we're going to use the Sequelize.js and SQlite packages. You can install and include them as dependencies by executing the following command:

```
npm install -s sequelize sqlite3
```

Now create a *models* directory inside the *lib* directory and add a file called *index.js*. This is going to contain the database and Sequelize.js setup. Add the following content:

```
const Fs = require('fs');
const Path = require('path');
const Sequelize = require('sequelize');
const Settings = require('../../settings');

// Database settings for the current environment
const dbSettings = Settings[Settings.env].db;
const sequelize = new Sequelize(
  dbSettings.database, dbSettings.user, dbSettings.password, dbSettings
);

const db = {};

// Read all the files in this directory and import them as models
Fs.readdirSync(__dirname)
  .filter(file => (file.indexOf('.') !== 0) && (file !== 'index.js'))
  .forEach((file) => {
    const model = sequelize.import(Path.join(__dirname, file));
```

```
    db[model.name] = model;
  });

db.sequelize = sequelize;
db.Sequelize = Sequelize;

module.exports = db;
'.'
```

First, we include the modules that we're going to use:

- *Fs*, to read the files inside the *models* folder, which is going to contain all the models.
- *Path*, to join the path of each file in the current directory.
- *Sequelize*, that will allow us to create a new Sequelize instance.
- *Settings*, which contains the data of our *settings.js* file from the root of our project.

Next, we create a new *sequelize* variable that will contain a Sequelize instance with our database settings for the current environment. We're going to use *sequelize* to import all the models and make them available in our *db* object.

The *db* object is going to be exported and will contain our database methods for each model; it will be available in our application when we need to do something with our data.

To load all the models, instead of defining them manually, we look for all the files inside the *models* directory (with the exception of the *index.js* file) and load them using the *import* function. The returned object will provide us with the CRUD methods, which we then add to the *db* object.

At the end, we add *sequelize* and *Sequelize* properties to our *db* object. The first one is going to be used in our *server.js* file to connect to the database before starting the server, and the second one is included for convenience if you need it in other files too.

Creating our Note model

In this section, we're going to use the Moment.js package to help with date formatting. You can install it and include it as a dependency with the following command:

```
npm install -s moment
```

Now, we're going to create a *note.js* file inside the *models* directory, which is going to be the only model in our application; it will provide us with all the functionality we need.

Add the following content to that file:

```
const Moment = require('moment');

module.exports = (sequelize, DataTypes) => {
  const Note = sequelize.define('Note', {
    date: {
      type: DataTypes.DATE,
      get() {
        return Moment(this.getDataValue('date')).format('MMMM Do, YYYY');
      },
    },
    title: DataTypes.STRING,
    slug: DataTypes.STRING,
    description: DataTypes.STRING,
    content: DataTypes.STRING,
  });

  return Note;
};
```

We export a function that accepts a *sequelize* instance, to define the model, and a *DataTypes* object with all the types available in our database.

Next, we define the structure of our data using an object where each key

corresponds to a database column and the value of the key defines the type of data that we're going to store. You can see the list of data types in the Sequelize.js documentation. The tables in the database are going to be created automatically based on this information.

In the case of the *date* column, we also define a how Sequelize should return the value using a *getter* function (*get* key). We indicate that before returning the information it should first be passed through the Moment utility to be formatted in a more readable way (*MMMM Do, YYYY*).

Not Destructive

Although we're getting a simple and easy to read date string, it is stored as a precise date string product of the Date object of JavaScript. So this is not a destructive operation.

Finally, we return our model.

Now that we have our model pieces done, next we'll synchronize our database so we can use it in the app.

Chapter 6

Synchronizing the Database

Now, we have to synchronize our database before we're able to use it in our application. In `server.js`, import the models at the top of the file:

```
const Models = require('./lib/models/');
```

Next, wrap the following method call:

```
init();
```

with a call to *Models.sequelize.sync*, like so:

```
Models.sequelize.sync().then(() => {
  init();
});
```

This code is going to synchronize the models with our database and, once that is done, the *init* method will kick everything off. Doing things this way means we don't clutter the application while making use of Sequelize's features.

Building the controllers

Controllers are functions that accept the request object and the response toolkit from Hapi.js. The *request* object contains information about the requested resource and we use methods from the *response toolkit* to return information to the client.

In our application, we're going to return only a JSON object for now, but we will add the views once we build them.

We can think of controllers as functions that will join our models with our views; they will communicate with our models to get the data, and then return that data inside a view.

The Home Controller

The first controller that we're going to build will handle the home page of our site. Create a *lib/controllers* directory with a *home.js* file inside, then add the following content:

```
const Models = require('../models/');

module.exports = async (request, h) => {
  const notes = await Models.Note.findAll({
    order: [['date', 'DESC']],
  });

  return { notes };
};
```

First, we get all the notes in our database using the *findAll* method of our model. This function will return a promise which, when it resolves, will give us an array containing all the notes in our database.

We can arrange the results in descending order, using the *order* parameter in the options object passed to the *findAll* method, so the last item will appear first. You can check all the available options in the Sequelize.js documentation (see previous link).

Finally we return an object containing a *note* property, the value of which is our array of notes.

Once we have the home controller, we can edit our *routes.js* file. First, we import the module at the top of the file, along with the Path module:

```
const Path = require('path');
const Home = require('./controllers/home');
```

Then we add the controller we just made to the array:

```
{
  method: 'GET',
  path: '/',
  handler: Home,
  config: {
    description: 'Gets all the notes available'
  }
},
```

To make sure things are working properly, restart your server, then visit http://localhost:3000/. You should see the following response: `{"notes":[]}`

Next, we'll set up the boilerplate of a note controller.

Chapter

7

Boilerplate of the Note Controller

```
npm install -s slugify
```

The last controller that we have to define in our application will allow us to create, read, update, and delete notes.

We can proceed to create a *note.js* file inside the *lib/controllers* directory and add the following content:

```
const Path = require('path');
const Slugify = require('slugify');
const Models = require('../models/');

module.exports = {
  // Here we're going to include our functions that will handle each request
  // in the routes.js file.
};
```

The “create” function

To add a note to our database, we’re going to write a *create* function that is going to wrap the *create* method on our model using the data contained in the payload object.

Add the following inside the object that we’re exporting:

```
create: async (request, h) => {
  const note = await Models.Note
    .create({
      date: new Date(),
      title: request.payload.noteTitle,
      slug: Slugify(request.payload.noteTitle, { lower: true }),
      description: request.payload.noteDescription,
      content: request.payload.noteContent,
    });
```

```
  return { note }
},
```

We're going to generate a view later, but for now let's just return the result. Once we build the views in the next section, we will be able to generate the HTML with the new note and add it dynamically on the client. Although this is not completely necessary and will depend on how you are going to handle your front-end logic, we're going to return an HTML block to simplify the logic on the client.

Also, note that the date is being generated on the fly when we execute the function, using new *Date()*.

The "read" function

To search for just one element we use the *findOne* method on our model. Since we identify notes by their slug, the *where* filter must contain the slug provided by the client in the URL (*http://localhost:3000/note/:slug*:).

```
read: async (request, h) => {
  const note = await Models.Note.findOne({
    where: { slug: request.params.slug },
  });

  return { note };
},
```

As in the previous function, we will just return the result, which is going to be an object containing the note information. The views are going to be used once we create them in the Building the Views section.

The "update" function

To update a note, we use the *update* method on our model. It takes two objects, the new values that we're going to replace and the options containing a *where* filter with the note slug (which identifies the note that we're going to update).

```
update: async (request, h) => {
  const values = {
    title: request.payload.noteTitle,
    description: request.payload.noteDescription,
    content: request.payload.noteContent,
  };

  const options = {
    where: { slug: request.params.slug },
  };

  await Models.Note.update(values, options);
  const note = await Models.Note.findOne(options);

  return { note };
},
```

After updating our data, our database won't return the updated note. Therefore we should find the modified note and return it to the client, so we can show the updated version as soon as the changes are made.

The "delete" function

The *delete* controller will remove the note by providing the slug to the *destroy* function of our model. Then, once the note is deleted, we redirect to the home page. To accomplish this, we use the redirect function provided by the response toolkit.

```
 delete: async (request, h) => {
  await Models.Note.destroy({ where: { slug: request.params.slug } });
  return h.redirect('/');
},
```

Using the Note controller in our routes

At this point, we should have our note controller file ready with all the CRUD

actions. But to use them, we have to include it in our routes file.

First, let's import our controller at the top of the *routes.js* file:

```
const Note = require('./controllers/note');
```

We have to replace each hander with our new functions, so we should have our routes file as follows:

```
{
  method: 'POST',
  path: '/note',
  handler: Note.create,
  config: {
    description: 'Adds a new note',
  },
},
{
  method: 'GET',
  path: '/note/{slug}',
  handler: Note.read,
  config: {
    description: 'Gets the content of a note',
  },
},
{
  method: 'PUT',
  path: '/note/{slug}',
  handler: Note.update,
  config: {
    description: 'Updates the selected note',
  },
},
{
  method: 'GET',
  path: '/note/{slug}/delete',
  handler: Note.delete,
  config: {
```

```
    description: 'Deletes the selected note',
  },
},
```

Referencing Functions, Not Calling Them

We're including our functions without `()` at the end, that is because we're referencing our functions without calling them.

To test that everything works so far, restart your server, then visit: http://localhost:3000 - you should see `{"notes":[]}` http://localhost:3000/note/1 - you should see `{ "note": null }` http://localhost:3000/note/1/delete - you should be redirected to '/'.

Next, we'll set up the app's Views. Nearly there!

Chapter 8

Building the Views

At this point, our site is receiving HTTP calls and responding with JSON objects. To make it useful to everybody we have to create the pages that render our information in a nice way.

In this example, we're going to use the Pug templating language, although this is not mandatory and we can use other languages with Hapi.js. Also, we're going to use the Vision plugin to enable the view functionality in our server.

Pug?

If you're not familiar with Pug (formerly Jade), see our Jade Tutorial for Beginners.

You can install the packages with the following command:

```
npm install -s vision pug
```

The note component

First, we're going to build our note component that is going to be reused across our views. Also, we're going to use this component in some of our controller functions to build a note on the fly in the back-end. This will simplify the logic on the client.

Create a file in *lib/views/components* called *note.pug* with the following content:

```
article
  h2: a(href=`/note/${note.slug}`)= note.title
  small Published on #{note.date}
  p= note.content
```

It is composed of the title of the note, the publication date and the content of the note.

Whitespace Sensitivity

Pug is white space/indentation sensitive.

The base layout

The base layout contains the common elements of our pages; or in other words, for our example, everything that is not content. Create a file in *lib/views/* called *layout.pug* with the following content:

```
doctype html
  html(lang='en')
    head
      meta(charset='utf-8')
      meta(http-equiv='x-ua-compatible' content='ie=edge')
      title= page
      meta(name='description' content=description)
      meta(name='viewport' content='width=device-width, initial-scale=1')

      link(href='https://fonts.googleapis.com/css?family=Gentium+Book+Basic:400,
      ↪400i,700,700i|Ubuntu:500' rel='stylesheet')
      link(rel='stylesheet' href='/styles/main.css')
    body
      block content

      script(src='/scripts/jquery.min.js')
      script(src='/scripts/jquery.modal.min.js')
      script(src='/scripts/main.js')
```

The content of the other pages will be loaded in place of *block content*. Also, note that we will display a *page* variable in the *title* element, and a *description* variable in the *meta(name='description') element*. We will create those variables in our routes later.

We also include, a CSS file at the top of the page, as well as three JS files at the

bottom. These are jQuery, jQuery Modal and a *main.js* file which will contain all of our custom JS code for the front-end. Be sure to download all of these files and put them in the correct folder within the *static/public/* directory.

We're going to make them public in a short while, in the Serving Static Files section.

The home view

On our home page, we will show a list containing all the notes in our database and a button that will show a modal window with a form that allows us to create a new note via Ajax.

Create a file in *lib/views* called *home.pug* with the following content:

```
extends layout

  block content
    header(container)
      h1 Notes Board

      nav
        // This will show a modal window with a form to send new notes
        ul
          li: a(href='#note-form' rel='modal:open') Publish

    main(container).notes-list
      // We loop over all the notes received from our controller rendering our note
      ↪component with each entry each note in data.notes
        include components/note

    form(action='/note' method='POST').note-form#note-form
      // Form to add a new note, this is used by our controller `create` function.
      ↪form(action='/note' method='POST').note-form#note-form
      p: input(name='noteTitle' type='text' placeholder='Title…')
      p: input(name='noteDescription' type='text' placeholder='Short description…')
      p: textarea(name='noteContent' placeholder='Write the content of the new n
```

```
      ↪ote here…')
      p._text-right: input(type='submit' value='Submit')
```

The note view

The note page is pretty similar to the home page, but in this case, we show a menu with options specific to the current note, the content of the note and the same form as on the home page but with the current note information already filled out, so it's there when we update it.

Create a file in *lib/views* called *note.pug* with the following content:

```
extends layout

  block content
    header(container)
      h1 Notes Board

      nav
        ul
          li: a(href='/') Home
          li: a(href='#note-form' rel='modal:open') Update
          li: a(href=`/note/${note.slug}/delete`) Delete

    main(container).note-content
      include components/note

    form(action=`/note/${note.slug}` method='PUT').note-form#note-form
      p: input(name='noteTitle' type='text' value=note.title)
      p: input(name='noteDescription' type='text' value=note.description)
      p: textarea(name='noteContent')= note.content
      p._text-right: input(type='submit' value='Update')
```

The JavaScript on the client

To create and update notes we use jQuery's Ajax functionality. Although this is not strictly necessary, I feel it provides a better experience for the user.

This is the content of our *main.js* file in the *static/public/scripts/* directory:

```
var $noteForm = $('#note-form');

  $noteForm.submit(function (e) {
    e.preventDefault();

    var form = {
      url: $(this).attr('action'),
      type: $(this).attr('method')
    };

    $.ajax({
      url: form.url,
      type: form.type,
      data: $(this).serialize(),
      success: function (result) {
        $.modal.close();

        if (form.type === 'POST') {
          $('.notes-list').prepend(result);
          $noteForm.find('input[type=text], textarea').val('');
        } else if (form.type === 'PUT') {
          $('.note-content').html(result);
        }
      }
    });
  });
```

Every time the user submits the form in the modal window, we get the information from the form elements and send it to our back-end, depending on the action URL and the method (*POST* or *PUT*). Then, we will get the result as a block of HTML containing our new note data. When we add a note, we will just

add it to the top of the list on the home page, and when we update a note we replace the content for the new one in the note view.

We didn't update any of our controllers in this chapter, so there's nothing new to see in the browser.

That was a lot of code, but the good news is that we're nearly done! Now we have all of the Views set up and supported by our server, we'll quickly add support for views on the server, and make sure our server is serving static files. Once we've done that, we'll be all ready to go.

Chapter

9

Adding Support for Views on the Server, and Serving Static Files

To make use of our views, we have to include them in our controllers and add the required settings.

In our *server.js* file, let's import the Node Path utility, as well as the Vision and Pug libraries, that we installed earlier. Add the following to the top of the file:

```
const Path = require('path');
const Pug = require('pug');
const Vision = require('vision');
```

Now, alter the _init_function like so:

```
const init = async () => {
  await server.register([ Vision ]);

  server.views({
    engines: { pug: Pug },
    path: Path.join(__dirname, 'lib/views'),
    compileOptions: {
      pretty: false,
    },
    isCached: Settings.env === 'production'
  });

  server.route(Routes);

  await server.start();
  console.log(`Server running at: ${server.info.uri}`);
};
```

In the code we have added, we first register the Vision plugin with our Hapi.js server, which is going to provide the view functionality. Then, we add the settings for our views — like the engine that we're going to use and the path where the views are located. After this code block, we add our routes.

This will make our views work on the server, but we still have to declare the view that we're going to use for each route.

Setting the home view

Open the *lib/controllers/home.js* file and replace the *return { notes };* line with the following:

```
return h.view('home', {
  data: { notes },
  page: 'Home—Notes Board',
  description: 'Welcome to my Notes Board',
});
```

After registering the Vision plugin, we now have a *view* method available on the response toolkit (*h*), we're going to use it to select the home view in our *views* directory and to send the data that is going to be used when rendering the views.

In the data that we provide to the view, we also include the page title and a meta description for search engines.

Setting the note view: create method

Right now, every time we create a note, we get a JSON object sent from the server to the client. But since we're doing this process via Ajax, we can send the new note as HTML ready to be added to the page. To do this, we render the *note* component with the data we have.

In *lib/controllers/note.js*, require the Pug library:

```
const Pug = require('pug');
```

Then, replace the *create* method's return statement (*return { note }*) with the following:

```
const newNote = Pug.renderFile(
  Path.join(__dirname, '../views/components/note.pug'),
```

```
  { note },
);

return newNote;
```

Here, we use Pug's *renderFile* method to render the note template with the data we just received from our model.

Setting the note view: read method

When we enter a note page, we should get the note template with the content of our note. To do this, we have to replace the *read* method's return statement (*return { note }*) with the following:

```
return h.view('note', {
  note,
  page: `${note.title}–Notes Board`,
  description: note.description,
});
```

As with the home page, we select a view as the first argument and the data that we're going to use as the second one.

Setting the note view: update method

Every time we update a note, we will reply similarly as to when we create new notes. Replace the the *update* method's return statement (*return { note }*) with the following:

```
const updatedNote = Pug.renderFile(
  Path.join(__dirname, '../views/components/note.pug'),
  { note },
);

return updatedNote;
```

No View for Delete

The delete function doesn't need a view, since it will just redirect to the home page once the note is deleted.

Serving Static Files

The JavaScript and CSS files that we're using on the client side are served by Hapi.js from the *static/public/* directory. But it won't happen automatically; we have to indicate to the server that we want to define this folder as public. This is done using the Inert package, which you can install with the following command:

```
npm install -s inert
```

In *server.js* import the library at the top of the page:

```
const Inert = require('inert');
```

Then in the *server.register* function, register it with Hapi like so:

```
await server.register([ Vision, Inert ]);
```

Now we have to define the route where we're going to provide the static files, and their location on our server's file system. Add the following entry at the end of the exported object in *routes.js*:

```
{
  // Static files
  method: 'GET',
  path: '/{p'GET'}',
  handler: {
    directory: {
      path: Path.join(__dirname, '../static/public'),
```

```
    },
  },
  config: {
    description: 'Provides static resources',
  },
},
```

This route will use the *GET* method and we have replaced the handler function with an object containing the directory that we want to make public.

You can find more information about serving static content in the Hapi.js documentation.

Github

Check the GitHub repository for the rest of the static files, like the main style sheet..

Finally, restart your server and visit your fully functioning Node MVC application.

You did it!

You now have a basic Node/Hapi note-taking app using the MVC architecture.

Congratulations and thanks for joining me!