

9 PRACTICAL NODE.JS PROJECTS



LEVEL UP YOUR NODE KNOWLEDGE

9 Practical Node.js Projects

Copyright © 2018 SitePoint Pty. Ltd.

Cover Design: Alex Walker

Notice of Rights

All rights reserved. No part of this book may be reproduced, stored in a retrieval system or transmitted in any form or by any means, without the prior written permission of the publisher, except in the case of brief quotations embodied in critical articles or reviews.

Notice of Liability

The author and publisher have made every effort to ensure the accuracy of the information herein. However, the information contained in this book is sold without warranty, either express or implied. Neither the authors and SitePoint Pty. Ltd., nor its dealers or distributors will be held liable for any damages to be caused either directly or indirectly by the instructions contained in this book, or by the software or hardware products described herein.

Trademark Notice

Rather than indicating every occurrence of a trademarked name as such, this book uses the names only in an editorial fashion and to the benefit of the trademark owner with no intention of infringement of the trademark.



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, design, and more.

Preface

While there have been quite a few attempts to get JavaScript working as a server-side language, Node.js (frequently just called Node) has been the first environment that's gained any traction. It's now used by companies such as Netflix, Uber and Paypal to power their web apps. Node allows for blazingly fast performance; thanks to its event loop model, common tasks like network connection and database I/O can be executed very quickly indeed.

From a beginner's point of view, one of Node's obvious advantages is that it uses JavaScript, a ubiquitous language that many developers are comfortable with. If you can write JavaScript for the client-side, writing server-side applications with Node should not be too much of a stretch for you.

In this book, we'll offer a selection of nine different practical projects that you can follow along with.

Who Should Read This Book?

This book is for anyone who wants to start learning server-side development with Node.js. Familiarity with JavaScript is assumed, but we don't assume any previous back-end development experience.

Conventions Used

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");
```

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

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.

Chapter 1: Build a Simple Beginner App with Node, Bootstrap & MongoDB

BY JAMES HIBBARD

If you're just getting started with Node.js and want to try your hand at building a web app, things can often get a little overwhelming. Once you get beyond the "Hello, World!" tutorials, much of the material out there has you copy-pasting code, with little or no explanation as to what you're doing or why.

This means that, by the time you've finished, you've built something nice and shiny, but you also have relatively few takeaways that you can apply to your next project.

In this tutorial, I'm going to take a slightly different approach. Starting from the ground up, I'll demonstrate how to build a no-frills web app using Node.js, but instead of focusing on the end result, I'll focus on a range of things you're likely to encounter when building a real-world app. These include routing, templating, dealing with forms, interacting with a database and even basic authentication.

This won't be a JavaScript 101. If that's the kind of thing you're after, [look here](#). It will, however, be suitable for those people who feel reasonably confident with the JavaScript language, and who are looking to take their first steps in Node.js.

What We'll Be Building

We'll be using [Node.js](#) and the [Express framework](#) to build a simple registration form with basic validation, which persists its data to a [MongoDB database](#). We'll add a view to list successful registration, which we'll protect with basic HTTP authentication, and we'll use [Bootstrap](#) to add some styling. The tutorial is structured so that you can follow along step by step. However, if you'd like to jump ahead and see the end result, [the code for this tutorial is also available on GitHub](#).

Basic Setup

Before we can start coding, we'll need to get Node, npm and MongoDB installed on our machines. I won't go into depth on the various installation instructions, but if you have any trouble getting set up, please leave a comment below, or [visit our forums](#) and ask for help there.

NODE.JS

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 rights.

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.

NPM

npm is a JavaScript package manager which comes bundled with Node, so no extra installation is necessary here. We'll be making quite extensive use of npm throughout this tutorial, so if you're in need of a refresher, please consult: [A Beginner's Guide to npm — the Node Package Manager](#).

MONGODB

MongoDB is a document database which stores data in flexible, JSON-like documents.

The quickest way to get up and running with Mongo is to use a service such as mLabs. They have a free sandbox plan which provides a single database with 496 MB of storage running on a shared virtual machine. This is more than adequate for a simple app with a handful of users. If this sounds like the best option for you, please consult their [quick start guide](#).

You can also install Mongo locally. To do this, please visit the [official download page](#) and download the correct version of the community server for your operating system. There's a link to detailed, OS-specific installation instructions beneath every download link, which you can consult if you run into trouble.

A MONGODB GUI

Although not strictly necessary for following along with this tutorial, you might also like to install [Compass, the official GUI for MongoDB](#). This tool helps you visualize and manipulate your data, allowing you to interact with documents with full CRUD functionality.

At the time of writing, you'll need to fill out your details to download Compass, but you won't need to create an account.

CHECK THAT EVERYTHING IS INSTALLED CORRECTLY

To check that Node and npm are installed correctly, open your terminal and type:

```
node -v
```

followed by:

```
npm -v
```

This will output the version number of each program (8.9.4 and 5.6.0 respectively at the time of writing).

If you installed Mongo locally, you can check the version number using:

```
mongo --version
```

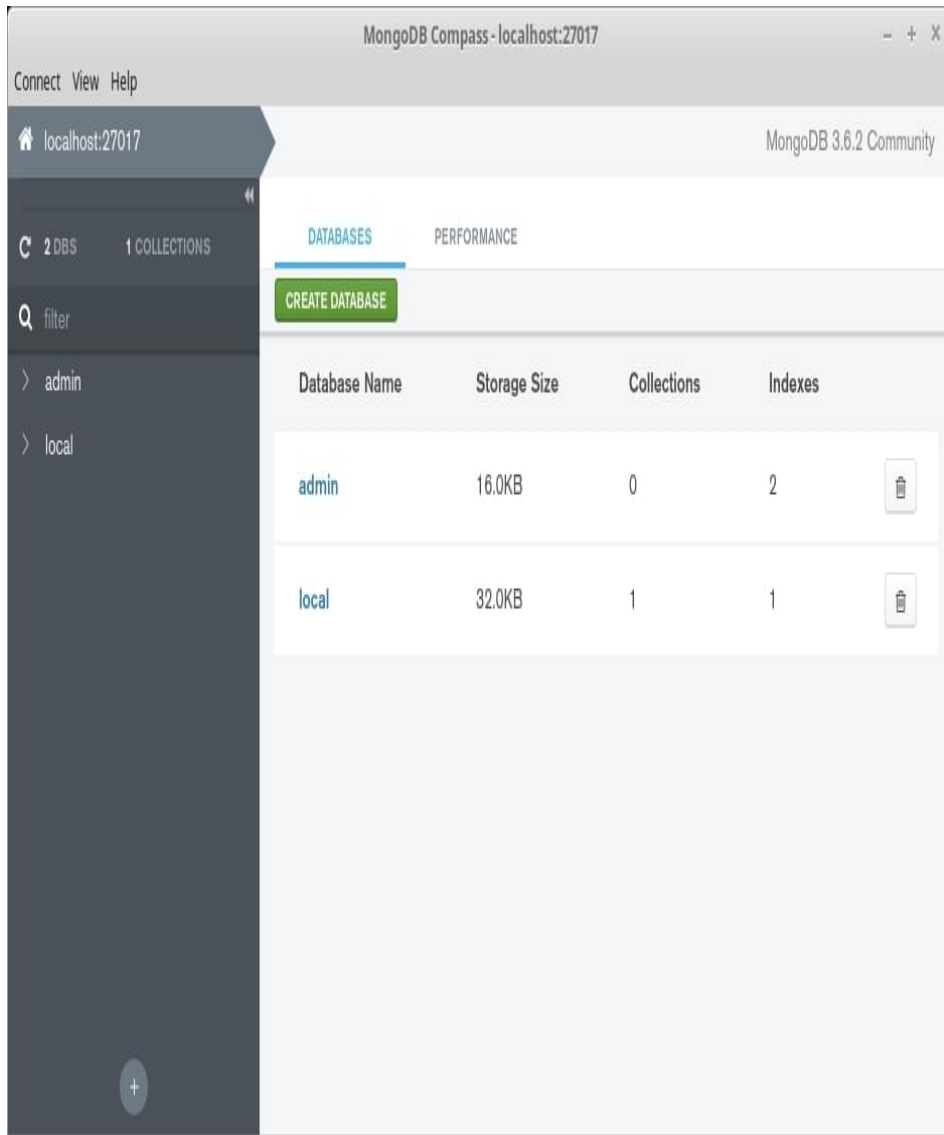
This should output a bunch of information, including the version number (3.6.2 at the time of writing).

CHECK THE DATABASE CONNECTION USING COMPASS

If you have installed Mongo locally, you start the server by typing the following command into a terminal:

```
mongod
```

Next, open Compass. You should be able to accept the defaults (server: localhost, port: 27017), press the *CONNECT* button, and establish a connection to the database server.



MongoDB Compass connected to localhost

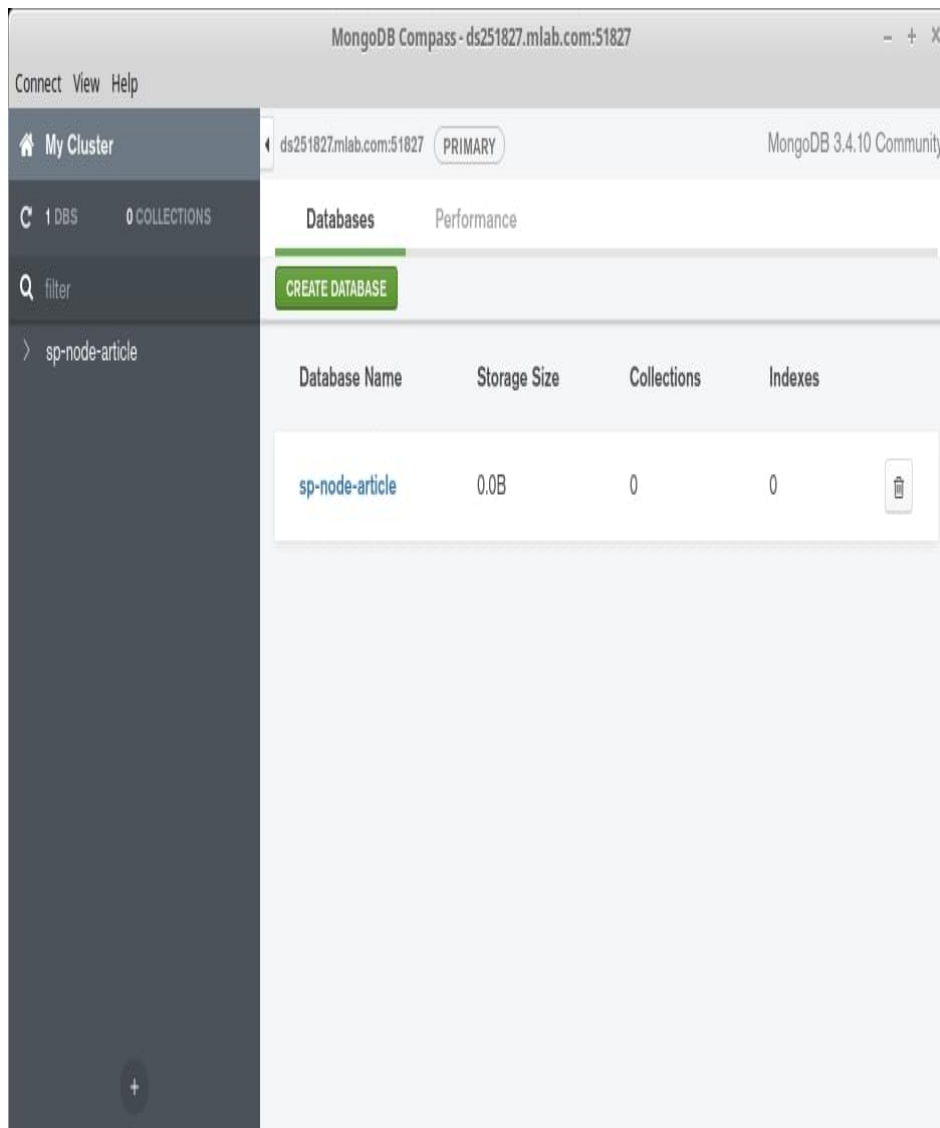
Note that the databases `admin` and `local` are created automatically.

Using a Cloud-hosted Solution

If you're using mLab, create a database subscription (as described in their [quick-start guide](#)), then copy the connection details to the clipboard. This should be in the form:

```
mongodb://<dbuser>:  
<dbpassword>@ds251827.mlab.com:51827/<dbname>
```

When you open Compass, it will inform you that it has detected a MongoDB connection string and asks if you would like to use it to fill out the form. Click *Yes*, noting that you might need to adjust the username and password by hand. After that, click *CONNECT* and you should be off to the races.



MongoDB Compass connected to mLabs

Note that I called my database `sp-node-article`. You can call yours what you like.

Initialize the Application

With everything set up correctly, the first thing we need to do is initialize our new project. To do this, create a folder named `demo-node-app`, enter that directory and type the following in a terminal:

```
npm init -y
```

This will create and auto-populate a `package.json` file in the project root. We can use this file to specify our dependencies and to create various npm scripts, which will aid our development workflow.

INSTALL EXPRESS

Express is a lightweight web application framework for Node.js, which provides us with a robust set of features for writing web apps. These features include such things as route handling, template engine integration and a middleware framework, which allows us to perform additional tasks on request and response objects. There is nothing you can do in Express that you couldn't do in plain Node.js, but using Express means we don't have to re-invent the wheel and reduces boilerplate.

So let's install Express. To do this, run the following in your terminal:

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

By passing the `--save` option to the `npm install` command, Express will be added to the `dependencies` section of the

`package.json` file. This signals to anyone else running our code that Express is a package our app needs to function properly.

INSTALL NODEMON

nodemon is a convenience tool. It will watch the files in the directory it was started in, and if it detects any changes, it will automatically restart your Node application (meaning you don't have to). In contrast to Express, nodemon is not something the app requires to function properly (it just aids us with development), so install it using:

```
npm install --save-dev nodemon
```

This will add nodemon to the `dev-dependencies` section of the `package.json` file.

CREATE SOME INITIAL FILES

We're almost through with the setup. All we need to do now is create a couple of initial files before kicking off the app.

In the `demo-node-app` folder create an `app.js` file and a `start.js` file. Also create a `routes` folder, with an `index.js` file inside. After you're done, things should look like this:

```
.
├── app.js
├── node_modules
│   └── ...
├── package.json
├── routes
│   └── index.js
└── start.js
```

Now, let's add some code to those files.

In `app.js`:

```
const express = require('express');
const routes = require('./routes/index');

const app = express();
app.use('/', routes);

module.exports = app;
```

Here, we're importing both the `express` module and (the export value of) our routes file into the application. The `require` function we're using to do this is a built-in Node function which imports an object from another file or module. If you'd like a refresher on importing and exporting modules, read [Understanding module.exports and exports in Node.js](#).

After that, we're creating a new Express app using the `express` function and assigning it to an `app` variable. We then tell the app that, whenever it receives a request from forward slash anything, it should use the routes file.

Finally, we export our app variable so that it can be imported and used in other files.

In `start.js`:

```
const app = require('./app');

const server = app.listen(3000, () => {
  console.log(`Express is running on port
${server.address().port}`);
});
```

Here we're importing the Express app we created in `app.js` (note that we can leave the `.js` off the file name in the `require` statement). We then tell our app to listen on port 3000 for incoming connections and output a message to the terminal to indicate that the server is running.

And in `routes/index.js`:

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

const router = express.Router();

router.get('/', (req, res) => {
  res.send('It works!');
});

module.exports = router;
```

Here, we’re importing Express into our routes file and then grabbing the router from it. We then use the router to respond to any requests to the root URL (in this case `http://localhost:3000`) with an “It works!” message.

KICK OFF THE APP

Finally, let’s add an npm script to make nodemon start watching our app. Change the `scripts` section of the `package.json` file to look like this:

```
"scripts": {
  "watch": "nodemon ./start.js"
},
```

The `scripts` property of the `package.json` file is extremely useful, as it lets you specify arbitrary scripts to run in different scenarios. This means that you don’t have to repeatedly type out long-winded commands with a difficult-to-remember syntax. If you’d like to find out more about what npm scripts can do, read [Give Grunt the Boot! A Guide to Using npm as a Build Tool](#).

Now, type `npm run watch` from the terminal and visit <http://localhost:3000>.

You should see “It works!”

Basic Templating with Pug

Returning an inline response from within the route handler is all well and good, but it's not very extensible, and this is where templating engines come in. As the [Express docs](#) state:

A template engine enables you to use static template files in your application. At runtime, the template engine replaces variables in a template file with actual values, and transforms the template into an HTML file sent to the client.

In practice, this means we can define template files and tell our routes to use them instead of writing everything inline. Let's do that now.

Create a folder named `views` and in that folder a file named `form.pug`. Add the following code to this new file:

```
form(action="." method="POST")
  label(for="name") Name:
  input(
    type="text"
    id="name"
    name="name"
  )

  label(for="email") Email:
  input(
    type="email"
    id="email"
    name="email"
  )

  input(type="submit" value="Submit")
```

As you can deduce from the file ending, we'll be using the [pug templating engine](#) in our app. Pug (formerly known as Jade) comes with its own indentation-sensitive syntax for writing dynamic and reusable HTML. Hopefully the above example is easy to follow, but if you have any difficulties understanding what it does, just wait until we view this in a browser, then inspect the page source to see the markup it produces.

If you'd like a refresher as to what JavaScript templates and/or templating engines are, and when you should use them, read [An Overview of JavaScript Templating Engines](#).

INSTALL PUG AND INTEGRATE IT INTO THE EXPRESS APP

Next, we'll need to install pug, saving it as a dependency:

```
npm i --save pug
```

Then configure `app.js` to use Pug as a layout engine and to look for templates inside the `views` folder:

```
const express = require('express');
const path = require('path');
const routes = require('./routes/index');

const app = express();

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

app.use('/', routes);

module.exports = app;
```

You'll notice that we're also requiring Node's native [Path module](#), which provides utilities for working with file and directory paths. This module allows us to build the path to our `views` folder using its [join method](#) and [__dirname](#) (which returns the directory in which the currently executing script resides).

ALTER THE ROUTE TO USE OUR TEMPLATE

Finally, we need to tell our route to use our new template. In `routes/index.js`:

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

This uses the `render` method on Express's response object to send the rendered view to the client.

So let's see if it worked. As we're using nodemon to watch our app for changes, you should simply be able to refresh your browser and see our brutalist masterpiece.

DEFINE A LAYOUT FILE FOR PUG

If you open your browser and inspect the page source, you'll see that Express only sent the HTML for the form: our page is missing a doc-type declaration, as well as a head and body section. Let's fix that by creating a master layout for all our templates to use.

To do this, create a `layout.pug` file in the `views` folder and add the following code:

```
doctype html  
html  
  head  
    title= `${title}`  
  
  body  
    h1 My Amazing App  
  
    block content
```

The first thing to notice here is the line starting `title=`. Appending an equals sign to an attribute is one of the methods that Pug uses for interpolation. You can read more about it [here](#). We'll use this to pass the title dynamically to each template.

The second thing to notice is the line that starts with the `block` keyword. In a template, a block is simply a “block” of Pug that a

child template may replace. We'll see how to use it shortly, but if you're keen to find out more, read [this page on the Pug website](#).

USE THE LAYOUT FILE FROM THE CHILD TEMPLATE

All that remains to do is to inform our `form.pug` template that it should use the layout file. To do this, alter `views/form.pug`, like so:

```
extends layout

block content
  form(action="." method="POST")
    label(for="name") Name:
    input(
      type="text"
      id="name"
      name="name"
    )

    label(for="email") Email:
    input(
      type="email"
      id="email"
      name="email"
    )

    input(type="submit" value="Submit")
```

And in `routes/index.js`, we need to pass in an appropriate title for the template to display:

```
router.get('/', (req, res) => {
  res.render('form', { title: 'Registration form' });
});
```

Now if you refresh the page and inspect the source, things should look a lot better.

Dealing with Forms in Express

Currently, if you hit our form's *Submit* button, you'll be redirected to a page with a message: "Cannot POST /". This is because when submitted, our form POSTs its contents back to / and we haven't defined a route to handle that yet.

Let's do that now. Add the following to `routes/index.js`:

```
router.post('/', (req, res) => {  
  res.render('form', { title: 'Registration form' });  
});
```

This is the same as our GET route, except for the fact that we're using `router.post` to respond to a different HTTP verb.

Now when we submit the form, the error message will be gone and the form should just re-render.

HANDLE FORM INPUT

The next task is to retrieve whatever data the user has submitted via the form. To do this, we'll need to install a package named `body-parser`, which will make the form data available on the request body:

```
npm install --save body-parser
```

We'll also need to tell our app to use this package, so add the following to `app.js`:

```
const bodyParser = require('body-parser');  
...  
app.use(bodyParser.urlencoded({ extended: true }));  
app.use('/', routes);  
  
module.exports = app;
```

Note that there are various ways to format the data you POST to the server, and using `body-parser`'s `urlencoded` method allows

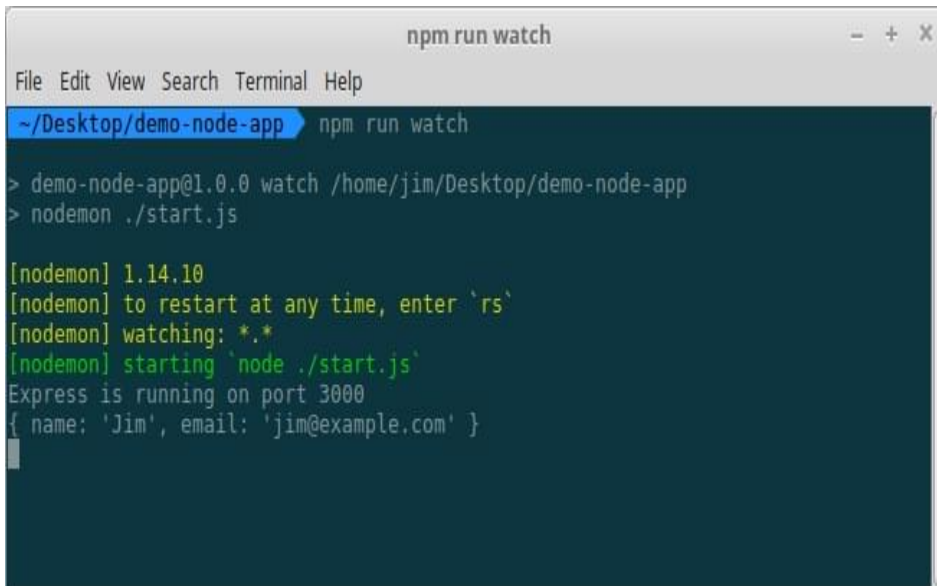
us to handle data sent as `application/x-www-form-urlencoded`.

Then we can try logging the submitted data to the terminal. Alter the route handler like so:

```
router.post('/', (req, res) => {  
  console.log(req.body);  
  res.render('form', { title: 'Registration form' });  
});
```

Now when you submit the form, you should see something along the lines of:

```
{name: 'Jim', email: 'jim@example.com'}
```



```
npm run watch  
File Edit View Search Terminal Help  
~/Desktop/demo-node-app npm run watch  
> demo-node-app@1.0.0 watch /home/jim/Desktop/demo-node-app  
> nodemon ./start.js  
  
[nodemon] 1.14.10  
[nodemon] to restart at any time, enter `rs`  
[nodemon] watching: *.*  
[nodemon] starting `node ./start.js`  
Express is running on port 3000  
{ name: 'Jim', email: 'jim@example.com' }
```

Form output logged to terminal

A NOTE ABOUT REQUEST AND RESPONSE OBJECTS

By now you've hopefully noticed the pattern we're using to handle routes in Express.

```
router.METHOD(route, (req, res) => {  
  // callback function  
});
```

The callback function is executed whenever somebody visits a URL that matches the route it specifies. The callback receives a `req` and `res` parameter, where `req` is an object full of information that is coming in (such as form data or query parameters) and `res` is an object full of methods for sending data back to the user. There's also an optional `next` parameter which is useful if you don't actually want to send any data back, or if you want to pass the request off for something else to handle.

Without getting too deep into the weeds, this is a concept known as middleware (specifically, router-level middleware) which is very important in Express. If you're interested in finding out more about how Express uses middleware, I recommend you read the [Express docs](#).

VALIDATING FORM INPUT

Now let's check that the user has filled out both our fields. We can do this using [express-validator module](#), a middleware that provides a number of useful methods for the sanitization and validation of user input.

You can install it like so:

```
npm install express-validator --save
```

And require the functions we'll need in `routes/index.js`:

```
const { body, validationResult } = require('express-validator/check');
```

We can include it in our route handler like so:

```

router.post('/',
  [
    body('name')
      .isLength({ min: 1 })
      .withMessage('Please enter a name'),
    body('email')
      .isLength({ min: 1 })
      .withMessage('Please enter an email'),
  ],
  (req, res) => {
    ...
  }
);

```

As you can see, we’re using the body method to validate two properties on `req.body` — namely, `name` and `email`. In our case, it’s sufficient to just check that these properties exist (i.e. that they have a length greater than one), but `express-validator` offers a whole host of other methods that you can read about on the project’s home page.

In a second step, we can call the validationResult method to see if validation passed or failed. If no errors are present, we can go ahead and render out a “Thanks for registering” message. Otherwise, we’ll need to pass these errors back to our template, so as to inform the user that something’s wrong.

And if validation fails, we’ll also need to pass `req.body` back to the template, so that any valid form inputs aren’t reset:

```

router.post(
  '/',
  [
    ...
  ],
  (req, res) => {
    const errors = validationResult(req);

    if (errors.isEmpty()) {
      res.send('Thank you for your registration!');
    } else {
      res.render('form', {
        title: 'Registration form',
        errors: errors.array(),
        data: req.body,
      });
    }
  }
);

```



```
    }  
  }  
);
```

Now we have to make a couple of changes to our `form.pug` template. We firstly need to check for an `errors` property, and if it's present, loop over any errors and display them in a list:

```
extends layout  
  
block content  
  if errors  
    ul  
      for error in errors  
        li= error.msg  
    ...
```

If the `li=` looks weird, remember that pug does interpolation by following the tag name with an equals sign.

Finally, we need to check if a `data` attribute exists, and if so, use it to set the values of the respective fields. If it doesn't exist, we'll initialize it to an empty object, so that the form will still render correctly when you load it for the first time. We can do this with some JavaScript, denoted in Pug by a minus sign:

```
-data = data || {}
```

We then reference that attribute to set the field's value:

```
input(  
  type="text"  
  id="name"  
  name="name"  
  value=data.name  
)
```

That gives us the following:

```
extends layout  
  
block content
```

```

    -data = data || {}

    if errors
      ul
        for error in errors
          li= error.msg

    form(action="." method="POST")
      label(for="name") Name:
      input(
        type="text"
        id="name"
        name="name"
        value=data.name
      )

      label(for="email") Email:
      input(
        type="email"
        id="email"
        name="email"
        value=data.email
      )

      input(type="submit" value="Submit")

```

Now, when you submit a successful registration, you should see a thank you message, and when you submit the form without filling out both field, the template should be re-rendered with an error message.

Interact with a Database

We now want to hook our form up to our database, so that we can save whatever data the user enters. If you're running Mongo locally, don't forget to start the server with the command `mongod`.

SPECIFY CONNECTION DETAILS

We'll need somewhere to specify our database connection details. For this, we'll use a configuration file (which *should not* be checked into version control) and the dotenv package.

Dotenv will load our connection details from the configuration file into Node's `process.env`.

Install it like so:

```
npm install dotenv --save
```

And require it at the top of `start.js`:

```
require('dotenv').config();
```

Next, create a file named `.env` in the project root (note that starting a filename with a dot may cause it to be hidden on certain operating systems) and enter your Mongo connection details on the first line.

If you're running Mongo locally:

```
DATABASE=mongodb://localhost:27017/node-demo-application
```

If you're using m Labs:

```
mongodb://<dbuser>:  
<dbpassword>@ds251827.mlab.com:51827/<dbname>
```

Note that local installations of MongoDB don't have a default user or password. This is definitely something you'll want to change in production, as it's otherwise a security risk.

CONNECT TO THE DATABASE

To establish the connection to the database and to perform operations on it, we'll be using Mongoose. Mongoose is an ORM for MongoDB, and as you can read on the project's home page:

Mongoose provides a straight-forward, schema-based solution to model your application data. It includes built-in type casting, validation, query building, business logic hooks and more, out of the box.

What this means in real terms is that it creates various abstractions over Mongo, which make interacting with our database easier and reduce the amount of boilerplate we have to write. If you'd like to find out more about how Mongo works under the hood, be sure to read our [Introduction to MongoDB](#).

To install Mongoose:

```
npm install --save mongoose
```

Then, require it in `start.js`:

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

The connection is made like so:

```
mongoose.connect(process.env.DATABASE, { useMongoClient:
true });
mongoose.Promise = global.Promise;
mongoose.connection
  .on('connected', () => {
    console.log(`Mongoose connection open on
${process.env.DATABASE}`);
  })
  .on('error', (err) => {
    console.log(`Connection error: ${err.message}`);
  });
```

Notice how we use the `DATABASE` variable we declared in the `.env` file to specify the database URL. We're also telling Mongo to use ES6 Promises (these are necessary, as database interactions are asynchronous), as its own default promise library is deprecated.

This is what `start.js` should now look like:

```
require('dotenv').config();
const mongoose = require('mongoose');

mongoose.connect(process.env.DATABASE, { useMongoClient:
true });
mongoose.Promise = global.Promise;
mongoose.connection
  .on('connected', () => {
    console.log(`Mongoose connection open on
${process.env.DATABASE}`);
  })
  .on('error', (err) => {
    console.log(`Connection error: ${err.message}`);
  });

const app = require('./app');
const server = app.listen(3000, () => {
  console.log(`Express is running on port
${server.address().port}`);
});
```

When you save the file, nodemon will restart the app and, if all's gone well, you should see something along the lines of:

```
Mongoose connection open on
mongodb://localhost:27017/node-demo-application
```

DEFINE A MONGOOSE SCHEMA

MongoDB can be used as a loose database, meaning it's not necessary to describe what data will look like ahead of time. However, out of the box it runs in strict mode, which means it'll only allow you to save data it knows about beforehand. As we'll be using strict mode, we'll need to define the shape our data using a schema. Schemas allow you to define the fields stored in each document along with their type, validation requirements and default values.

To this end, create a `models` folder in the project root, and within that folder, a new file named `Registration.js`.

Add the following code to `Registration.js`:

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

const registrationSchema = new mongoose.Schema({
  name: {
    type: String,
    trim: true,
  },
  email: {
    type: String,
    trim: true,
  },
});

module.exports = mongoose.model('Registration',
  registrationSchema);
```

Here, we're just defining a type (as we already have validation in place) and are making use of the trim helper method to remove any superfluous white space from user input. We then compile a model from the Schema definition, and export it for use elsewhere in our app.

The final piece of boilerplate is to require the model in `start.js`:

```
...

require('./models/Registration');
const app = require('./app');

const server = app.listen(3000, () => {
  console.log(`Express is running on port
  ${server.address().port}`);
});
```

SAVE DATA TO THE DATABASE

Now we're ready to save user data to our database. Let's begin by requiring Mongoose and importing our model into our `routes/index.js` file:

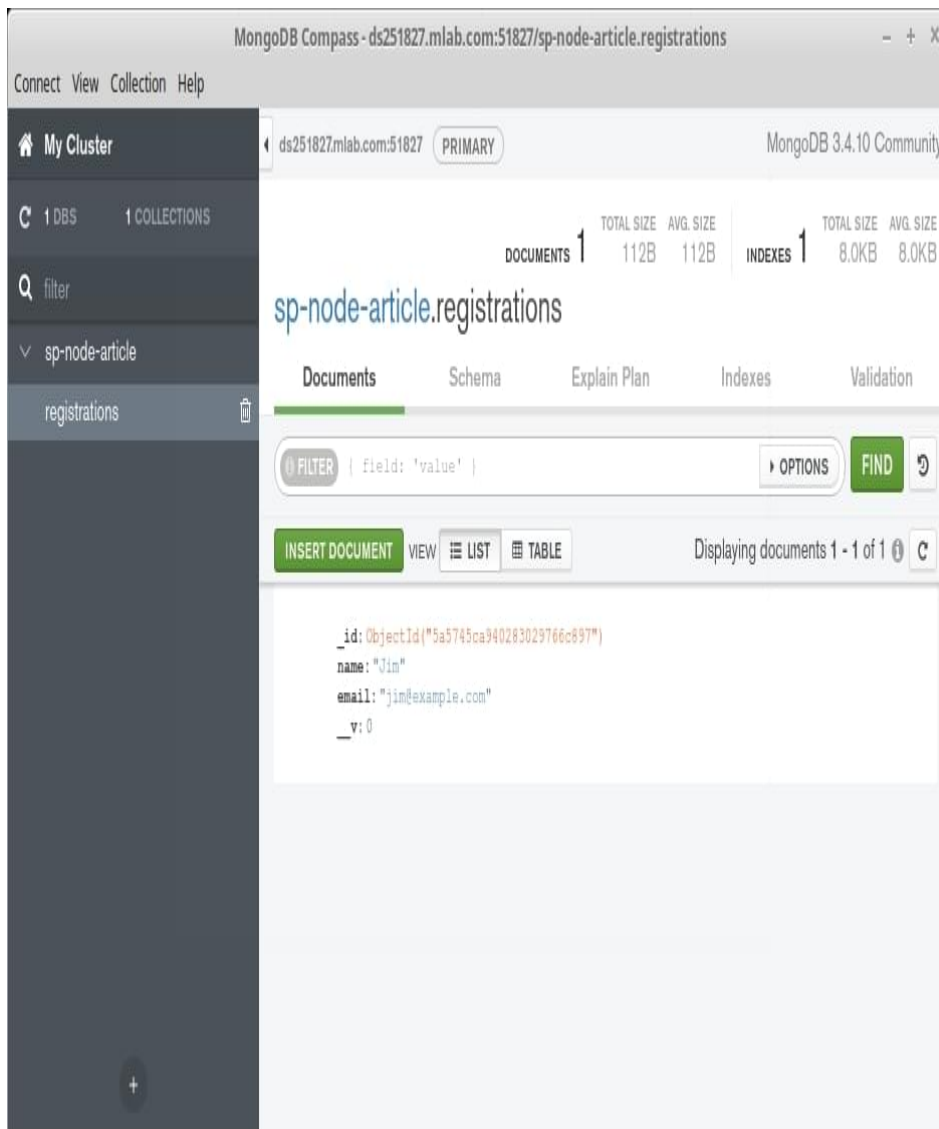
```
const express = require('express');
const mongoose = require('mongoose');
const { body, validationResult } = require('express-validator/check');

const router = express.Router();
const Registration = mongoose.model('Registration');
...
```

Now, when the user posts data to the server, if validation passes we can go ahead and create a new `Registration` object and attempt to save it. As the database operation is an asynchronous operation which returns a `Promise`, we can chain a `.then()` onto the end of it to deal with a successful insert and a `.catch()` to deal with any errors:

```
if (errors.isEmpty()) {
  const registration = new Registration(req.body);
  registration.save()
    .then(() => { res.send('Thank you for your registration!'); })
    .catch(() => { res.send('Sorry! Something went wrong.');
```

Now, if you enter your details into the registration form, they should be persisted to the database. You can check this using Compass (making sure to hit the refresh button in the top left if it's still running).



Using Compass to check that our data was saved to MongoDB

RETRIEVE DATA FROM THE DATABASE

To round the app off, let's create a final route, which lists out all of our registrations. Hopefully you should have a reasonable idea of the process by now.

Add a new route to `routes/index.js`, as follows:


```
router.get('/registrations', (req, res) => {  
  res.render('index', { title: 'Listing registrations'  
});  
});
```

This means that we'll also need a corresponding view template (views/index.pug):

```
extends layout  
  
block content  
  p No registrations yet :(
```

Now when you visit <http://localhost:3000/registrations>, you should see a message telling you that there aren't any registrations.

Let's fix that by retrieving our registrations from the database and passing them to the view. We'll still display the "No registrations yet" message, but only if there really aren't any.

In routes/index.js:

```
router.get('/registrations', (req, res) => {  
  Registration.find()  
    .then((registrations) => {  
      res.render('index', { title: 'Listing  
registrations', registrations });  
    })  
    .catch(() => { res.send('Sorry! Something went  
wrong.');
```

Here, we're using Mongo's [Collection#find method](#), which, if invoked without parameters, will return all of the records in the collection. Because the database lookup is asynchronous, we're waiting for it to complete before rendering the view. If any records were returned, these will be passed to the view template in the `registrations` property. If no records were returned `registrations` will be an empty array.

In `views/index.pug`, we can then check the length of whatever we're handed and either loop over it and output the records to the screen, or display a "No registrations" message:

```
extends layout

block content

  if registrations.length
    table
      tr
        th Name
        th Email
      each registration in registrations
        tr
          td= registration.name
          td= registration.email
  else
    p No registrations yet :(
```

Add HTTP Authentication

The final feature we'll add to our app is HTTP authentication, locking down the list of successful registrations from prying eyes.

To do this, we'll use the http-auth module, which we can install using:

```
npm install --save http-auth
```

Next we need to require it in `routes/index.js`, along with the Path module we met earlier:

```
const path = require('path');
const auth = require('http-auth');
```

Next, let it know where to find the file in which we'll list the users and passwords (in this case `users.htpasswd` in the project root):

```
const basic = auth.basic({
  file: path.join(__dirname, '../users.htpasswd'),
});
```

Create this `users.htpasswd` file next and add a username and password separated by a colon. This can be in plain text, but the `http-auth` module also supports hashed passwords, so you could also run the password through a service such as [Htpasswd Generator](#).

For me, the contents of `users.htpasswd` looks like this:

```
jim:$apr1$FhFmamtz$PgXfrNI95HFCuXIm30Q4V0
```

This translates to user: `jim`, password: `password`.

Finally, add it to the route you wish to protect and you're good to go:

```
router.get('/registrations', auth.connect(basic), (req,
res) => {
  ...
});
```

Serve Static Assets in Express

Let's give the app some polish and add some styling using [Twitter Bootstrap](#). We can serve static files such as images, JavaScript files and CSS files in Express using the built-in `express.static` middleware function.

Setting it up is easy. Just add the following line to `app.js`:

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

Now we can load files that are in the `public` directory.

STYLE THE APP WITH BOOTSTRAP

Create a `public` directory in the project root, and in the `public` directory create a `css` directory. Download the minified version of Bootstrap v4 into this directory, ensuring it's named `bootstrap.min.css`.

Next, we'll need to add some markup to our pug templates.

In `layout.pug`:

```
doctype html
html
  head
    title= `${title}`
    link(rel='stylesheet', href='/css/bootstrap.min.css')
    link(rel='stylesheet', href='/css/styles.css')

  body
    div.container.listing-reg
      h1 My Amazing App

      block content
```

Here, we're including two files from our previously created `css` folder and adding a wrapper `div`.

In `form.pug` we add some class names to the error messages and the form elements:

```
extends layout

block content
  -data = data || {}

  if errors
    ul.my-errors
      for error in errors
        li= error.msg

  form(action="." method="POST" class="form-registration")
    label(for="name") Name:
    input(
      type="text"
      id="name"
```

```

        name="name"
        class="form-control"
        value=data.name
    )

    label(for="email") Email:
    input(
        type="email"
        id="email"
        name="email"
        class="form-control"
        value=data.email
    )

    input(
        type="submit"
        value="Submit"
        class="btn btn-lg btn-primary btn-block"
    )

```

And in `index.pug`, more of the same:

```

extends layout

block content

    if registrations.length
        table.listing-table.table-dark.table-striped
            tr
                th Name
                th Email
            each registration in registrations
                tr
                    td= registration.name
                    td= registration.email
    else
        p No registrations yet :(

```

Finally, create a file called `styles.css` in the `css` folder and add the following:

```

body {
    padding: 40px 10px;
    background-color: #eee;
}
.listing-reg h1 {
    text-align: center;
    margin: 0 0 2rem;
}

/* css for registration form and errors*/

```

```

.form-registration {
  max-width: 330px;
  padding: 15px;
  margin: 0 auto;
}
.form-registration {
  display: flex;
  flex-wrap: wrap;
}
.form-registration input {
  width: 100%;
  margin: 0px 0 10px;
}
.form-registration .btn {
  flex: 1 0 100%;
}
.my-errors {
  margin: 0 auto;
  padding: 0;
  list-style: none;
  color: #333;
  font-size: 1.2rem;
  display: table;
}
.my-errors li {
  margin: 0 0 1rem;
}
.my-errors li:before {
  content: "! Error : ";
  color: #f00;
  font-weight: bold;
}

/* Styles for listing table */
.listing-table {
  width: 100%;
}
.listing-table th,
.listing-table td {
  padding: 10px;
  border-bottom: 1px solid #666;
}
.listing-table th {
  background: #000;
  color: #fff;
}
.listing-table td:first-child,
.listing-table th:first-child {
  border-right: 1px solid #666;
}

```

Now when you refresh the page, you should see all of the Bootstrap glory!

Conclusion

I hope you've enjoyed this tutorial. While we didn't build the next Facebook, I hope that I was nonetheless able to help you get your feet wet in the world of Node-based web apps and offer you some solid takeaways for your next project in the process.

Of course it's hard to cover everything in one tutorial and there are lots of ways you could elaborate on what we've built here. For example, you could check out [our article on deploying Node apps](#) and try your hand at launching it to [Heroku](#) or [now](#). Alternatively, you might augment the CRUD functionality with the ability to delete registrations, or even write a couple of tests to test the app's functionality.

Chapter 2: How to Build a File Upload Form with Express and Dropzone.js

BY LUKAS WHITE

Let's face it, nobody likes forms. Developers don't like building them, designers don't particularly enjoy styling them and users certainly don't like filling them in.

Of all the components that can make up a form, the file control could just be the most frustrating of the lot. A real pain to style, clunky and awkward to use and uploading a file will slow down the submission process of any form.

That's why a plugin to enhance them is always worth a look, and DropzoneJS is just one such option. It will make your file upload controls look better, make them more user-friendly and by using AJAX to upload the file in the background, at the very least make the process *seem* quicker. It also makes it easier to validate files before they even reach your server, providing near-instantaneous feedback to the user.

We're going to take a look at DropzoneJS in some detail; show how to implement it and look at some of the ways in which it can be tweaked and customized. We'll also implement a simple server-side upload mechanism using Node.js.

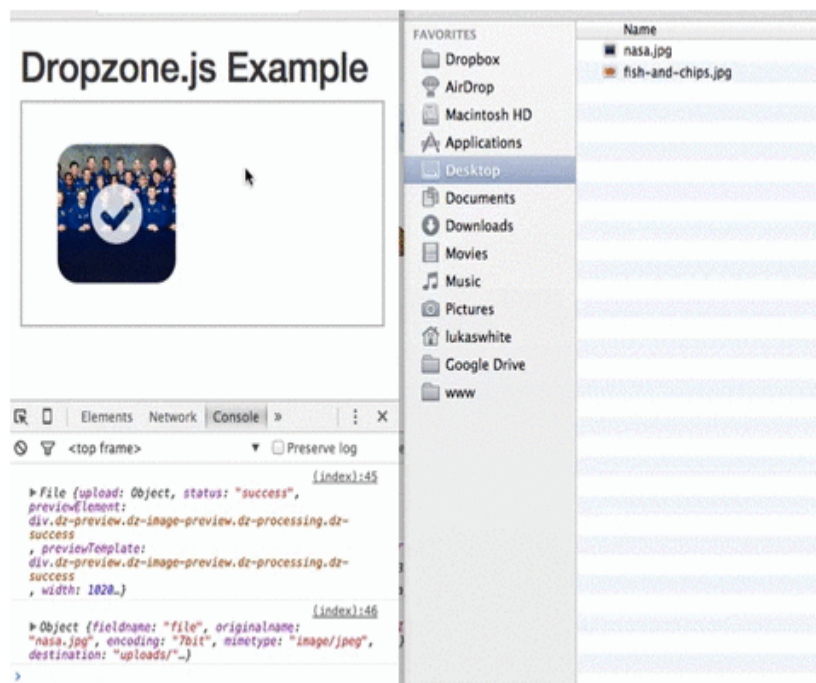
As ever, you can find the code for this tutorial on our GitHub repository.

Introducing DropzoneJS

As the name implies, DropzoneJS allows users to upload files using drag n' drop. Whilst the usability benefits could justifiably be debated, it's an increasingly common approach and one which is in tune with the way a lot of people work with files on their desktop. It's also pretty well supported across major browsers.

DropzoneJS isn't simply a drag n' drop based widget, however; clicking the widget launches the more conventional file chooser dialog approach.

Here's a screenshot of the widget in action:



Alternatively, take a look at this, most minimal of examples.

You can use DropzoneJS for any type of file, though the nice little thumbnail effect makes it ideally suited to uploading images in particular.

FEATURES

To summarize some of the plugin's features and characteristics:

- Can be used **with or without jQuery**
- Drag and drop support
- Generates thumbnail images
- Supports multiple uploads, optionally in parallel
- Includes a progress bar
- Fully themeable
- Extensible file validation support
- Available as an AMD module or RequireJS module
- It comes in at around 33Kb when minified

BROWSER SUPPORT

Taken from the official documentation, browser support is as follows:

- Chrome 7+
- Firefox 4+
- IE 10+
- Opera 12+ (Version 12 for MacOS is disabled because their API is buggy)
- Safari 6+

There are a couple of ways to handle fallbacks for when the plugin isn't fully supported, which we'll look at later.

Installation

The simplest way to install DropzoneJS is via Bower:

```
bower install dropzone
```

Alternatively you can grab it [from Github](#), or simply download the standalone [JavaScript file](#) — though bear in mind you'll also need [the basic styles](#), also available in the Github repo.

There are also third-party packages providing support for [ReactJS](#) and implementing the widget as an [Angular directive](#).

First Steps

If you've used the Bower or download method, make sure you include both the main JS file and the styles (or include them into your application's stylesheet), e.g:

```
<link rel="stylesheet" href="/path/to/dropzone.css">
<script type="text/javascript"
src="/path/to/dropzone.js"></script>
```

Pre-minified versions are also supplied in the package.

If You're Using RequireJS

If you're using RequireJS, use `dropzone-amd-module.js` instead.

Basic Usage

The simplest way to implement the plugin is to attach it to a form, although you can use any HTML such as a `div` tag. Using a form, however, means less options to set — most notably the URL, which is the most important configuration property.

You can initialize it simply by adding the `dropzone` class, for example:

```
<form id="upload-widget" method="post" action="/upload"
class="dropzone">
</form>
```

Technically that's all you need to do, though in most cases you'll want to set some additional options. The format for that is as follows:

```
Dropzone.options.WIDGET_ID = {
  //
};
```

To derive the widget ID for setting the options, take the ID you defined in your HTML and camel-case it. For example, `upload-widget` becomes `uploadWidget`:

```
Dropzone.options.uploadWidget = {
  //
};
```

You can also create an instance programmatically:

```
var uploader = new Dropzone('#upload-widget', options);
```

Next up, we'll look at some of the available configuration options.

Basic Configuration Options

The `url` option defines the target for the upload form, and is the only required parameter. That said, if you're attaching it to a form element then it'll simply use the form's `action` attribute, in which case you don't even need to specify that.

The `method` option sets the HTTP method and again, it will take this from the form element if you use that approach, or else it'll simply default to `POST`, which should suit most scenarios.

The `paramName` option is used to set the name of the parameter for the uploaded file; were you using a file upload form element, it would match the `name` attribute. If you don't include it then it defaults to `file`.

`maxFiles` sets the maximum number of files a user can upload, if it's not set to null.

By default the widget will show a file dialog when it's clicked, though you can use the `clicked` parameter to disable this by setting it to `false`, or alternatively you can provide an HTML element or CSS selector to customize the clickable element.

Those are the basic options, but let's now look at some of the more advanced options.

ENFORCING MAXIMUM FILE SIZE

The `maxFileSize` property determines the maximum file size in megabytes. This defaults to a size of 1000 bytes, but using the `filesizeBase` property, you could set it to another value — for example, 1024 bytes. You may need to tweak this to ensure that your client and server code calculate any limits in precisely the same way.

RESTRICTING TO CERTAIN FILE TYPES

The `acceptFiles` parameter can be used to restrict the type of file you want to accept. This should be in the form of a comma-separated list of MIME-types, although you can also use wildcards.

For example, to only accept images:

```
acceptedFiles: 'image/*',
```

MODIFYING THE SIZE OF THE THUMBNAIL

By default, the thumbnail is generated at 120px by 120px; i.e., it's square. There are a couple of ways you can modify this behavior.

The first is to use the `thumbnailWidth` and/or the `thumbnailHeight` configuration options.

If you set both `thumbnailWidth` and `thumbnailHeight` to `null`, the thumbnail won't be resized at all.

If you want to completely customize the thumbnail generation behavior, you can even override the `resize` function.

One important point about modifying the size of the thumbnail; the `dz-image` class provided by the package sets the thumbnail size in the CSS, so you'll need to modify that accordingly as well.

ADDITIONAL FILE CHECKS

The `accept` option allows you to provide additional checks to determine whether a file is valid, before it gets uploaded. You shouldn't use this to check the number of files (`maxFiles`), file type (`acceptedFiles`), or file size (`maxFileSize`), but you can write custom code to perform other sorts of validation.

You'd use the `accept` option like this:

```
accept: function(file, done) {
  if ( !someCheck() ) {
    return done('This is invalid!');
  }
  return done();
}
```

As you can see it's asynchronous; call `done()` with no arguments and validation passes, or provide an error message and the file will be rejected, displaying the message alongside the file as a popover.

We'll look at a more complex, real-world example later in the article, when we'll look at how to enforce minimum or maximum image sizes.

Sending Additional Headers

Often you'll need to attach additional headers to the uploader's HTTP request.

As an example, one approach to CSRF (Cross Site Request Forgery) protection is to output a token in the view, then have your `POST/PUT/DELETE` endpoints check the request headers for a valid token. Suppose you outputted your token like this:

```
<meta name="csrf-token"
content="CL2tR2J4UHZXcR9BjRtSYOKzSmL8U1zTc7T8d6Jz">
```

Then, you could add this to the configuration:

```
headers: {
  'x-csrf-token':
    document.querySelectorAll('meta[name=csrf-token]')
    [0].getAttributeNode('content').value,
},
```

Alternatively, here's the same example but using jQuery:

```
headers: {
  'x-csrf-token': $('meta[name="csrf-
token"]').attr('content')
},
```

Your server should then verify the `x-csrf-token` header, perhaps using some middleware.

Handling Fallbacks

The simplest way to implement a fallback is to insert a `<div>` into your form containing input controls, setting the class name on the element to `fallback`. For example:

```
<form id="upload-widget" method="post" action="/upload"
class="dropzone">
  <div class="fallback">
    <input name="file" type="file" />
  </div>
</form>
```

Alternatively, you can provide a function to be executed when the browser doesn't support the plugin using the `fallback` configuration parameter.

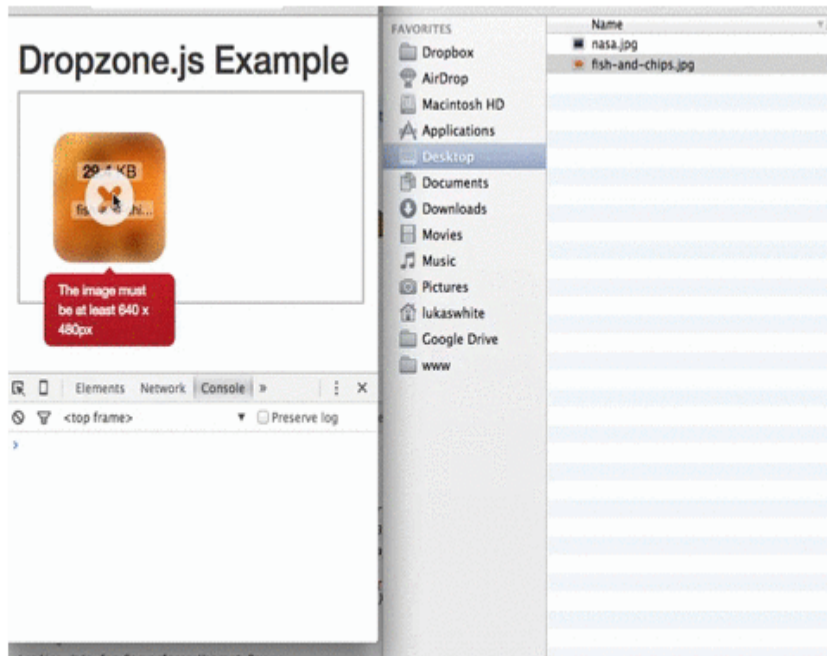
You can force the widget to use the fallback behavior by setting `forceFallback` to `true`, which might help during development.

Handling Errors

You can customize the way the widget handles errors by providing a custom function using the `error` configuration parameter. The first argument is the file, the error message the second and if the error occurred server-side, the third parameter will be an instance of `XMLHttpRequest`.

As always, client-side validation is only half the battle. You must also perform validation on the server. When we implement a simple server-side component later we'll look at the expected format of the error response, which when properly configured

will be displayed in the same way as client-side errors (illustrated below).



Overriding Messages and Translation

There are a number of additional configuration properties which set the various messages displayed by the widget. You can use these to customize the displayed text, or to translate them into another language.

Most notably, `dictDefaultMessage` is used to set the text which appears in the middle of the dropzone, prior to someone selecting a file to upload.

You'll find a complete list of the configurable string values - all of which begin with `dict` - [in the documentation](#).

Events

There are a number of events you can listen to in order to customize or enhance the plugin.

There are two ways to listen to an event. The first is to create a listener within an initialization function:

```
Dropzone.options.uploadWidget = {
  init: function() {
    this.on('success', function( file, resp ){
      ...
    });
  },
  ...
};
```

Or the alternative approach, which is useful if you decide to create the Dropzone instance programmatically:

```
var uploader = new Dropzone('#upload-widget');
uploader.on('success', function( file, resp ){
  ...
});
```

Perhaps the most notable is the `success` event, which is fired when a file has been successfully uploaded. The `success` callback takes two arguments; the first, a file object and the second an instance of `XMLHttpRequest`.

Other useful events include `addedfile` and `removedfile` for when a file has been added or removed from the upload list, `thumbnail` which fires once the thumbnail has been generated and `uploadprogress` which you might use to implement your own progress meter.

There are also a bunch of events which take an event object as a parameter and which you could use to customize the behavior of

the widget itself - drop, dragstart, dragend, dragenter, dragover and dragleave.

You'll find a complete list of events in the relevant section of the [documentation](#).

A More Complex Validation Example: Image Dimensions

Earlier we looked at the asynchronous `accept()` option, which you can use to run checks (validation) on files before they get uploaded.

A common requirement when you're uploading images is to enforce minimum or maximum image dimensions. We can do this with DropzoneJS, although it's slightly more complex.

Although the `accept` callback receives a file object, in order to check the image dimensions we need to wait until the thumbnail has been generated, at which point the dimensions will have been set on the file object. To do so, we need to listen to the `thumbnail` event.

Here's the code; in this example we're checking that the image is at least 640 x 480px before we upload it:

```
init: function() {
  this.on('thumbnail', function(file) {
    if ( file.width < 1024 || file.height < 768 ) {
      file.rejectDimensions();
    }
    else {
      file.acceptDimensions();
    }
  });
},
accept: function(file, done) {
  file.acceptDimensions = done;
  file.rejectDimensions = function() {
    done('The image must be at least 1024 by 768 pixels
```

```
in size');  
    };  
  },
```

A Complete Example

Having gone through the options, events and some slightly more advanced validation, let's look at a complete and relatively comprehensive example. Obviously we're not taking advantage of every available configuration option, since there are so many — making it incredibly flexible.

Here's the HTML for the form:

```
<form id="upload-widget" method="post" action="/upload"  
class="dropzone">  
  <div class="fallback">  
    <input name="file" type="file" />  
  </div>  
</form>
```

If you're implementing CSRF protection, you may want to add something like this to your layouts:

```
<head>  
  <!-- -->  
  <meta name="csrf-token" content="XYZ123">  
</head>
```

Now the JavaScript - notice we're not using jQuery!

```
Dropzone.options.uploadWidget = {  
  paramName: 'file',  
  maxFileSize: 2, // MB  
  maxFiles: 1,  
  dictDefaultMessage: 'Drag an image here to upload, or  
click to select one',  
  headers: {  
    'x-csrf-token':  
document.querySelectorAll('meta[name=csrf-token]')  
[0].getAttributeNode('content').value,  
  },  
  acceptedFiles: 'image/*',
```

```

init: function() {
  this.on('success', function( file, resp ){
    console.log( file );
    console.log( resp );
  });
  this.on('thumbnail', function(file) {
    if ( file.width < 640 || file.height < 480 ) {
      file.rejectDimensions();
    }
    else {
      file.acceptDimensions();
    }
  });
},
accept: function(file, done) {
  file.acceptDimensions = done;
  file.rejectDimensions = function() {
    done('The image must be at least 640 x 480px')
  };
}
};

```

A reminder that you'll find the code for this example on our [GitHub repository](#).

Hopefully, this is enough to get you started for most scenarios; check out the [full documentation](#) if you need anything more complex.

Theming

There are a number of ways to customize the look and feel of the widget, and indeed it's possible to completely transform the way it looks.

For an example of just how customizable the appearance is, here is [a demo](#) of the widget tweaked to look and feel exactly like the [jQuery File Upload](#) widget using Bootstrap.

Obviously the simplest way to change the widget's appearance is to use CSS. The widget has a class of `dropzone` and its component elements have classes prefixed with `dz-`; for example `dz-clickable` for the clickable area inside the

dropzone, dz-message for the caption, dz-preview / dz-image-preview for wrapping the previews of each of the uploaded files, and so on. Take a look at the `dropzone.css` file for reference.

You may also wish to apply styles to the hover state; that is, when the user hovers a file over the dropzone before releasing their mouse button to initiate the upload. You can do this by styling the `dz-drag-hover` class, which gets added automatically by the plugin.

Beyond CSS tweaks, you can also customize the HTML which makes up the previews by setting the `previewTemplate` configuration property. Here's what the default preview template looks like:

```
<div class="dz-preview dz-file-preview">
  <div class="dz-image">
    <img data-dz-thumbnail />
  </div>
  <div class="dz-details">
    <div class="dz-size">
      <span data-dz-size></span>
    </div>
    <div class="dz-filename">
      <span data-dz-name></span>
    </div>
  </div>
  <div class="dz-progress">
    <span class="dz-upload" data-dz-uploadprogress>
</span>
</div>
  <div class="dz-error-message">
    <span data-dz-errormessage></span>
  </div>
  <div class="dz-success-mark">
    <svg>REMOVED FOR BREVITY</svg>
  </div>
  <div class="dz-error-mark">
    <svg>REMOVED FOR BREVITY</svg>
  </div>
</div>
```

As you can see, you get complete control over how files are rendered once they've been queued for upload, as well as

success and failure states.

That concludes the section on using the DropzoneJS plugin. To round up, let's look at how to get it working with server-side code.

A Simple Server-Side Upload Handler with Node.js and Express

Naturally you can use any server-side technology for handling uploaded files. In order to demonstrate how to integrate your server with the plugin, we'll build a very simple example using Node.js and Express.

To handle the uploaded file itself we'll use multer, a package which provides some Express middleware that makes it really easy. In fact, this easy:

```
var upload = multer( { dest: 'uploads/' } );

app.post( '/upload', upload.single( 'file' ), function(
  req, res, next ) {
  // Metadata about the uploaded file can now be found in
  req.file
});
```

Before we continue the implementation, the most obvious question to ask when dealing with a plugin like DropzoneJS which makes requests for you behind the scenes is: “what sort of responses does it expect?”

Handling Upload Success

If the upload process is successful, the only requirement as far as your server-side code is concerned, is to return a 2xx response code. The content and format of your response is

entirely up to you, and will probably depend on how you're using it; for example you might return a JSON object which contains a path to the uploaded file, or the path to an automatically generated thumbnail. For the purposes of this example we'll simply return the contents of the file object - i.e. a bunch of metadata - provided by Multer:

```
return res.status( 200 ).send( req.file );
```

The response will look something like this:

```
{ fieldname: 'file',  
  originalname: 'myfile.jpg',  
  encoding: '7bit',  
  mimetype: 'image/jpeg',  
  destination: 'uploads/',  
  filename: 'fbcc2ddb0dd11858427d7f0bb2273f5',  
  path: 'uploads/fbcc2ddb0dd11858427d7f0bb2273f5',  
  size: 15458 }
```

Handling Upload Errors

If your response is in JSON format - that is to say, your response type is set to `application/json` - then DropzoneJS default error plugin expects the response to look like this:

```
{  
  error: 'The error message'  
}
```

If you aren't using JSON, it'll simply use the response body, for example:

```
return res.status( 422 ).send( 'The error message' );
```

Let's demonstrate this by performing a couple of validation checks on the uploaded file. We'll simply duplicate the checks

we performed on the client — remember, client-side validation is never sufficient on its own.

To verify that the file is an image, we'll simply check that the **Mime-type** starts with `image/`. ES6's

`String.prototype.startsWith()` is ideal for this, but let's install a polyfill for it:

```
npm install string.prototype.startswith --save
```

Here's how we might run that check and, if it fails, return the error in the format which Dropzone's default error handler expects:

```
if ( !req.file.mimetype.startsWith( 'image/' ) ) {  
  return res.status( 422 ).json( {  
    error : 'The uploaded file must be an image'  
  } );  
}
```

Status Codes

I'm using HTTP Status Code 422, Unprocessable Entity here for validation failure, but 400 Bad Request is just as valid; indeed anything outside of the 2xx range will cause the plugin to report the error.

Let's also check that the image is of a certain size; the image-size package makes it really straightforward to get the dimensions of an image. You can use it asynchronously or synchronously; we'll use the latter to keep things simple:

```
var dimensions = sizeOf( req.file.path );  
  
if ( ( dimensions.width < 640 ) || ( dimensions.height < 480 ) ) {  
  return res.status( 422 ).json( {  
    error : 'The image must be at least 640 x 480px'  
  } );  
}
```

Let's put all of it together in a complete (mini) application:

```
var express = require( 'express' );
var multer  = require( 'multer' );
var upload  = multer( { dest: 'uploads/' } );
var sizeof  = require( 'image-size' );
var exphbs  = require( 'express-handlebars' );
require( 'string.prototype.startswith' );

var app = express();

app.use( express.static( __dirname + '/bower_components'
) );

app.engine( '.hbs', exphbs( { extname: '.hbs' } ) );
app.set('view engine', '.hbs');

app.get( '/', function( req, res, next ){
    return res.render( 'index' );
});

app.post( '/upload', upload.single( 'file' ), function(
req, res, next ) {

    if ( !req.file.mimetype.startsWith( 'image/' ) ) {
        return res.status( 422 ).json( {
            error : 'The uploaded file must be an image'
        } );
    }

    var dimensions = sizeof( req.file.path );

    if ( ( dimensions.width < 640 ) || ( dimensions.height
< 480 ) ) {
        return res.status( 422 ).json( {
            error : 'The image must be at least 640 x 480px'
        } );
    }

    return res.status( 200 ).send( req.file );
});

app.listen( 8080, function() {
    console.log( 'Express server listening on port 8080' );
});
```

CSRF Protection

For brevity, this server-side code doesn't implement CSRF protection; you might want to look at a [package like CSURF](#) for that.

You'll find this code, along with the supporting assets such as the view, in the [accompanying repository](#).

Summary

DropzoneJS is a slick, powerful and highly customizable JavaScript plugin for super-charging your file upload controls and performing AJAX uploads. In this article we've taken a look at a number of the available options, at events and how to go about customizing the plugin. There's a lot more to it than can reasonably be covered in one article, so check out the [official website](#) if you'd like to know more — but hopefully this is enough to get you started.

We've also built a really simple server-side component to handle file uploads, demonstrating how to get the two working in tandem.

Chapter 3: How to Build and Structure a Node.js MVC Application

BY JAMES KOLCE

In a non-trivial application, the architecture is as important as the quality of the code itself. We can have well-written pieces of code, but if we don't have a good organization, we'll have a hard time as the complexity increases. There's no need to wait until the project is half-way done to start thinking about the architecture. The best time is before starting, using our goals as beacons for our choices.

Node.js doesn't have a de facto framework with strong opinions on architecture and code organization in the same way that Ruby has the Rails framework, for example. As such, it can be difficult to get started with building full web applications with Node.

In this chapter, we're going to build the basic functionality of a note-taking app using the MVC architecture. To accomplish this, 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, see, 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.

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 Commands

If you're not familiar with these commands, checkout 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 hoek
```

This command will download the latest version of Hapi.js and add it to our `package.json` file as a dependency. It will also download the Hoek utility library that will help us write shorter error handlers, among other things.

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:

```
'use strict';

const Hapi = require('hapi');
const Hoek = require('hoek');
const Settings = require('./settings');

const server = new Hapi.Server();
server.connection({ port: Settings.port });

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

```
server.start((err) => {  
  Hoek.assert(!err, err);  
  
  console.log(`Server running at: ${server.info.uri}`);  
});
```

This is going to be the foundation of our application.

First, we indicate that we're going to use strict mode, which is a common practice when using the Hapi.js framework.

Next, we include our dependencies and instantiate a new server object where we set the connection port to 3000 (the port can be any number above 1023 and below 65535.)

Our 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, and a handler, which is a function that will process the HTTP request. The handler function can take two arguments: `request` and `reply`. The first one contains information about the HTTP call, and the second will provide us with methods to handle our response to that call.

Finally, we start our server with the `server.start` method. As you can see, we can use Hoek to improve our error handling, making it shorter. This is completely optional, so feel free to omit it in your code, just be sure to handle any errors.

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 on 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`.

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 the found values to the environment. You can find an example in the [dotenv repository](#).

If You Use 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.

Our `settings.js` file will look like this:

```
// This will load our .env file and add the values to
process.env,
// IMPORTANT: Omit this line if you don't want to use
this functionality
```



```
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:'
    }
  },
  production: {
    db: {
      dialect: 'sqlite',
      storage: 'db/database.sqlite'
    }
  }
};
```

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

```
node server.js
```

Ensure Your Installation is Up-to-date

This project was tested on Node v6. If you get any errors, ensure you have an updated installation.

Defining the Routes

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:

```
'use strict';

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

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

```
{
  method: 'GET',
  path: '/',
  handler: (request, reply) => {
    reply('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.

Then, we 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 next to the previous route:

```
{
  method: 'POST',
  path: '/note',
  handler: (request, reply) => {
    reply('New note');
  },
  config: {
```

```

    description: 'Adds a new note'
  },
  {
    method: 'GET',
    path: '/note/{slug}',
    handler: (request, reply) => {
      reply('This is a note');
    },
    config: {
      description: 'Gets the content of a note'
    }
  },
  {
    method: 'PUT',
    path: '/note/{slug}',
    handler: (request, reply) => {
      reply('Edit a note');
    },
    config: {
      description: 'Updates the selected note'
    }
  },
  {
    method: 'GET',
    path: '/note/{slug}/delete',
    handler: (request, reply) => {
      reply('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.

Strict 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 on 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);
```

Building the Models

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 [Sequelize.js](#) and [SQLite](#). You can install and include them as dependencies by executing the following command:

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

Now create a `models` directory inside `lib/` with a file called `index.js` which is going to contain the database and `Sequelize.js` setup, and include the following content:

```
'use strict';

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` as part of 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:

```
'use strict';

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

```

module.exports = (sequelize, DataTypes) => {
  let Note = sequelize.define('Note', {
    date: {
      type: DataTypes.DATE,
      get: function () {
        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 be first passed through the `Moment` utility to be formatted in a more readable way (`MMMM Do, YYYY`).

Ahem, Excuse Me ...

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.

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:

```
// Import the index.js file inside the models directory
const Models = require('./lib/models/');
```

Next, replace the following code block:

```
server.start((err) => {
  Hoek.assert(!err, err);
  console.log(`Server running at: ${server.info.uri}`);
});
```

with this one:

```
Models.sequelize.sync().then(() => {
  server.start((err) => {
    Hoek.assert(!err, err);
    console.log(`Server running at: ${server.info.uri}`);
  });
});
```

This code is going to synchronize the models to our database and, once that is done, the server will be started.

Building the controllers

Controllers are functions that accept the request and reply objects from Hapi.js. The `request` object contains information about the requested resource and we use `reply` 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 `home.js` file inside the `lib/controllers` directory with the following content:

```
'use strict';

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

module.exports = (request, reply) => {
  Models.Note
    .findAll({
      order: [['date', 'DESC']]
    })
    .then((result) => {
      reply({
        data: {
          notes: result
        },
        page: 'Home-Notes Board',
        description: 'Welcome to my Notes Board'
      });
    });
};
```

First, we get all the notes in our database using the `findAll` method of our model. This function will return a promise and, if it resolves, we will get 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](#).

Once we have the home controller, we can edit our `routes.js` file. First, we import the module at the top of the file, next to the

Path module import:

```
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'
  }
},
```

BOILERPLATE OF THE NOTE CONTROLLER

Since we're going to identify our notes with a slug, we can generate one using the title of the note and the `slug` library, so let's install it and include it as a dependency with the following command:

```
npm install -s slug
```

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:

```
'use strict';

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

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: (request, reply) => {
  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
    })
    .then((result) => {
      // We're going to generate a view later, but for
now lets just return the result.
      reply(result);
    });
},
```

Once the note is created, we will get back the note data and send it to the client as JSON using the `reply` function.

For now, we just return the result, but 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 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: (request, reply) => {
  Models.Note
    .findOne({
      where: {
        slug: request.params.slug
      }
    })
    .then((result) => {
      reply(result);
    });
},
```

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 build 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 is the note that we’re going to update.

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

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

  Models.Note
    .update(values, options)
    .then(() => {
      Models.Note
        .findOne(options)
        .then((result) => {
```

```
        reply(result);
      });
    });
  },
```

After updating our data, since our database won't return the updated note, we can find the modified note again to 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 of the Hapi.js reply object.

```
delete: (request, reply) => {
  Models.Note
    .destroy({
      where: {
        slug: request.params.slug
      }
    })
    .then(() => reply.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 handler 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

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

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 to 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.

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='https://code.jquery.com/jquery-
3.1.1.min.js' integrity='sha256-
hVVnYaiADRT02PzUGmuLJr8BLUSjGIZsDYGmIJLv2b8='
crossorigin='anonymous')
    script(src='/scripts/jquery.modal.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, at the bottom of the page, three JS files, 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 those packages and put them in a `static/public/scripts/` directory. We're going to make them public 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
      ul
        // This will show a modal window with a form to
        // send new notes
        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 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') Write here the
        content of the new note...
      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 in the home page but with the current note information already filled, 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 the Ajax functionality of jQuery. 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:

```

$('#note-form').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);
    } 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 on 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.

ADDING SUPPORT FOR VIEWS ON THE SERVER

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 at the top of the file, since we're using it in our code to indicate the path of our views.

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

Now, replace the `server.route(Routes) ;` line with the following code block:

```

server.register([
  require('vision')
], (err) => {
  Hoek.assert(!err, err);

  // View settings
  server.views({
    engines: { pug: require('pug') },
    path: Path.join(__dirname, 'lib/views'),
    compileOptions: {
      pretty: false
    }
  });
});

```

```
    },
    isCached: Settings.env === 'production'
  });

  // Add routes
  server.route(Routes);
});
```

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. At the end of the code block, we add again our routes.

This will make work our views 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 `reply(result);` line with the following:

```
reply.view('home', {
  data: {
    notes: result
  },
  page: 'Home-Notes Board',
  description: 'Welcome to my Notes Board'
});
```

After registering the Vision plugin, we now have a `view` method available on the `reply` object, 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 FUNCTION

Right now, every time we create a note we get a JSON object from the server to the client. But since we're doing this process with 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. Replace the line `reply(result);` with the following code block:

```
// Generate a new note with the 'result' data
const newNote = Pug.renderFile(
  Path.join(__dirname, '../views/components/note.pug'),
  {
    note: result
  }
);

reply(newNote);
```

We use the `renderFile` method from Pug to render the note template with the data we just received from our model.

SETTING THE NOTE VIEW: READ FUNCTION

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 `reply(result);` line with this:

```
reply.view('note', {
  note: result,
  page: `${result.title}-Notes Board`,
  description: result.description
});
```

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

SETTING THE NOTE VIEW: UPDATE FUNCTION

Every time we update a note, we will reply similarly to when we create new notes. Replace the `reply(result);` line with the following code:

```
// Generate a new note with the updated data
const updatedNote = Pug.renderFile(
  Path.join(__dirname, '../views/components/note.pug'),
  {
    note: result
  }
);

reply(updatedNote);
```

No View for the Delete Function

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 provided 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 the `server.register` function inside the `server.js` file, import the Inert plugin and register it with Hapi like this:

```
server.register([
  require('vision'),
  require('inert')
], (err) => {
```

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

```
{
  // Static files
  method: 'GET',
  path: '/{param*}',
  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](#).

More Files in the Repo

Check the [Github repository](#) for the rest of the [static files](#), like the [main stylesheet](#).

Conclusion

At this point, we have a very basic Hapi.js application using the MVC architecture. Although there are still things that we should take care of before putting our application in production (e.g. input validation, error handling, error pages, etc.) this should work as a foundation to learn and build your own applications.

If you would like to take this example a bit further, after finishing all the small details (not related the architecture) to make this a robust application, you could implement an authentication system so only registered users are able to publish and edit notes. But your imagination is the limit, so feel free to fork the application repository and experiment with your ideas.

Chapter 4: User Authentication with the MEAN Stack

BY SIMON HOLMES & JEREMY WILKEN

In this article, we're going to look at managing user authentication in the MEAN stack. We'll use the most common MEAN architecture of having an Angular single-page app using a REST API built with Node, Express and MongoDB.

When thinking about user authentication, we need to tackle the following things:

1. let a user register
2. save their data, but never directly store their password
3. let a returning user log in
4. keep a logged in user's session alive between page visits
5. have some pages that can only be seen by logged in users
6. change output to the screen depending on logged in status (e.g. a "login" button or a "my profile" button).

Before we dive into the code, let's take a few minutes for a high-level look at how authentication is going to work in the MEAN stack.

The MEAN Stack Authentication Flow

So what does authentication look like in the MEAN stack?

Still keeping this at a high level, these are the components of the flow:

- user data is stored in MongoDB, with the passwords hashed
- CRUD functions are built in an Express API — Create (register), Read (login, get profile), Update, Delete
- an Angular application calls the API and deals with the responses
- the Express API generates a JSON Web Token (JWT, pronounced “Jot”) upon registration or login, and passes this to the Angular application
- the Angular application stores the JWT in order to maintain the user’s session
- the Angular application checks the validity of the JWT when displaying protected views
- the Angular application passes the JWT back to Express when calling protected API routes.

JWTs are preferred over cookies for maintaining the session state in the browser. Cookies are better for maintaining state when using a server-side application.

The Example Application

The code for this article is available on [GitHub](#). To run the application, you’ll need to have [Node.js installed](#), along with MongoDB. (For instructions on how to install, please refer to [Mongo’s official documentation — Windows, Linux, macOS](#)).

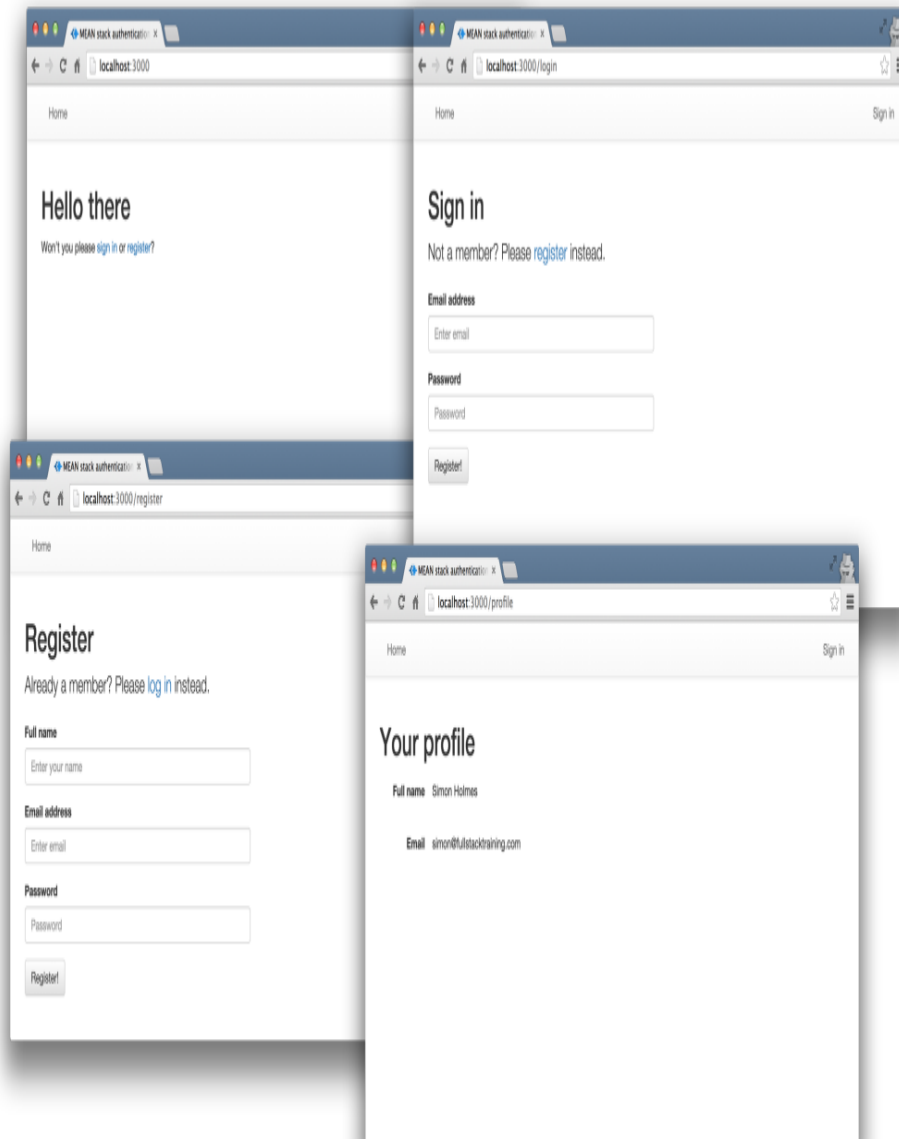
THE ANGULAR APP

To keep the example in this article simple, we’ll start with an Angular app with four pages:

1. home page
2. register page

3. login page
4. profile page

The pages are pretty basic and look like this to start with:



The profile page will only be accessible to authenticated users. All the files for the Angular app are in a folder inside the Angular CLI app called `/client`.

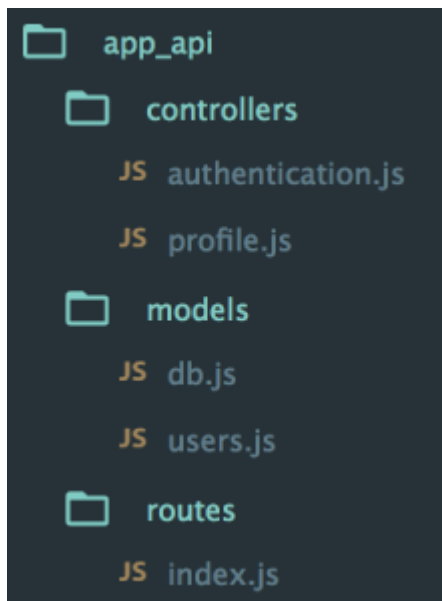
We'll use the Angular CLI for building and running the local server. If you're unfamiliar with the Angular CLI, refer to the [Angular 2 Tutorial: Create a CRUD App with Angular CLI](#) to get started.

THE REST API

We'll also start off with the skeleton of a REST API built with Node, Express and MongoDB, using [Mongoose](#) to manage the schemas. This API has three routes:

1. `/api/register` (POST) — to handle new users registering
2. `/api/login` (POST) — to handle returning users logging in
3. `/api/profile/USERID` (GET) — to return profile details when given a `USERID`.

The code for the API is all held in another folder inside the Express app, called `api`. This holds the routes, controllers and model, and is organized like this:



At this starting point, each of the controllers simply responds with a confirmation, like this:

```
module.exports.register = function(req, res) {  
  console.log("Registering user: " + req.body.email);  
  res.status(200);  
  res.json({  
    "message" : "User registered: " + req.body.email  
  });  
};
```

Okay, let's get on with the code, starting with the database.

Creating the MongoDB Data Schema with Mongoose

There's a simple user schema defined in `/api/models/users.js`. It defines the need for an email address, a name, a hash and a salt. The hash and salt will be used instead of saving a password. The `email` is set to unique as we'll use it for the login credentials. Here's the schema:

```
var userSchema = new mongoose.Schema({  
  email: {  
    type: String,  
    unique: true,  
    required: true  
  },  
  name: {  
    type: String,  
    required: true  
  },  
  hash: String,  
  salt: String  
});
```

MANAGING THE PASSWORD WITHOUT SAVING IT

Saving user passwords is a big no-no. Should a hacker get a copy of your database, you want to make sure they can't use it to log in to accounts. This is where the hash and salt come in.

The salt is a string of characters unique to each user. The hash is created by combining the password provided by the user and the salt, and then applying one-way encryption. As the hash can't be decrypted, the only way to authenticate a user is to take the password, combine it with the salt and encrypt it again. If the output of this matches the hash, the password must have been correct.

To do the setting and the checking of the password, we can use Mongoose schema methods. These are essentially functions that you add to the schema. They'll both make use of the Node.js `crypto` module.

At the top of the `users.js` model file, require `crypto` so that we can use it:

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

Nothing needs installing, as `crypto` ships as part of Node. `Crypto` itself has several methods; we're interested in `randomBytes` to create the random salt and `pbkdf2Sync` to create the hash (there's much more about `Crypto` in the [Node.js API docs](#)).

Setting the Password

To save the reference to the password, we can create a new method called `setPassword` on the `userSchema` schema that accepts a password parameter. The method will then use `crypto.randomBytes` to set the salt, and `crypto.pbkdf2Sync` to set the hash:

```
userSchema.methods.setPassword = function(password) {  
  this.salt = crypto.randomBytes(16).toString('hex');  
  this.hash = crypto.pbkdf2Sync(password, this.salt,  
    1000, 64, 'sha512').toString('hex');  
};
```

We'll use this method when creating a user. Instead of saving the password to a `password` path, we'll be able to pass it to the `setPassword` function to set the `salt` and `hash` paths in the user document.

Checking the Password

Checking the password is a similar process, but we already have the salt from the Mongoose model. This time we just want to encrypt the salt and the password and see if the output matches the stored hash.

Add another new method to the `users.js` model file, called `validatePassword`:

```
userSchema.methods.validatePassword = function(password) {  
  var hash = crypto.pbkdf2Sync(password, this.salt, 1000,  
    64, 'sha512').toString('hex');  
  return this.hash === hash;  
};
```

GENERATING A JSON WEB TOKEN (JWT)

One more thing the Mongoose model needs to be able to do is generate a JWT, so that the API can send it out as a response. A Mongoose method is ideal here too, as it means we can keep the code in one place and call it whenever needed. We'll need to call it when a user registers and when a user logs in.

To create the JWT, we'll use a module called jsonwebtoken which needs to be installed in the application, so run this on the command line:

```
npm install jsonwebtoken --save
```

Then require this in the `users.js` model file:

```
var jwt = require('jsonwebtoken');
```

This module exposes a `sign` method that we can use to create a JWT, simply passing it the data we want to include in the token, plus a secret that the hashing algorithm will use. The data should be sent as a JavaScript object, and include an expiry date in an `exp` property.

Adding a `generateJwt` method to `userSchema` in order to return a JWT looks like this:

```
userSchema.methods.generateJwt = function() {  
  var expiry = new Date();  
  expiry.setDate(expiry.getDate() + 7);  
  
  return jwt.sign({  
    _id: this._id,  
    email: this.email,  
    name: this.name,  
    exp: parseInt(expiry.getTime() / 1000),  
  }, "MY_SECRET"); // DO NOT KEEP YOUR SECRET IN THE  
  CODE!  
};
```

Keep the Secret Safe!

It's important that your secret is kept safe: only the originating server should know what it is. It's best practice to set the secret as an environment variable, and not have it in the source code, especially if your code is stored in version control somewhere.

That's everything we need to do with the database.

Set Up Passport to Handle the Express Authentication

Passport is a Node module that simplifies the process of handling authentication in Express. It provides a common gateway to work with many different authentication

“strategies”, such as logging in with Facebook, Twitter or Oauth. The strategy we’ll use is called “local”, as it uses a username and password stored locally.

To use Passport, first install it and the strategy, saving them in `package.json`:

```
npm install passport --save
npm install passport-local --save
```

CONFIGURE PASSPORT

Inside the `api` folder, create a new folder `config` and create a file in there called `passport.js`. This is where we define the strategy.

Before defining the strategy, this file needs to require Passport, the strategy, Mongoose and the `User` model:

```
var passport = require('passport');
var LocalStrategy = require('passport-local').Strategy;
var mongoose = require('mongoose');
var User = mongoose.model('User');
```

For a local strategy, we essentially just need to write a Mongoose query on the `User` model. This query should find a user with the email address specified, and then call the `validatePassword` method to see if the hashes match. Pretty simple.

There’s just one curiosity of Passport to deal with. Internally, the local strategy for Passport expects two pieces of data called `username` and `password`. However, we’re using `email` as our unique identifier, not `username`. This can be configured in an options object with a `usernameField` property in the strategy definition. After that, it’s over to the Mongoose query.

So all in, the strategy definition will look like this:

```
passport.use(new LocalStrategy({
  usernameField: 'email'
}),
function(username, password, done) {
  User.findOne({ email: username }, function (err,
user) {
  if (err) { return done(err); }
  // Return if user not found in database
  if (!user) {
    return done(null, false, {
      message: 'User not found'
    });
  }
  // Return if password is wrong
  if (!user.validPassword(password)) {
    return done(null, false, {
      message: 'Password is wrong'
    });
  }
  // If credentials are correct, return the user
  object
  return done(null, user);
});
}
```

Note how the `validPassword` schema method is called directly on the `user` instance.

Now Passport just needs to be added to the application. So in `app.js` we need to require the Passport module, require the Passport config and initialize Passport as middleware. The placement of all of these items inside `app.js` is quite important, as they need to fit into a certain sequence.

The Passport module should be required at the top of the file with the other general `require` statements:

```
var express = require('express');
var path = require('path');
var favicon = require('serve-favicon');
var logger = require('morgan');
var cookieParser = require('cookie-parser');
var bodyParser = require('body-parser');
var passport = require('passport');
```

The config should be required *after* the model is required, as the config references the model.

```
require('./api/models/db');  
require('./api/config/passport');
```

Finally, Passport should be initialized as Express middleware just before the API routes are added, as these routes are the first time that Passport will be used.

```
app.use(passport.initialize());  
app.use('/api', routesApi);
```

We've now got the schema and Passport set up. Next, it's time to put these to use in the routes and controllers of the API.

Configure API Endpoints

With the API we've got two things to do:

1. make the controllers functional
2. secure the `/api/profile` route so that only authenticated users can access it.

CODE THE REGISTER AND LOGIN API CONTROLLERS

In the example app the register and login controllers are in `/api/controllers/authentication.js`. In order for the controllers to work, the file needs to require Passport, Mongoose and the user model:

```
var passport = require('passport');  
var mongoose = require('mongoose');  
var User = mongoose.model('User');
```

The Register API Controller

The register controller needs to do the following:

1. take the data from the submitted form and create a new Mongoose model instance
2. call the `setPassword` method we created earlier to add the salt and the hash to the instance
3. save the instance as a record to the database
4. generate a JWT
5. send the JWT inside the JSON response.

In code, all of that looks like this:

```
module.exports.register = function(req, res) {
  var user = new User();

  user.name = req.body.name;
  user.email = req.body.email;

  user.setPassword(req.body.password);

  user.save(function(err) {
    var token;
    token = user.generateJwt();
    res.status(200);
    res.json({
      "token" : token
    });
  });
};
```

This makes use of the `setPassword` and `generateJwt` methods we created in the Mongoose schema definition. See how having that code in the schema makes this controller really easy to read and understand.

Don't forget that, in reality, this code would have a number of error traps, validating form inputs and catching errors in the `save` function. They're omitted here to highlight the main functionality of the code.

The Login API Controller

The login controller hands over pretty much all control to Passport, although you could (and should) add some validation beforehand to check that the required fields have been sent.

For Passport to do its magic and run the strategy defined in the config, we need to call the `authenticate` method as shown below. This method will call a callback with three possible parameters `err`, `user` and `info`. If `user` is defined, it can be used to generate a JWT to be returned to the browser:

```
module.exports.login = function(req, res) {

  passport.authenticate('local', function(err, user,
info){
    var token;

    // If Passport throws/catches an error
    if (err) {
      res.status(404).json(err);
      return;
    }

    // If a user is found
    if(user){
      token = user.generateJwt();
      res.status(200);
      res.json({
        "token" : token
      });
    } else {
      // If user is not found
      res.status(401).json(info);
    }
  })(req, res);

};
```

SECURING AN API ROUTE

The final thing to do in the back end is make sure that only authenticated users can access the `/api/profile` route. The way to validate a request is to ensure that the JWT sent with it is genuine, by using the secret again. This is why you should keep it a secret and not in the code.

Configuring the Route Authentication

First we need to install a piece of middleware called `express-jwt`:

```
npm install express-jwt --save
```

Then we need to require it and configure it in the file where the routes are defined. In the sample application, this is `/api/routes/index.js`. Configuration is a case of telling it the secret, and — optionally — the name of the property to create on the `req` object that will hold the JWT. We'll be able to use this property inside the controller associated with the route. The default name for the property is `user`, but this is the name of an instance of our Mongoose `User` model, so we'll set it to `payload` to avoid confusion:

```
var jwt = require('express-jwt');  
var auth = jwt({  
  secret: 'MY_SECRET',  
  userProperty: 'payload'  
});
```

Again, *don't keep the secret in the code!*

Applying the Route Authentication

To apply this middleware, simply reference the function in the middle of the route to be protected, like this:

```
router.get('/profile', auth, ctrlProfile.profileRead);
```

If someone tries to access that route now without a valid JWT, the middleware will throw an error. To make sure our API plays nicely, catch this error and return a 401 response by adding the following into the error handlers section of the main `app.js` file:

```
// error handlers
// Catch unauthorised errors
app.use(function (err, req, res, next) {
  if (err.name === 'UnauthorizedError') {
    res.status(401);
    res.json({"message" : err.name + ": " +
err.message});
  }
});
```

Using the Route Authentication

In this example, we only want people to be able to view their own profiles, so we get the user ID from the JWT and use it in a Mongoose query.

The controller for this route is in [/api/controllers/profile.js](#). The entire contents of this file look like this:

```
var mongoose = require('mongoose');
var User = mongoose.model('User');

module.exports.profileRead = function(req, res) {

  // If no user ID exists in the JWT return a 401
  if (!req.payload._id) {
    res.status(401).json({
      "message" : "UnauthorizedError: private profile"
    });
  } else {
    // Otherwise continue
    User
      .findById(req.payload._id)
      .exec(function(err, user) {
        res.status(200).json(user);
      });
  }
};
```

Naturally, this should be fleshed out with some more error trapping — for example, if the user isn't found — but this snippet is kept brief to demonstrate the key points of the approach.

That's it for the back end. The database is configured, we have API endpoints for registering and logging in that generate and return a JWT, and also a protected route. On to the front end!

Create Angular Authentication Service

Most of the work in the front end can be put into an Angular service, creating methods to manage:

- saving the JWT in local storage
- reading the JWT from local storage
- deleting the JWT from local storage
- calling the register and login API endpoints
- checking whether a user is currently logged in
- getting the details of the logged-in user from the JWT.

We'll need to create a new service called `AuthenticationService`. With the CLI, this can be done by running `ng generate service authentication`, and making sure it's listed in the app module providers. In the example app, this is in the file `/client/src/app/authentication.service.ts`.

LOCAL STORAGE: SAVING, READING AND DELETING A JWT

To keep a user logged in between visits, we use `localStorage` in the browser to save the JWT. An alternative is to use `sessionStorage`, which will only keep the token during the current browser session.

First, we want to create a few interfaces to handle the data types. This is useful for type checking our application. The

profile returns an object formatted as `UserDetails`, and the login and register endpoints expect a `TokenPayload` during the request and return a `TokenResponse` object:

```
export interface UserDetails {  
  _id: string;  
  email: string;  
  name: string;  
  exp: number;  
  iat: number;  
}  
  
interface TokenResponse {  
  token: string;  
}  
  
export interface TokenPayload {  
  email: string;  
  password: string;  
  name?: string;  
}
```

This service uses the `HttpClient` service from Angular to make HTTP requests to our server application (which we'll use in a moment) and the `Router` service to navigate programmatically. We must inject them into our service constructor.

Then we define four methods that interact with the JWT token. We implement `saveToken` to handle storing the token into `localStorage` and onto the `token` property, a `getToken` method to retrieve the token from `localStorage` or from the `token` property, and a `logout` function that removes the JWT token from memory and redirects to the home page.

It's important to note that this code doesn't run if you're using server-side rendering, because APIs like `localStorage` and `window.atob` aren't available, and there are details about solutions to address server-side rendering in the Angular documentation.

```

import { Injectable } from '@angular/core';
import { HttpClient } from '@angular/common/http';
import { Observable } from 'rxjs/Observable';
import { map } from 'rxjs/operators/map';
import { Router } from '@angular/router';

// Interfaces here

@Injectable()
export class AuthenticationService {
  private token: string;

  constructor(private http: HttpClient, private router:
Router) {}

  private saveToken(token: string): void {
    localStorage.setItem('mean-token', token);
    this.token = token;
  }

  private getToken(): string {
    if (!this.token) {
      this.token = localStorage.getItem('mean-token');
    }
    return this.token;
  }

  public logout(): void {
    this.token = '';
    window.localStorage.removeItem('mean-token');
    this.router.navigateByUrl('/');
  }
}

```

Now let's add a method to check for this token — and the validity of the token — to find out if the visitor is logged in.

Getting Data from a JWT

When we set the data for the JWT (in the `generateJwt` Mongoose method) we included the expiry date in an `exp` property. But if you look at a JWT, it seems to be a random string, like this following example:

```

eyJ0eXAiOiJKV1QiLCJhbGciOiJIUzI1NiJ9.eyJfaWQiOiI1NWQ0MjNjMTUxMzcxMmNmZS3YTRkYTciLCJlbWFpbCI6InNpbW9uQGZ1bGxzZGFja3RyYWluaW5nLmNvbSIsIm5hbWUiOiJTaW1vbiBib2xtZXMiLCJleHAiOiJEOjE0NDAlNzA5NDUsImhhdCI6MTQzOTk2NjE0NX0.jS50GlmolxLoKrA_24L
DKaW3vNaY94Y9EqYAFvsTiLg

```

So how do you read a JWT?

A JWT is actually made up of three separate strings, separated by a dot `.`. These three parts are:

1. *Header* — an encoded JSON object containing the type and the hashing algorithm used
2. *Payload* — an encoded JSON object containing the data, the real body of the token
3. *Signature* — an encrypted hash of the header and payload, using the “secret” set on the server.

It’s the second part we’re interested in here — the payload. Note that this is *encoded* rather than encrypted, meaning that we can *decode* it.

There’s a function called `atob()` that’s native to modern browsers, and which will decode a Base64 string like this.

So we need to get the second part of the token, decode it and parse it as JSON. Then we can check that the expiry date hasn’t passed.

At the end of it, the `getUserDetails` function should return an object of the `UserDetails` type or `null`, depending on whether a valid token is found or not. Put together, it looks like this:

```
public getUserDetails(): UserDetails {
  const token = this.getToken();
  let payload;
  if (token) {
    payload = token.split('.')[1];
    payload = window.atob(payload);
    return JSON.parse(payload);
  } else {
    return null;
  }
}
```

The user details that are provided include the information about the user's name, email, and the expiration of the token, which we'll use to check if the user session is valid.

CHECK WHETHER A USER IS LOGGED IN

Add a new method called `isLoggedIn` to the service. It uses the `getUserDetails` method to get the token details from the JWT token and checks the expiration hasn't passed yet:

```
public isLoggedIn(): boolean {
  const user = this.getUserDetails();
  if (user) {
    return user.exp > Date.now() / 1000;
  } else {
    return false;
  }
}
```

If the token exists, the method will return if the user is logged in as a boolean value. Now we can construct our HTTP requests to load data, using the token for authorization.

STRUCTURING THE API CALLS

To facilitate making API calls, add the `request` method to the `AuthenticationService`, which is able to construct and return the proper HTTP request observable depending on the specific type of request. It's a private method, since it's only used by this service, and exists just to reduce code duplication. This will use the Angular `HttpClient` service; remember to inject this into the `AuthenticationService` if it's not already there:

```
private request(method: 'post' | 'get', type:
  'login' | 'register' | 'profile', user?: TokenPayload):
  Observable<any> {
  let base;

  if (method === 'post') {
```

```

        base = this.http.post(`/api/${type}`, user);
    } else {
        base = this.http.get(`/api/${type}`, { headers: {
        Authorization: `Bearer ${this.getToken()}` }});
    }

    const request = base.pipe(
        map((data: TokenResponse) => {
            if (data.token) {
                this.saveToken(data.token);
            }
            return data;
        })
    );

    return request;
}

```

It does require the `map` operator from RxJS in order to intercept and store the token in the service if it's returned by an API login or register call. Now we can implement the public methods to call the API.

CALLING THE REGISTER AND LOGIN API ENDPOINTS

Just three methods to add. We'll need an interface between the Angular app and the API, to call the login and register endpoints and save the returned token, or the profile endpoint to get the user details:

```

public register(user: TokenPayload): Observable<any> {
    return this.request('post', 'register', user);
}

public login(user: TokenPayload): Observable<any> {
    return this.request('post', 'login', user);
}

public profile(): Observable<any> {
    return this.request('get', 'profile');
}

```

Each method returns an observable that will handle the HTTP request for one of the API calls we need to make. That finalizes the service; now to tie everything together in the Angular app.

Apply Authentication to Angular App

We can use the `AuthenticationService` inside the Angular app in a number of ways to give the experience we're after:

1. wire up the register and sign-in forms
2. update the navigation to reflect the user's status
3. only allow logged-in users access to the `/profile` route
4. call the protected `/api/profile` API route.

CONNECT THE REGISTER AND LOGIN CONTROLLERS

We'll begin by looking at the register and login forms.

The Register Page

The HTML for the registration form already exists and has `NgModel` directives attached to the fields, all bound to properties set on the `credentials` controller property. The form also has a `(submit)` event binding to handle the submission. In the example application, it's in `/client/src/app/register/register.component.html` and looks like this:

```
<form (submit)="register()">
  <div class="form-group">
    <label for="name">Full name</label>
    <input type="text" class="form-control" name="name"
placeholder="Enter your name"
[(ngModel)]="credentials.name">
  </div>
  <div class="form-group">
    <label for="email">Email address</label>
    <input type="email" class="form-control" name="email"
placeholder="Enter email"
[(ngModel)]="credentials.email">
  </div>
  <div class="form-group">
    <label for="password">Password</label>
    <input type="password" class="form-control"
```

```

name="password" placeholder="Password"
[ (ngModel) ]="credentials.password">
  </div>
  <button type="submit" class="btn btn-default">Register!
</button>
</form>

```

The first task in the controller is to ensure our `AuthenticationService` and the `Router` are injected and available through the constructor. Next, inside the `register` handler for the form submit, make a call to `auth.register`, passing it the credentials from the form.

The `register` method returns an observable, which we need to subscribe to in order to trigger the request. The observable will emit success or failure, and if someone has successfully registered, we'll set the application to redirect them to the profile page or log the error in the console.

In the sample application, the controller is in `/client/src/app/register/register.component.ts` and looks like this:

```

import { Component } from '@angular/core';
import { AuthenticationService, TokenPayload } from
'../authentication.service';
import { Router } from '@angular/router';

@Component({
  templateUrl: './register.component.html'
})
export class RegisterComponent {
  credentials: TokenPayload = {
    email: '',
    name: '',
    password: ''
  };

  constructor(private auth: AuthenticationService,
private router: Router) {}

  register() {
    this.auth.register(this.credentials).subscribe(() =>
{
    this.router.navigateByUrl('/profile');
  }, (err) => {

```

```

        console.error(err);
    });
}
}

```

The Login Page

The login page is very similar in nature to the register page, but in this form we don't ask for the name, just email and password.

In the sample application, it's in

/client/src/app/login/login.component.html and looks like this:

```

<form (submit)="login()">
  <div class="form-group">
    <label for="email">Email address</label>
    <input type="email" class="form-control" name="email"
placeholder="Enter email"
[(ngModel)]="credentials.email">
  </div>
  <div class="form-group">
    <label for="password">Password</label>
    <input type="password" class="form-control"
name="password" placeholder="Password"
[(ngModel)]="credentials.password">
  </div>
  <button type="submit" class="btn btn-default">Sign in!
</button>
</form>

```

Once again, we have the form submit handler, and `NgModel` attributes for each of the inputs. In the controller, we want the same functionality as the register controller, but this time called the `login` method of the `AuthenticationService`.

In the sample application, the controller is in

/client/src/app/login/login.controller.ts and look like this:

```

import { Component } from '@angular/core';
import { AuthenticationService, TokenPayload } from
'../authentication.service';
import { Router } from '@angular/router';

@Component({

```



```

    templateUrl: './login.component.html'
  })
  export class LoginComponent {
    credentials: TokenPayload = {
      email: '',
      password: ''
    };

    constructor(private auth: AuthenticationService,
      private router: Router) {}

    login() {
      this.auth.login(this.credentials).subscribe(() => {
        this.router.navigateByUrl('/profile');
      }, (err) => {
        console.error(err);
      });
    }
  }
}

```

Now users can register and sign in to the application. Note that, again, there should be more validation in the forms to ensure that all required fields are filled before submitting. These examples are kept to the bare minimum to highlight the main functionality.

CHANGE CONTENT BASED ON USER STATUS

In the navigation, we want to show the *Sign in* link if a user isn't logged in, and their username with a link to the profile page if they are logged in. The navbar is found in the App component.

First, we'll look at the App component controller. We can inject the `AuthenticationService` into the component and call it directly in our template. In the sample app, the file is in `/client/src/app/app.component.ts` and looks like this:

```

import { Component } from '@angular/core';
import { AuthenticationService } from
  './authentication.service';

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html'
})

```

```
})  
export class AppComponent {  
  constructor(public auth: AuthenticationService) {}  
}
```

That's pretty simple, right? Now, in the associated template we can use `auth.isLoggedIn()` to determine whether to display the sign-in link or the profile link. To add the user's name to the profile link, we can access the name property of `auth.getUserDetails()?.name`. Remember that this is getting the data from the JWT. The `?.` operator is a special way to access a property on an object that may be undefined, without throwing an error.

In the sample app, the file is in `/client/src/app/app.component.html` and the updated part looks like this:

```
<ul class="nav navbar-nav navbar-right">  
  <li *ngIf="!auth.isLoggedIn()"><a  
routerLink="/login">Sign in</a></li>  
  <li *ngIf="auth.isLoggedIn()"><a routerLink="/profile">  
{{ auth.getUserDetails()?.name }}</a></li>  
  <li *ngIf="auth.isLoggedIn()"><a  
(click)="auth.logout()">Logout</a></li>  
</ul>
```

PROTECT A ROUTE FOR LOGGED IN USERS ONLY

In this step, we'll see how to make a route accessible only to logged-in users, by protecting the `/profile` path.

Angular allows you to define a route guard, which can run a check at several points of the routing life cycle to determine if the route can be loaded. We'll use the `CanActivate` hook to tell Angular to load the profile route only if the user is logged in.

To do, this we need to create a route guard service, `ng generate service auth-guard`. It must implement the `CanActivate` interface, and the associated `canActivate` method. This method returns a boolean value from the `AuthenticationService.isLoggedIn` method (basically checks if the token is found, and still valid), and if the user is not valid also redirects them to the home page:

```
import { Injectable } from '@angular/core';
import { Router, CanActivate } from '@angular/router';
import { AuthenticationService } from
'./authentication.service';

@Injectable()
export class AuthGuardService implements CanActivate {

  constructor(private auth: AuthenticationService,
private router: Router) {}

  canActivate() {
    if (!this.auth.isLoggedIn()) {
      this.router.navigateByUrl('/');
      return false;
    }
    return true;
  }
}
```

To enable this guard, we have to declare it on the route configuration. There's a property called `canActivate`, which takes an array of services that should be called before activating the route. Ensure you also declare these services in the `App NgModule's providers` array. The routes are defined in the [App module](#), which contains the routes like you see here:

```
const routes: Routes = [
  { path: '', component: HomeComponent },
  { path: 'login', component: LoginComponent },
  { path: 'register', component: RegisterComponent },
  { path: 'profile', component: ProfileComponent,
canActivate: [AuthGuardService] }
];
```

With that route guard in place, now if an unauthenticated user tries to visit the profile page, Angular will cancel the route change and redirect to the home page, thus protecting it from unauthenticated users.

CALL A PROTECTED API ROUTE

The `/api/profile` route has been set up to check for a JWT in the request. Otherwise, it will return a 401 unauthorized error.

To pass the token to the API, it needs to be sent through as a header on the request, called `Authorization`. The following snippet shows the main data service function, and the format required to send the token. The `AuthenticationService` already handles this, but you can find this in `/client/src/app/authentication.service.ts`.

```
base = this.http.get(`/api/${type}`, { headers: {  
  Authorization: `Bearer ${this.getToken()}` }});
```

Remember that the back-end code is validating that the token is genuine when the request is made, by using the secret known only to the issuing server.

To make use of this in the profile page, we just need to update the controller, in `/client/src/app/profile/profile.component.ts` in the sample app. This will populate a `details` property when the API returns some data, which should match the `UserDetails` interface.

```
import { Component } from '@angular/core';  
import { AuthenticationService, UserDetails } from  
  '../authentication.service';  
  
@Component({  
  templateUrl: './profile.component.html'  
})  
export class ProfileComponent {
```

```

    details: UserDetails;

    constructor(private auth: AuthenticationService) {}

    ngOnInit() {
      this.auth.profile().subscribe(user => {
        this.details = user;
      }, (err) => {
        console.error(err);
      });
    }
  }
}

```

Then, of course, it's just a case of updating the bindings in the view

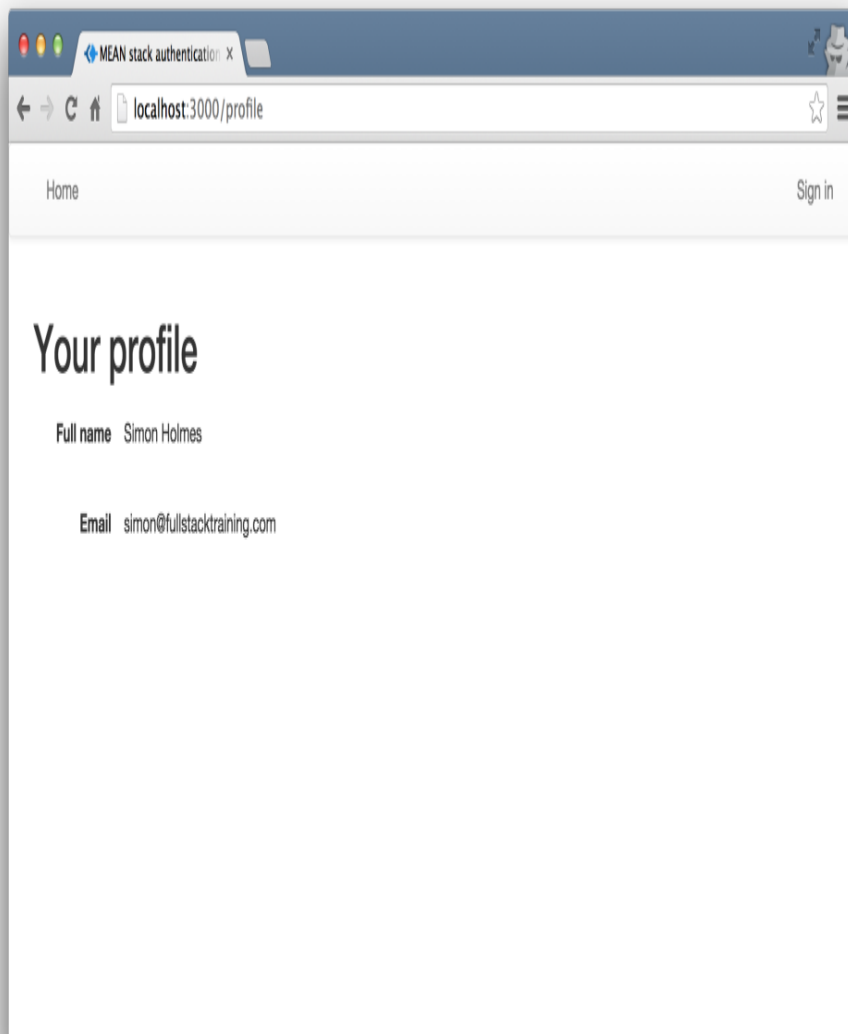
(</client/src/app/profile/profile.component.html>). Again, the `?.` is a safety operator for binding properties that don't exist on first render (since data has to load first).

```

<div class="form-horizontal">
  <div class="form-group">
    <label class="col-sm-3 control-label">Full
name</label>
    <p class="form-control-static">{{ details?.name }}
</p>
  </div>
  <div class="form-group">
    <label class="col-sm-3 control-label">Email</label>
    <p class="form-control-static">{{ details?.email }}
</p>
  </div>
</div>

```

And here's the final profile page, when logged in:



That's how to manage authentication in the MEAN stack, from securing API routes and managing user details to working with JWTs and protecting routes.

Chapter 5: Build a JavaScript Command Line Interface (CLI) with Node.js

BY LUKAS WHITE & MICHAEL WANYOIKE

As great as Node.js is for “traditional” web applications, its potential uses are far broader. Microservices, REST APIs, tooling, working with the Internet of Things and even desktop applications: it’s got your back.

Another area where Node.js is really useful is for building command-line applications — and that’s what we’re going to be doing in this article. We’re going to start by looking at a number of third-party packages designed to help work with the command line, then build a real-world example from scratch.

What we’re going to build is a tool for initializing a Git repository. Sure, it’ll run `git init` under the hood, but it’ll do more than just that. It will also create a remote repository on GitHub right from the command line, allow the user to interactively create a `.gitignore` file, and finally perform an initial commit and push.

As ever, the code accompanying this tutorial can be found on our [GitHub repo](#).

Why Build a Command-line Tool with Node.js?

Before we dive in and start building, it's worth looking at why we might choose Node.js to build a command-line application.

The most obvious advantage is that, if you're reading this, you're probably already familiar with it — and, indeed, with JavaScript.

Another key advantage, as we'll see as we go along, is that the strong Node.js ecosystem means that among the hundreds of thousands of packages available for all manner of purposes, there are a number which are specifically designed to help build powerful command-line tools.

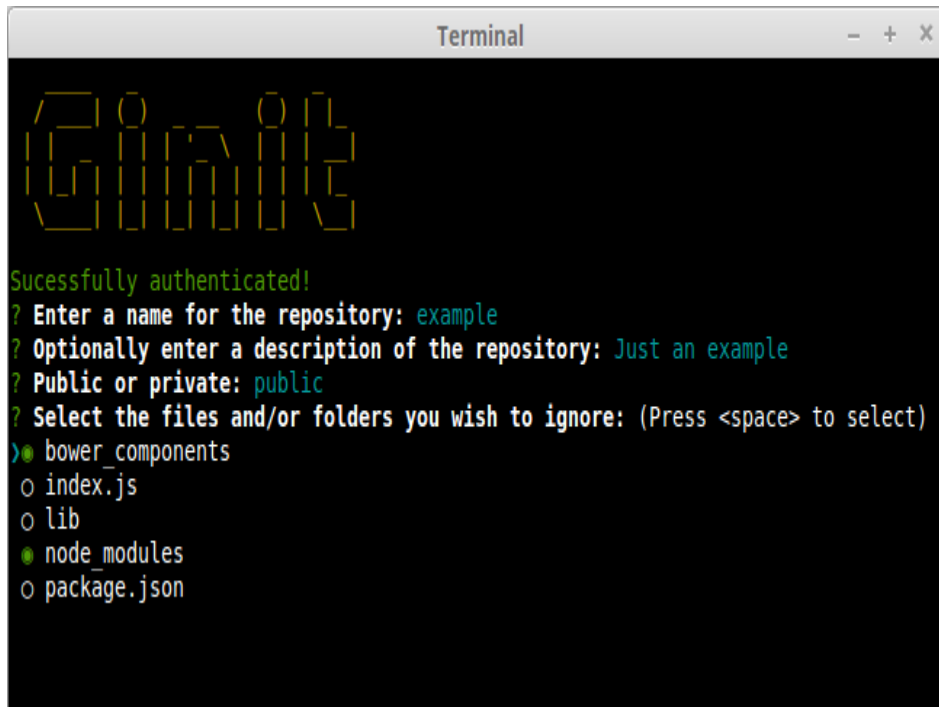
Finally, we can use `npm` to manage any dependencies, rather than have to worry about OS-specific package managers such as Aptitude, Yum or Homebrew.

Although Your Tool May Have Other Dependencies

That isn't necessarily true, in that your command-line tool may have other external dependencies.

Tip: that isn't necessarily true, in that your command-line tool may have other external dependencies.

What We're Going to Build: ginit



```
Terminal
Ginit
Sucessfully authenticated!
? Enter a name for the repository: example
? Optionally enter a description of the repository: Just an example
? Public or private: public
? Select the files and/or folders you wish to ignore: (Press <space> to select)
● bower_components
○ index.js
○ lib
● node_modules
○ package.json
```

For this tutorial, We're going to create a command-line utility which I'm calling **ginit**. It's `git init`, but on steroids.

You're probably wondering what on earth that means.

As you no doubt already know, `git init` initializes a git repository in the current folder. However, that's usually only one of a number of repetitive steps involved in the process of hooking up a new or existing project to Git. For example, as part of a typical workflow, you may well:

1. initialise the local repository by running `git init`
2. create a remote repository, for example on GitHub or Bitbucket — typically by leaving the command line and firing up a web browser
3. add the remote
4. create a `.gitignore` file
5. add your project files
6. commit the initial set of files
7. push up to the remote repository.

There are often more steps involved, but we'll stick to those for the purposes of our app. Nevertheless, these steps are pretty repetitive. Wouldn't it be better if we could do all this from the command line, with no copy-pasting of Git URLs and such like?

So what ginit will do is create a Git repository in the current folder, create a remote repository — we'll be using GitHub for this — and then add it as a remote. Then it will provide a simple interactive “wizard” for creating a `.gitignore` file, add the contents of the folder and push it up to the remote repository. It might not save you hours, but it'll remove some of the initial friction when starting a new project.

With that in mind, let's get started.

The Application Dependencies

One thing is for certain: in terms of appearance, the console will never have the sophistication of a graphical user interface. Nevertheless, that doesn't mean it has to be plain, ugly, monochrome text. You might be surprised by just how much you can do visually, while at the same time keeping it functional. We'll be looking at a couple of libraries for enhancing the display: `chalk` for colorizing the output and `clui` to add some additional visual components. Just for fun, we'll use `figlet` to create a fancy ASCII-based banner and we'll also use `clear` to clear the console.

In terms of input and output, the low-level `Readline` Node.js module could be used to prompt the user and request input, and in simple cases is more than adequate. But we're going to take advantage of a third-party package which adds a greater degree of sophistication — `Inquirer`. As well as providing a mechanism for asking questions, it also implements simple input controls: think radio buttons and checkboxes, but in the console.

We'll also be using minimist to parse command-line arguments.

Here's a complete list of the packages we'll use specifically for developing on the command line:

- chalk — colorizes the output
- clear — clears the terminal screen
- clui — draws command-line tables, gauges and spinners
- figlet — creates ASCII art from text
- inquirer — creates interactive command-line user interface
- minimist — parses argument options
- configstore — easily loads and saves config without you having to think about where and how.

Additionally, we'll also be using the following:

- @octokit/rest — a GitHub REST API client for Node.js
- lodash — a JavaScript utility library
- simple-git — a tool for running Git commands in a Node.js application
- touch — a tool for implementating the Unix touch command.

Getting Started

Although we're going to create the application from scratch, don't forget that you can also grab a copy of the code from the repository which accompanies this article.

Create a new directory for the project. You don't have to call it `ginit`, of course:

```
mkdir ginit
cd ginit
```

Create a new `package.json` file:

```
npm init
```

Follow the simple wizard, for example:

```
name: (ginit)
version: (1.0.0)
description: "git init" on steroids
entry point: (index.js)
test command:
git repository:
keywords: Git CLI
author: [YOUR NAME]
license: (ISC)
```

Now install the dependencies:

```
npm install chalk clear clui figlet inquirer minimist
configstore @octokit/rest lodash simple-git touch --save
```

Alternatively, simply copy-paste the following `package.json` file — modifying the author appropriately — or grab it from [the repository which accompanies this article](#):

```
{
  "name": "ginit",
  "version": "1.0.0",
  "description": "\"git init\" on steroids",
  "main": "index.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "keywords": [
    "Git",
    "CLI"
  ],
  "author": "Lukas White <hello@lukaswhite.com>",
  "license": "ISC",
  "bin": {
    "ginit": "./index.js"
  },
  "dependencies": {
    "@octokit/rest": "^14.0.5",
    "chalk": "^2.3.0",
    "clear": "0.0.1",
    "clui": "^0.3.6",
```

```
"configstore": "^3.1.1",
"figlet": "^1.2.0",
"inquirer": "^5.0.1",
"lodash": "^4.17.4",
"minimist": "^1.2.0",
"simple-git": "^1.89.0",
"touch": "^3.1.0"
}
}
```

Now create an `index.js` file in the same folder and require the following dependencies:

```
const chalk      = require('chalk');
const clear      = require('clear');
const figlet     = require('figlet');
```

Adding Some Helper Methods

We're going to create a `lib` folder where we'll split our helper code into modules:

- **files.js** — basic file management
- **inquirer.js** — command-line user interaction
- **github.js** — access token management
- **repo.js** — Git repository management.

Let's start with `lib/files.js`. Here, we need to:

- get the current directory (to get a default repo name)
- check whether a directory exists (to determine whether the current folder is already a Git repository by looking for a folder named `.git`).

This sounds straightforward, but there are a couple of gotchas to take into consideration.

Firstly, you might be tempted to use the `fs` module's `realpathSync` method to get the current directory:

```
path.basename(path.dirname(fs.realpathSync(__filename)));
```

This will work when we're calling the application from the same directory (e.g. using `node index.js`), but bear in mind that we're going to be making our console application available globally. This means we'll want the name of the directory we're working in, not the directory where the application resides. For this purpose, it's better to use `process.cwd`:

```
path.basename(process.cwd());
```

Secondly, the preferred method of checking whether a file or directory exists keeps changing. The current way is to use `fs.stat` or `fs.statSync`. These throw an error if there's no file, so we need to use a `try ... catch` block.

Finally, it's worth noting that when you're writing a command-line application, using the synchronous version of these sorts of methods is just fine.

Putting that all together, let's create a utility package in `lib/files.js`:

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

module.exports = {
  getCurrentDirectoryBase : () => {
    return path.basename(process.cwd());
  },

  directoryExists : (filePath) => {
    try {
      return fs.statSync(filePath).isDirectory();
    } catch (err) {
      return false;
    }
  }
}
```

```
}  
};
```

Go back to `index.js` and ensure you require the new file:

```
const files = require('./lib/files');
```

With this in place, we can start developing the application.

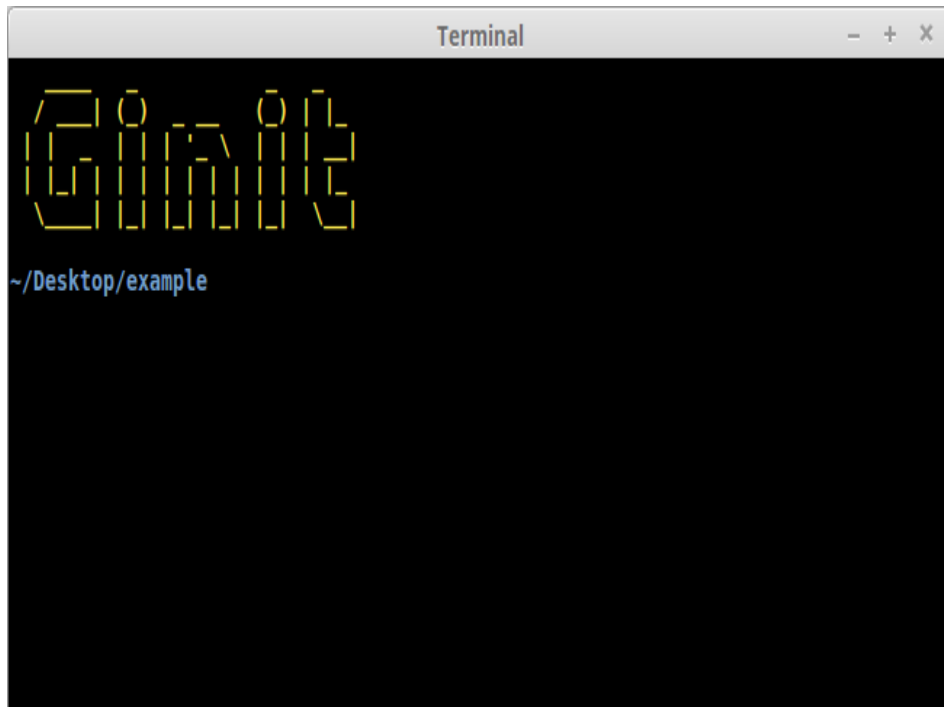
Initializing the Node CLI

Now let's implement the start-up phase of our console application.

In order to demonstrate some of the packages we've installed to enhance the console output, let's clear the screen and then display a banner:

```
clear();  
console.log(  
  chalk.yellow(  
    figlet.textSync('Ginit', { horizontalLayout: 'full'  
  })  
)  
);
```

The output from this is shown below.



Next up, let's run a simple check to ensure that the current folder isn't already a Git repository. That's easy: we just check for the existence of a `.git` folder using the utility method we just created:

```
if (files.directoryExists('.git')) {  
  console.log(chalk.red('Already a git repository!'));  
  process.exit();  
}
```

Coloring the Message

Notice we're using the `chalk` module to show a red-colored message.

Prompting the User for Input

The next thing we need to do is create a function that will prompt the user for their GitHub credentials.

We can use Inquirer for this. The module includes a number of methods for various types of prompts, which are roughly analogous to HTML form controls. In order to collect the user's GitHub username and password, we're going to use the `input` and `password` types respectively.

First, create `lib/inquirer.js` and insert this code:

```
const inquirer = require('inquirer');
const files = require('./files');

module.exports = {

  askGithubCredentials: () => {
    const questions = [
      {
        name: 'username',
        type: 'input',
        message: 'Enter your GitHub username or e-mail address:',
        validate: function( value ) {
          if (value.length) {
            return true;
          } else {
            return 'Please enter your username or e-mail address.';
          }
        },
      },
      {
        name: 'password',
        type: 'password',
        message: 'Enter your password:',
        validate: function(value) {
          if (value.length) {
            return true;
          } else {
            return 'Please enter your password.';
          }
        },
      },
    ];
    return inquirer.prompt(questions);
  },
}
```

As you can see, `inquirer.prompt()` asks the user a series of questions, provided in the form of an array as the first argument. Each question is made up of an object which defines

the name of the field, the type (we're just using `input` and `password` respectively here, but later we'll look at a more advanced example), and the prompt (`message`) to display.

The input the user provides will be passed in to the calling function as a promise. If successful, we'll end up with a simple object with two properties; `username` and `password`.

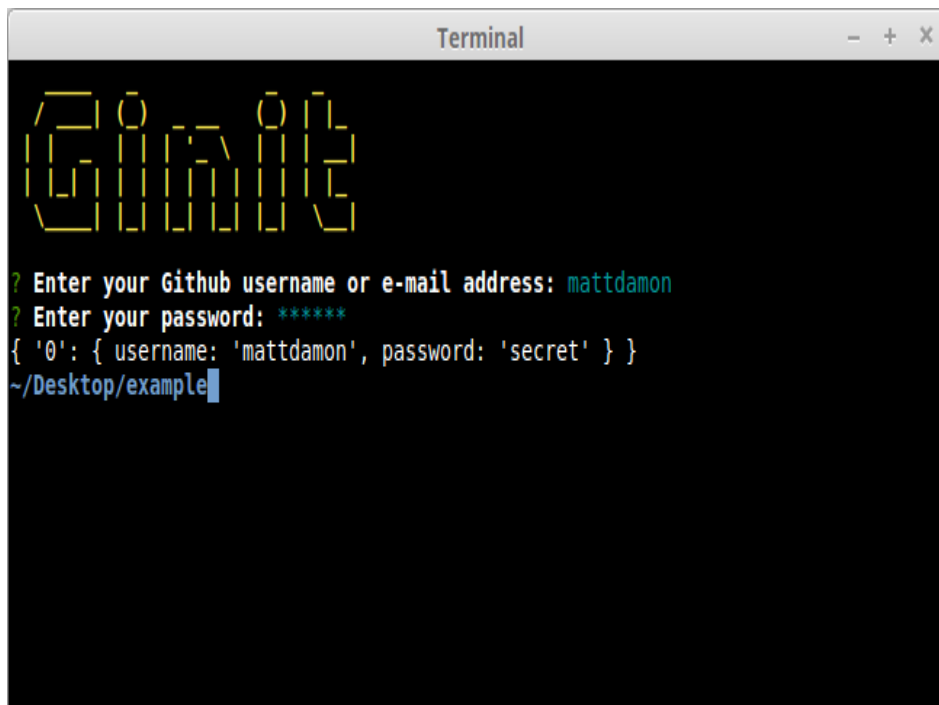
You can test all of this out by adding the following to `index.js`:

```
const inquirer = require('./lib/inquirer');

const run = async () => {
  const credentials = await
  inquirer.askGithubCredentials();
  console.log(credentials);
}

run();
```

Then run the script using `node index.js`.



```
Terminal
? Enter your Github username or e-mail address: mattdamon
? Enter your password: *****
{ '0': { username: 'mattdamon', password: 'secret' } }
~/Desktop/example
```

Dealing With GitHub Authentication

The next step is to create a function to retrieve an OAuth token for the GitHub API. Essentially, we're going to "exchange" the username and password for a token.

Of course, we don't want users to have to enter their credentials every time they use the tool. Instead, we'll store the OAuth token for subsequent requests. This is where the `configstore` package comes in.

STORING CONFIG

Storing config is outwardly quite straightforward: you can simply read and write to/from a JSON file without the need for a third-party package. However, the `configstore` package provides a few key advantages:

1. It determines the most appropriate location for the file for you, taking into account your operating system and the current user.
2. There's no need to explicitly read or write to the file. You simply modify a `configstore` object and that's taken care of for you in the background.

To use it, simply create an instance, passing it an application identifier. For example:

```
const Configstore = require('configstore');
const conf = new Configstore('ginit');
```

If the `configstore` file doesn't exist, it'll return an empty object and create the file in the background. If there's already a `configstore` file, the contents will be decoded into JSON and made available to your application. You can now use `conf` as a simple object, getting or setting properties as required. As mentioned above, you don't need to worry about saving it afterwards. That gets taken care of for you.

On macOS/Linux

On macOS/Linux, you'll find the file in `/Users/[YOUR-USERNAME]/.config/configstore/ginit.json`

Communicating with the GitHub API

Let's create a library for handling the GitHub token. Create the file `lib/github.js` and place the following code inside it:

```
const octokit      = require('@octokit/rest')();
const Configstore  = require('configstore');
const pkg          = require('../package.json');
const _            = require('lodash');
const CLI          = require('clui');
const Spinner      = CLI.Spinner;
const chalk        = require('chalk');

const inquirer     = require('./inquirer');

const conf = new Configstore(pkg.name);
```

Now let's add the function that checks whether we've already got an access token. We'll also add a function that allows other libs to access `octokit`(GitHub) functions:

```
...
module.exports = {

  getInstance: () => {
    return octokit;
  },

  getStoredGithubToken : () => {
    return conf.get('github.token');
  },

  setGithubCredentials : async () => {
    ...
  },

  registerNewToken : async () => {
    ...
  }
}
```

```
}  
  
}
```

If a `conf` object exists and it has `github.token` property, this means that there's already a token in storage. In this case, we return the token value back to the invoking function. We'll get to that later on.

If no token is detected, we need to fetch one. Of course, getting an OAuth token involves a network request, which means a short wait for the user. This gives us an opportunity to look at the `clui` package which provides some enhancements for console-based applications, among them an animated spinner.

Creating a spinner is easy:

```
const status = new Spinner('Authenticating you, please  
wait...');  
status.start();
```

Once you're done, simply stop it and it will disappear from the screen:

```
status.stop();
```

Dynamic Captions

You can also set the caption dynamically using the `update` method. This could be useful if you have some indication of progress, for example displaying the percentage complete.

Here's the code to authenticate with GitHub:

```
...  
setGithubCredentials : async () => {  
  const credentials = await  
  inquirer.askGithubCredentials();  
  octokit.authenticate(  

```

```

        .extend(
        {
            type: 'basic',
        },
        credentials
    )
    );
},

registerNewToken : async () => {
    const status = new Spinner('Authenticating you,
please wait...');
    status.start();

    try {
        const response = await
octokit.authorization.create({
            scopes: ['user', 'public_repo', 'repo',
'repo:status'],
            note: 'ginit, the command-line tool for
initializing Git repos'
        });
        const token = response.data.token;
        if(token) {
            conf.set('github.token', token);
            return token;
        } else {
            throw new Error("Missing Token", "GitHub token was
not found in the response");
        }
    } catch (err) {
        throw err;
    } finally {
        status.stop();
    }
},

```

Let's step through this:

1. we prompt the user for their credentials using the `setGithubCredentials` method we defined earlier
2. we use basic authentication prior to trying to obtain an OAuth token
3. we attempt to register a new access token for our application
4. if we manage to get an access token, we set it in the `configstore` for next time.
5. we then return the token.

Any access tokens you create, whether manually or via the API as we're doing here, you'll be able to see them here. During the course of development, you may find you need to delete ginit's

access token — identifiable by the `note` parameter supplied above — so that you can re-generate it.

Working with 2FA

If you have two-factor authentication enabled on your GitHub account, the process is slightly more complicated. You'll need to request the confirmation code — for example, one sent via SMS — then supply it using the `X-GitHub-OTP` header. See [the documentation](#) for further information.

If you've been following along and would like to try out what we have so far, you can update the `run()` function in `index.js` as follows:

```
const run = async () => {
  let token = github.getStoredGithubToken();
  if(!token) {
    await github.setGithubCredentials();
    token = await github.registerNewToken();
  }
  console.log(token);
}
```

Please note you may get a `Promise` error if something goes wrong, such as inputting the wrong password. I'll show you the proper way of handling such errors later.

Creating a Repository

Once we've got an OAuth token, we can use it to create a remote repository with GitHub.

Again, we can use `Inquirer` to ask a series of questions. We need a name for the repo, we'll ask for an optional description, and we also need to know whether it should be public or private.

We'll use `minimist` to grab defaults for the name and description from optional command-line arguments. For

example:

```
ginit my-repo "just a test repository"
```

This will set the default name to `my-repo` and the description to `just a test repository`.

The following line will place the arguments in an array indexed by an underscore:

```
const argv = require('minimist')(process.argv.slice(2));  
// { _: [ 'my-repo', 'just a test repository' ] }
```

More to minimist

This only really scratches the surface of the `minimist` package. You can also use it to interpret flags, switches and name/value pairs. Check out the documentation for more information.

We'll write code to parse the command-line arguments and ask a series of questions. First, update `lib/inquirer.js` by inserting the following code right after `askGithubCredentials` function:

```
...  
askRepoDetails: () => {  
  const argv = require('minimist')  
(process.argv.slice(2));  
  
  const questions = [  
    {  
      type: 'input',  
      name: 'name',  
      message: 'Enter a name for the repository:',  
      default: argv._[0] ||  
files.getCurrentDirectoryBase(),  
      validate: function( value ) {  
        if (value.length) {  
          return true;  
        } else {  
          return 'Please enter a name for the  
repository.';  
        }  
      }  
    }  
  ]  
}
```



```

    }
  },
  {
    type: 'input',
    name: 'description',
    default: argv._[1] || null,
    message: 'Optionally enter a description of the
repository:'
  },
  {
    type: 'list',
    name: 'visibility',
    message: 'Public or private:',
    choices: [ 'public', 'private' ],
    default: 'public'
  }
];
return inquirer.prompt(questions);
},

```

Next, create the file `lib/repo.js` and add this code:

```

const _      = require('lodash');
const fs     = require('fs');
const git    = require('simple-git')();
const CLI    = require('clui');
const Spinner = CLI.Spinner;

const inquirer = require('./inquirer');
const gh       = require('./github');

module.exports = {
  createRemoteRepo: async () => {
    const github = gh.getInstance();
    const answers = await inquirer.askRepoDetails();

    const data = {
      name : answers.name,
      description : answers.description,
      private : (answers.visibility === 'private')
    };

    const status = new Spinner('Creating remote
repository...');
    status.start();

    try {
      const response = await github.repos.create(data);
      return response.data.ssh_url;
    } catch(err) {
      throw err;
    } finally {
      status.stop();
    }
  }
}

```

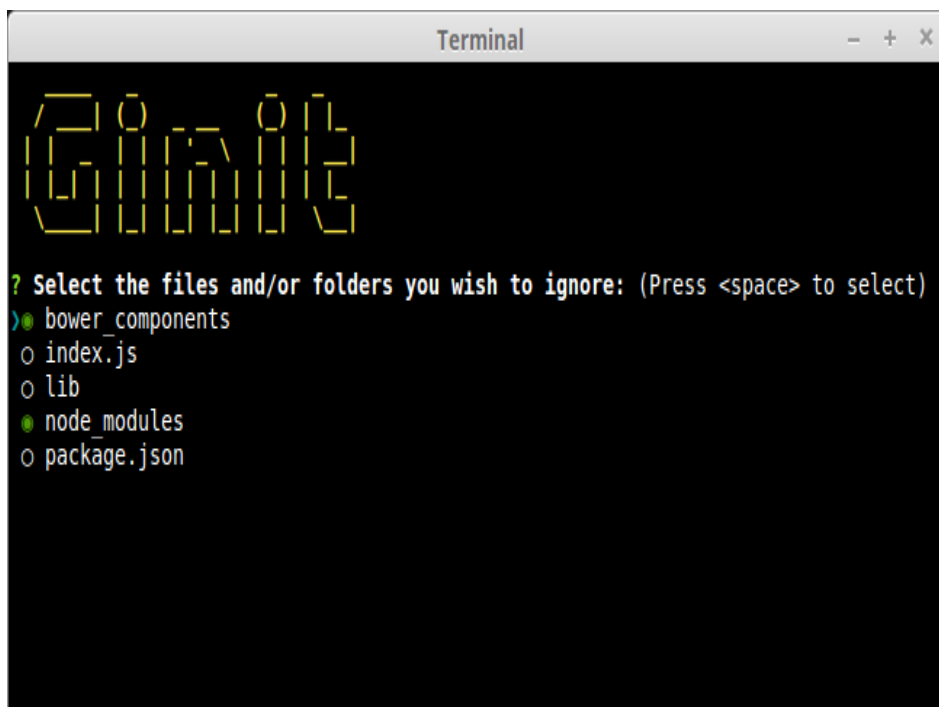
```
},  
}
```

Once we have that information, we can simply use the GitHub package to create a repo, which will give us a URL for the newly created repository. We can then set that up as a remote in our local Git repository. First, however, let's interactively create a `.gitignore` file.

Creating a `.gitignore` File

For the next step, we'll create a simple command-line “wizard” to generate a `.gitignore` file. If the user is running our application in an existing project directory, let's show them a list of files and directories already in the current working directory, and allow them to select which ones to ignore.

The Inquirer package provides a `checkbox` input type for just that.



The first thing we need to do is scan the current directory, ignoring the `.git` folder and any existing `.gitignore` file (we do this by making use of `lodash's without` method):

```
const filelist = _.without(fs.readdirSync('.'), '.git',
  '.gitignore');
```

If there's nothing to add, there's no point in continuing, so let's simply touch the current `.gitignore` file and bail out of the function:

```
if (filelist.length) {
  ...
} else {
  touch('.gitignore');
}
```

Finally, let's utilize Inquirer's checkbox "widget" to list the files. Insert the following code in `lib/inquirer.js`:

```
...
askIgnoreFiles: (filelist) => {
  const questions = [
    {
      type: 'checkbox',
      name: 'ignore',
      message: 'Select the files and/or folders you
wish to ignore:',
      choices: filelist,
      default: ['node_modules', 'bower_components']
    }
  ];
  return inquirer.prompt(questions);
},
..
```

Notice that we can also provide a list of defaults. In this case, we're pre-selecting `node_modules` and `bower_components`, should they exist.

With the Inquirer code in place, we can now construct the `createGitignore()` function. Insert this code in

lib/repo.js:

```
...
  createGitignore: async () => {
    const filelist = _.without(fs.readdirSync('.'),
      '.git', '.gitignore');

    if (filelist.length) {
      const answers = await
inquirer.askIgnoreFiles(filelist);
      if (answers.ignore.length) {
        fs.writeFileSync( '.gitignore',
answers.ignore.join( '\n' ) );
      } else {
        touch( '.gitignore' );
      }
    } else {
      touch('.gitignore');
    }
  },
  ...
```

Once “submitted”, we then generate a `.gitignore` by joining up the selected list of files, separated with a newline. Our function now pretty much guarantees we’ve got a `.gitignore` file, so we can proceed with initializing a Git repository.

Interacting with Git from Within the App

There are a number of ways to interact with Git, but perhaps the simplest is to use the `simple-git` package. This provides a set of chainable methods which, behind the scenes, run the Git executable.

These are the repetitive tasks we’ll use it to automate:

1. run `git init`
2. add the `.gitignore` file
3. add the remaining contents of the working directory
4. perform an initial commit
5. add the newly-created remote repository

6. push the working directory up to the remote.

Insert the following code in `lib/repo.js`:

```
...
  setupRepo: async (url) => {
    const status = new Spinner('Initializing local
repository and pushing to remote...');
    status.start();

    try {
      await git
        .init()
        .add('.gitignore')
        .add('.*')
        .commit('Initial commit')
        .addRemote('origin', url)
        .push('origin', 'master');
      return true;
    } catch(err) {
      throw err;
    } finally {
      status.stop();
    }
  },
  ...
```

Putting It All Together

First, let's set up a couple of helper functions in `lib/github.js`. We need a convenient function for accessing the stored token, and another function for setting up an `oauth` authentication:

```
...
  githubAuth : (token) => {
    octokit.authenticate({
      type : 'oauth',
      token : token
    });
  },

  getStoredGithubToken : () => {
    return conf.get('github.token');
  },
  ...
```

Next, we create a function in `index.js` for handling the logic of acquiring the token. Place this code before the `run()` function:

```
const getGithubToken = async () => {
  // Fetch token from config store
  let token = github.getStoredGithubToken();
  if(token) {
    return token;
  }

  // No token found, use credentials to access GitHub
  // account
  await github.setGithubCredentials();

  // register new token
  token = await github.registerNewToken();
  return token;
}
```

Finally, we update the `run()` function by writing code that will handle the main logic of the app:

```
const run = async () => {
  try {
    // Retrieve & Set Authentication Token
    const token = await getGithubToken();
    github.githubAuth(token);

    // Create remote repository
    const url = await repo.createRemoteRepo();

    // Create .gitignore file
    await repo.createGitignore();

    // Set up local repository and push to remote
    const done = await repo.setupRepo(url);
    if(done) {
      console.log(chalk.green('All done!'));
    }
  } catch(err) {
    if (err) {
      switch (err.code) {
        case 401:
          console.log(chalk.red('Couldn\'t log you in. Please provide correct credentials/token.'));
          break;
        case 422:
          console.log(chalk.red('There already exists a remote repository with the same name'));
          break;
      }
    }
  }
}
```

```
        default:
            console.log(err);
        }
    }
}
```

As you can see, we ensure the user is authenticated before calling all of our other functions (`createRemoteRepo()`, `createGitignore()`, `setupRepo()`) sequentially. The code also handles any errors and offers the user appropriate feedback.

You can check out the completed [index.js](#) file on our GitHub repo.

Making the ginit Command Available Globally

The one remaining thing to do is to make our command available globally. To do this, we'll need to add a [shebang](#) line to the top of `index.js`:

```
#!/usr/bin/env node
```

Next, we need to add a `bin` property to our `package.json` file. This maps the command name (`ginit`) to the name of the file to be executed (relative to `package.json`).

```
"bin": {
  "ginit": "./index.js"
}
```

After that, install the module globally and you'll have a working shell command:

```
npm install -g
```

Works on Windows

This will also work on Windows, as `npm` will helpfully install a `cmd` wrapper alongside your script.

Taking it Further

We've got a fairly nifty, albeit simple command-line app for initializing Git repositories. But there's plenty more you could do to enhance it further.

If you're a Bitbucket user, you could adapt the program to use the Bitbucket API to create a repository. There's a [Node.js API wrapper](#) available to help you get started. You may wish to add an additional command-line option or prompt to ask the user whether they want to use GitHub or Bitbucket (Inquirer would be perfect for just that) or merely replace the GitHub-specific code with a Bitbucket alternative.

You could also provide the facility to specify your own set of defaults for the `.gitignore` file, instead of a hardcoded list. The `preferences` package might be suitable here, or you could provide a set of "templates" — perhaps prompting the user for the type of project. You might also want to look at integrating it with the [.gitignore.io](#) command-line tool/API.

Beyond all that, you may also want to add additional validation, provide the ability to skip certain sections, and more.

Chapter 6: Building a Real-time Chat App with Sails.js

BY MICHAEL WANYOIKE

If you're a developer who currently uses frameworks such as Django, Laravel or Rails, you've probably heard about Node.js. You might already be using a popular front-end library such as Angular or React in your projects. By now, you should be thinking about doing a complete switchover to a server technology based on Node.js.

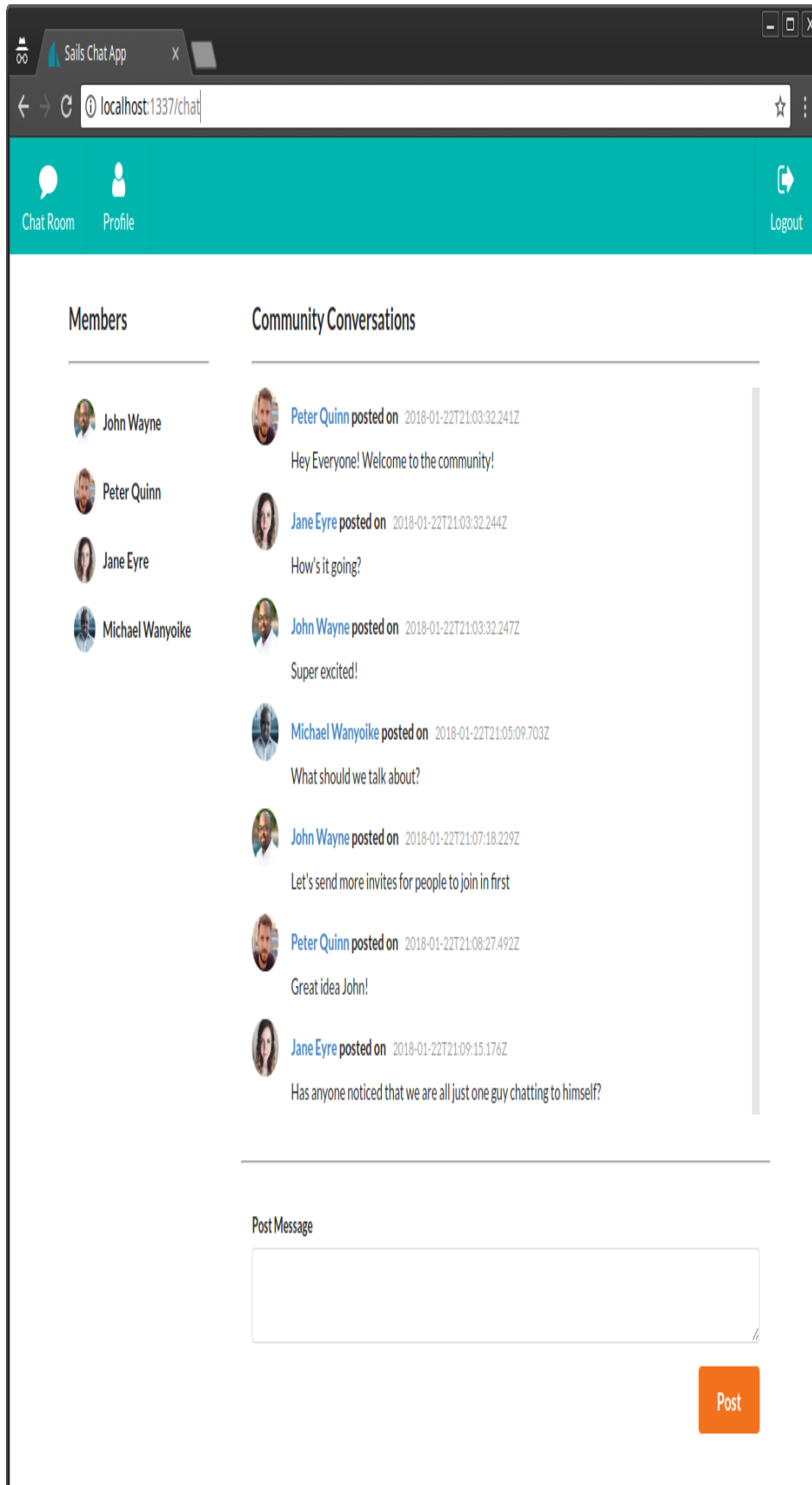
However, the big question is where to start. Today, the JavaScript world has grown at an incredibly fast pace in the last few years, and it seeming to be ever expanding.

If you're afraid of losing your hard-earned programming experience in the Node universe, fear not, as we have Sails.js.

Sails.js is a real-time MVC framework designed to help developers build production-ready, enterprise-grade Node.js apps in a short time. Sails.js is a pure JavaScript solution that supports multiple databases (simultaneously) and multiple front-end technologies. If you're a Rails developer, you'll be happy to learn that Mike McNeil, the Sails.js founder, was inspired by Rails. You'll find a lot of similarities between Rails and Sails.js projects.

In this article, I'll teach you the fundamentals of Sails.js, by showing you how to build a simple, user-friendly chat

application. The complete source code for the sails-chat project can be found in this [GitHub repo](#).



Prerequisites

Before you start, you need at least to have experience developing applications using MVC architecture. This tutorial is intended for intermediate developers. You'll also need at least to have a basic foundation in these:

- [Node.js](#)
- [Modern JavaScript syntax \(ES6+\)](#).

To make it practical and fair for everyone, this tutorial will use core libraries that are installed by default in a new Sails.js project. Integration with modern front-end libraries such as React, Vue or Angular won't be covered here. However, I highly recommend you look into them after this article. Also, we won't do database integrations. We'll instead use the default, local-disk, file-based database for development and testing.

Project Plan

The goal of this tutorial is to show you how to build a chat application similar to [Slack](#), [Gitter](#) or [Discord](#).

Not really! A lot of time and sweat went into building those wonderful platforms. The current number of features developed into them is quite huge.

Instead, we'll build a minimum viable product version of a chat application which consists of:

- single chat room
- basic authentication (passwordless)
- profile update.

I've added the *profile feature* as a bonus in order to cover a bit more ground on Sails.js features.

Installing Sails.js

Before we start installing Sails.js, we need first to set up a proper Node.js environment. At the time of writing, the latest stable version currently available is v0.12.14. Sails.js v1.0.0 is also available but is currently in beta, not recommended for production use.

The latest stable version of Node I have access to is v8.9.4. Unfortunately, Sails.js v0.12 doesn't work properly with the current latest LTS. However, I've tested with Node v.7.10 and found everything works smoothly. This is still good since we can use some new ES8 syntax in our code.

As a JavaScript developer, you'll realize working with one version of Node.js is not enough. Hence, I recommend using the nvm tool to manage multiple versions of Node.js and NPM easily. If you haven't done so, just purge your existing Node.js installation, then install nvm to help you manage multiple versions of Node.js.

Here are the basic instructions of installing Node v7 and Sails.js:

```
# Install the latest version of Node v7 LTS
nvm install v7

# Make Node v7 the default
nvm default alias v7

# Install Sails.js Global
npm install -g sails
```

If you have a good internet connection, this should only take a couple of minutes or less. Let's now go ahead and create our

new application using the Sails generator command:

```
# Go to your projects folder
cd Projects

# Generate your new app
sails generate new chat-app

# Wait for the install to finish then navigate to the
project folder
cd chat-app

# Start the app
sails lift
```

It should take a few seconds for the app to start. You need to manually open the url `http://localhost:1337` in your browser to see your newly created web app.



Seeing this confirms that we have a running project with no errors, and that we can start working. To stop the project, just press `control + c` at the terminal. Use your favorite code editor (I'm using Atom) to examine the generated project structure. Below are the main folders you should be aware of:

- `api`: controllers, models, services and policies (permissions)
- `assets`: images, fonts, JS, CSS, Less, Sass etc.
- `config`: project configuration e.g. database, routes, credentials, locales, security etc.
- `node_modules`: installed npm packages
- `tasks`: Grunt config scripts and pipeline script for compiling and injecting assets
- `views`: view pages — for example, EJS, Jade or whatever templating engine you prefer
- `.tmp`: temporary folder used by Sails to build and serve your project while in development mode.

Before we proceed, there are a couple of things we need to do:

- **Update EJS package.** If you have EJS 2.3.4 listed in `package.json`, you need to update it by changing it to 2.5.5 immediately. It contains a serious security vulnerability. After changing the version number, do an npm install to perform the update.
- **Hot reloading.** I suggest you install sails-hook-autoreload to enable hot reloading for your Sails.js app. It's not a perfect solution but will make development easier. To install it for this current version of Sails.js, execute the following:

```
npm install sails-hook-autoreload@for-sails-0.12 --save
```

Installing Front-end Dependencies

For this tutorial, we'll spend as little time as possible building an UI. Any CSS framework you're comfortable with will do. For this tutorial, I'll go with the Semantic UI CSS library.

Sails.js doesn't have a specific guide on how to install CSS libraries. There are three or more ways you can go about it. Let's look at each.

1. MANUAL DOWNLOAD

You can download the CSS files and JS scripts yourself, along with their dependencies. After downloading, place the files inside the `assets` folder.

I prefer not to use this method, as it requires manual effort to keep the files updated. I like automating tasks.

2. USING BOWER

This method requires you to create a file called `.bowerrc` at the root of your project. Paste the following snippet:

```
{
  "directory" : "assets/vendor"
}
```

This will instruct Bower to install to the `assets/vendor` folder instead of the default `bower_components` folder. Next, install Bower globally, and your front-end dependencies locally using Bower:

```
# Install bower globally via npm-
npm install -g bower

# Create bower.json file, accept default answers (except
choose y for private)
bower init

# Install semantic-ui via bower
bower install semantic-ui --save
```

```
# Install jsrender
bower install jsrender --save
```

I'll explain the purpose of `jsrender` later. I thought it best to finish the task of installing dependencies in one go. You should take note that `jQuery` has been installed as well, since it's a dependency for `semantic-ui`.

After installing, update `assets/style/importer.less` to include this line:

```
@import '../vendor/semantic/dist/semantic.css';
```

Next include the JavaScript dependencies in `tasks/pipeline.js`:

```
var jsFilesToInject = [

  // Load Sails.io before everything else
  'js/dependencies/sails.io.js',

  // Vendor dependencies
  'vendor/jquery/dist/jquery.js',
  'vendor/semantic/dist/semantic.js',
  'vendor/jsrender/jsrender.js',

  // Dependencies like jQuery or Angular are brought in
  here
  'js/dependencies/**/*.js',

  // All of the rest of your client-side JS files
  // will be injected here in no particular order.
  'js/**/*.js'
];
```

When we run `sails lift`, the JavaScript files will automatically be injected into `views/layout.ejs` file as per `pipeline.js` instructions. The current `grunt` setup will take care of injecting our CSS dependencies for us.

Don't Add Vendor Dependencies

Add the word `vendor` in the `.gitignore` file. We don't want vendor dependencies saved in our repository.

3. USING NPM + GRUNT.COPY

The third method requires a little bit more effort to set up, but will result in a lower footprint. Install the dependencies using npm as follows:

```
npm install semantic-ui-css jsrender --save
```

jQuery will be installed automatically, since it's also listed as a dependency for `semantic-ui-css`. Next we need to place code in `tasks/config/copy.js`. This code will instruct Grunt to copy the required JS and CSS files from `node_modules` to the `assets/vendor` folder for us. The entire file should look like this:

```
module.exports = function(grunt) {

  grunt.config.set('copy', {
    dev: {
      files: [{
        expand: true,
        cwd: './assets',
        src: ['**/*.!(coffee|less)'],
        dest: './tmp/public'
      }],
      //Copy JQuery
      {
        expand: true,
        cwd: './node_modules/jquery/dist/',
        src: ['jquery.min.js'],
        dest: './assets/vendor/jquery'
      },
      //Copy jsrender
      {
        expand: true,
        cwd: './node_modules/jsrender/',
        src: ['jsrender.js'],
        dest: './assets/vendor/jsrender'
      },
      // copy semantic-ui CSS and JS files
      {
        expand: true,
```

```

        cwd: './node_modules/semantic-ui-css/',
        src: ['semantic.css', 'semantic.js'],
        dest: './assets/vendor/semantic-ui'
    },
    //copy semantic-ui icon fonts
    {
        expand: true,
        cwd: './node_modules/semantic-ui-css/themes',
        src: ['**.*', '**/*.css'],
        dest: './assets/vendor/semantic-ui/themes'
    }
  ],
  build: {
    files: [{
      expand: true,
      cwd: './tmp/public',
      src: ['**/*'],
      dest: 'www'
    }
  ]
}
});

grunt.loadNpmTasks('grunt-contrib-copy');
};

```

Add this line to `assets/styles/importer.less`:

```
@import '../vendor/semantic-ui/semantic.css';
```

Add the JS files to `config/pipeline.js`:

```

// Vendor Dependencies
'vendor/jquery/jquery.min.js',
'vendor/semantic-ui/semantic.js',
'vendor/jsrender/jsrender.js',

```

Finally, execute this command to copy the files from `node_modules` the `assets/vendor` folder. You only need to do this once for every clean install of your project:

```
grunt copy:dev
```

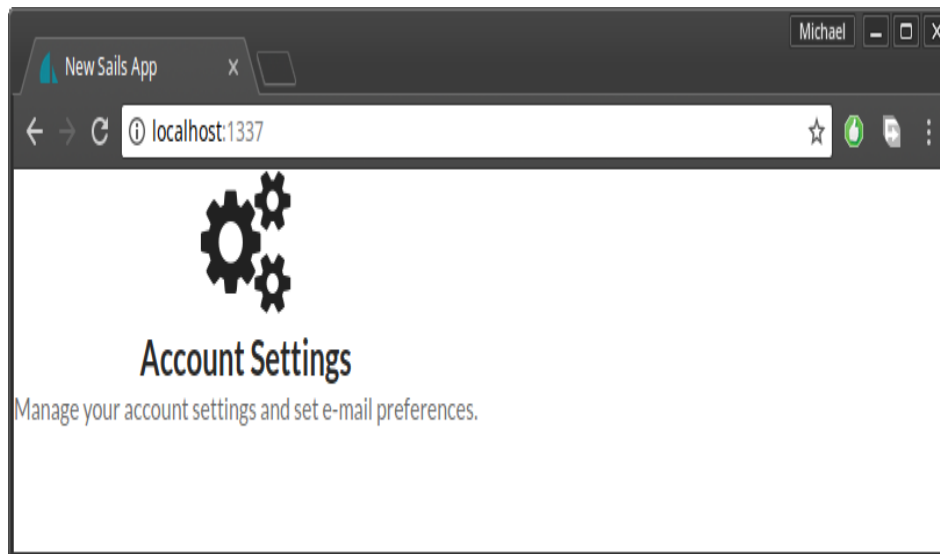
Remember to add `vendor` to your `.gitignore`.

TESTING DEPENDENCIES INSTALLATION

Whichever method you've chosen, you need to ensure that the required dependencies are being loaded. To do this, replace the code in `view/homepage.ejs` with the following:

```
<h2 class="ui icon header">
<i class="settings icon"></i>
<div class="content">
  Account Settings
  <div class="sub header">Manage your account settings
and set e-mail preferences.</div>
</div>
</h2>
```

After saving the file, do a `sails lift`. Your home page should now look like this:



Always do a refresh after restarting your app. If the icon is missing or the font looks off, please review the steps carefully and see what you missed. Use the browser's console to see which files are not loading. Otherwise, proceed with the next stage.

Creating Views

When it comes to project development, I like starting with the user interface. We'll use the Embedded JavaScript Template to create the views. It's a templating engine that's installed by default in every Sails.js project. However, you should be aware it has limited functionality and is no longer under development.

Open `config/bootstrap.js` and insert this line in order to give a proper title to our web pages. Place it right within the existing function before the `cb()` statement:

```
sails.config.appName = "Sails Chat App";
```

You can take a peek at `views/layout.ejs` to see how the `title` tag is set. Next, we begin to build our home page UI.

HOME PAGE DESIGN

Open `/views/homepage.ejs` and replace the existing code with this:

```
<div class="banner">
  <div class="ui segment teal inverted">
    <h1 class="ui center aligned icon header">
      <i class="chat icon"></i>
      <div class="content">
        <a href="/">Sails Chat</a>
        <div class="sub header">Discuss your favorite
technology with the community!</div>
      </div>
    </h1>
  </div>
</div>
<div class="section">
  <div class="ui three column grid">
    <div class="column"></div>
    <div class="column">
      <div class="ui centered padded compact raised
segment">
        <h3>Sign Up or Sign In</h3>
        <div class="ui divider"></div>
        [TODO : Login Form goes here]
      </div>
    </div>
    <div class="column"></div>
```

```
</div>  
</div>
```

To understand the UI elements used in the above code, please refer to the Semantic UI documentation. I've outlined the exact links below:

- [Segment](#)
- [Icon](#)
- [Header](#)
- [Grid](#)

Create a new file in `assets/styles/theme.less` and paste the following content:

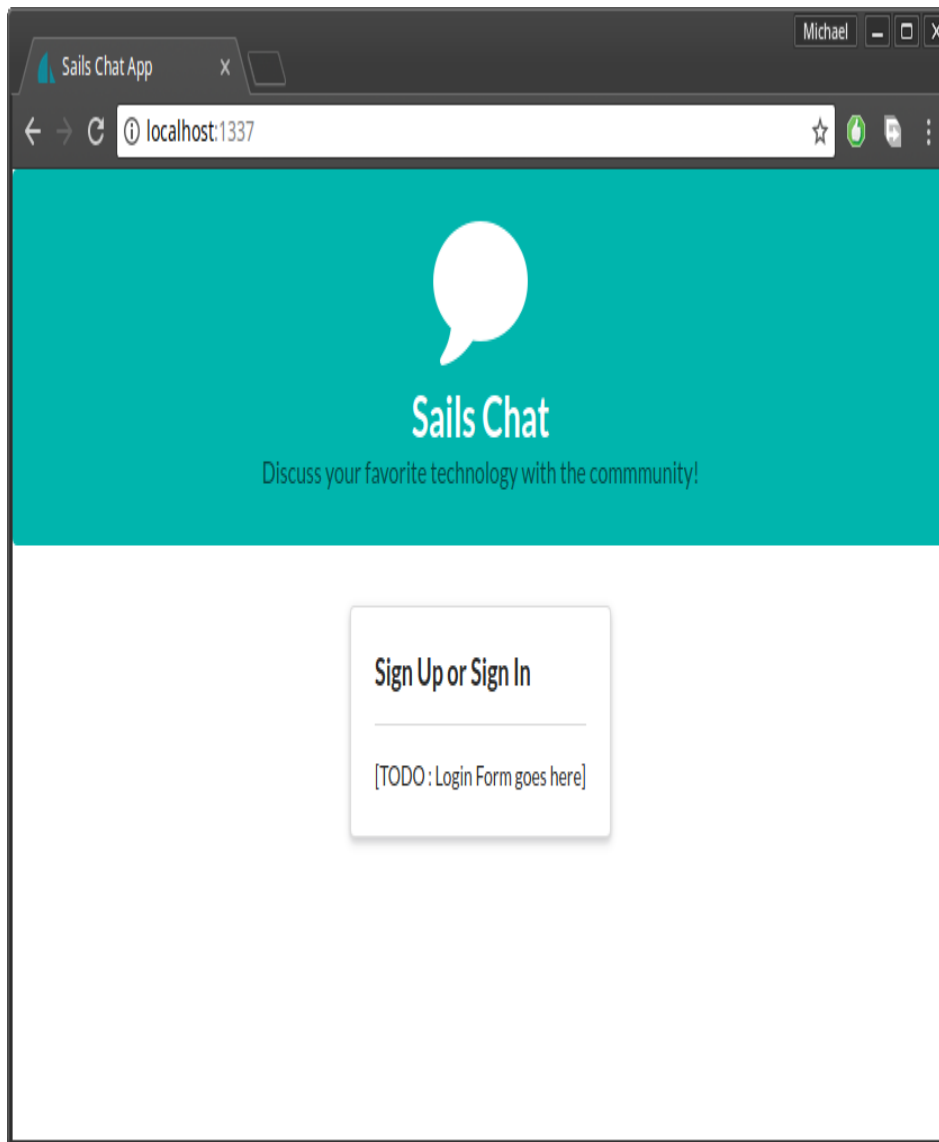
```
.banner a {  
  color: #fff;  
}  
  
.centered {  
  margin-left: auto !important;  
  margin-right: auto !important;  
  margin-bottom: 30px !important;  
}  
  
.section {  
  margin-top: 30px;  
}  
  
.menu {  
  border-radius: 0 !important;  
}  
  
.note {  
  font-size: 11px;  
  color: #2185D0;  
}  
  
#chat-content {  
  height: 90%;  
  overflow-y: scroll;  
}
```

These are all the custom styles we'll use in our project. The rest of the styling will come from the Semantic UI library.

Next, update `assets/styles/importer.less` to include the theme file we just created:

```
@import 'theme.less';
```

Execute `sails lift`. Your project should now look like this:



Next, we'll look at building the navigation menu.

NAVIGATION MENU

This will be created as a partial since it will be shared by multiple view files. Inside the `views` folder, create a folder called `partials`. Then create the file `views/partials/menu.ejs` and paste the following code:

```
<div class="ui labeled icon inverted teal menu">
  <a class="item" href="/chat">
    <i class="chat icon"></i>
    Chat Room
  </a>
  <a class="item" href="/profile">
    <i class="user icon"></i>
    Profile
  </a>
  <div class="right menu">
    <a class="item" href="/auth/logout">
      <i class="sign out icon"></i>
      Logout
    </a>
  </div>
</div>
```

To understand the above code, just refer to the [Menu documentation](#).

If you inspect the above code, you'll notice that we've created a link for `/chat`, `/profile` and `/auth/logout`. Let's first create the views for `profile` and `chat room`.

PROFILE

Create the file `view/profile.ejs` and paste the following code:

```
<% include partials/menu %>

<div class="ui container">
  <h1 class="ui centered header">Profile Updated!</h1>
  <hr>
  <div class="section">
    [ TODO put user-form here ]
  </div>
</div>
```

By now you should be familiar with `header` and `grid` UI elements if you've read the linked documentation. At the root of the document, you'll notice we have a `container` element. (Find out more about this in the [Container](#) documentation.)

We'll build the user form later, once we've built the API. Next we'll create a layout for the chat room.

CHAT ROOM LAYOUT

The chat room will be made up of three sections:

- **Chat users** — list of users
- **Chat messages** — list of messages
- **Chat post** — form for posting new messages.

Create `views/chatroom.ejs` and paste the following code:

```
<% include partials/menu %>

<div class="chat-section">
  <div class="ui container grid">

    <!-- Members List Section -->
    <div class="four wide column">
      [ TODO chat-users ]
    </div>

    <div class="twelve wide column">

      <!-- Chat Messages -->
      [ TODO chat-messages ]

      <hr>

      <!-- Chat Post -->
      [ TODO chat-post ]

    </div>
  </div>
</div>
```

Before we can view the pages, we need to set up routing.

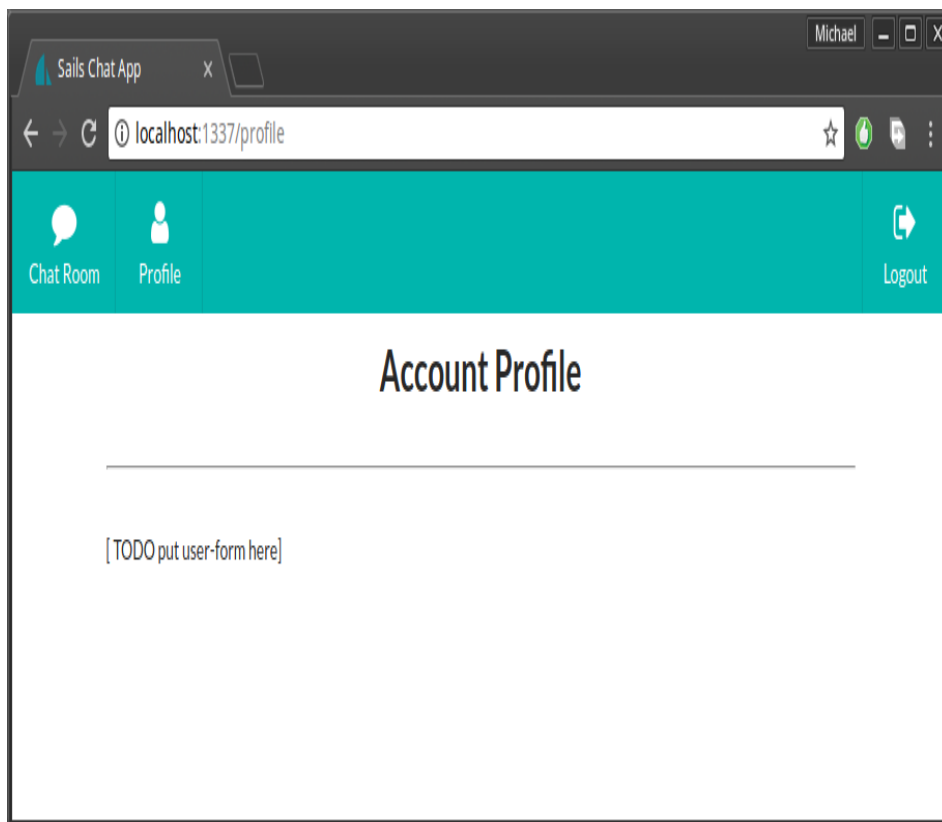
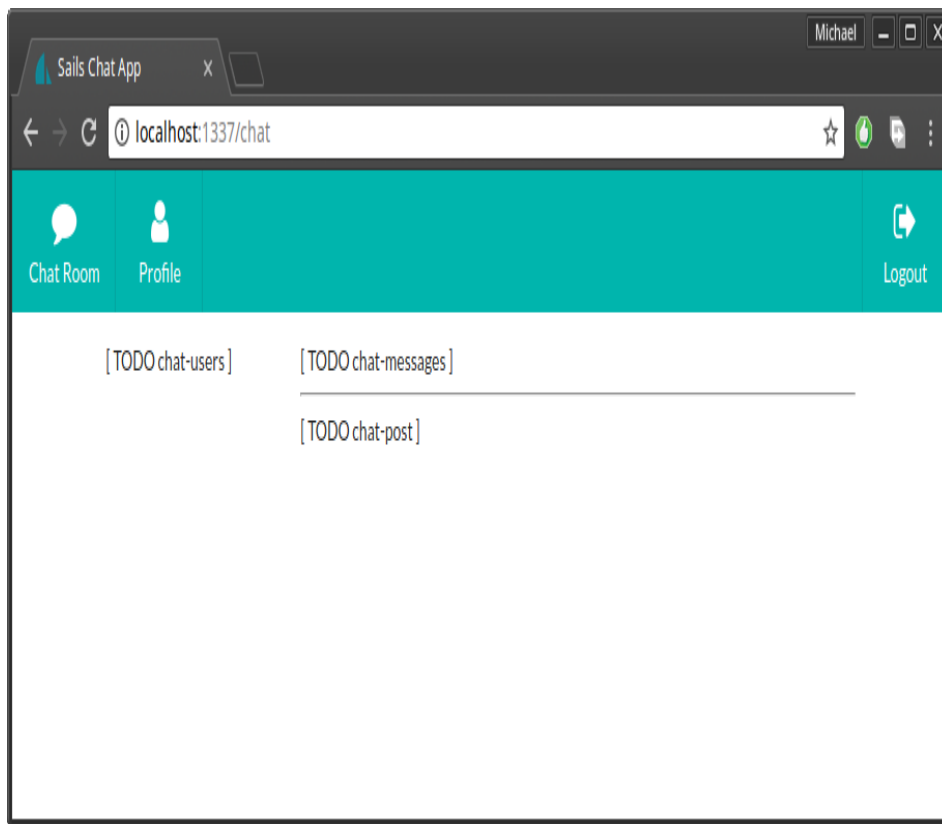
Routing

Open `config/routes.js` and update it like this:

```
'/': {  
  view: 'homepage'  
},  
'/profile': {  
  view: 'profile'  
},  
'/chat': {  
  view: 'chatroom'  
}
```

Sails.js routing is quite flexible. There are many ways of defining routing depending on the scenario. This is the most basic version where we map a URL to a view.

Fire up your Sails app or just refresh your page if it's still running in the background. Currently there's no link between the home page and the other pages. This is intentional, as we'll later build a rudimentary authentication system that will redirect logged in users to `/chat`. For now, use your browser's address bar and add `/chat` or `/profile` at the end URL.



At this stage you should be having the above views. Let's go ahead and start creating the API.

Generating a User API

We're going to use the Sails.js command-line utility to generate our API. We'll need to stop the app for this step:

```
sails generate api User
```

Within a second, we get the message "Created a new api!" Basically, a `User.js` model and a `UserController.js` has just been created for us. Let's update the `api/model/User.js` with some model attributes:

```
module.exports = {  
  attributes: {  
    name: {  
      type: 'string',  
      required: true  
    },  
    email: {  
      type: 'string',  
      required: true,  
      unique: true  
    },  
    avatar: {  
      type: 'string',  
      required: true,  
      defaultsTo: 'https://s.gravatar.com/avatar/e28f6f64608c970c663197d7fe1f5a59?s=60'  
    },  
    location: {  
      type: 'string',  
      required: false,  
      defaultsTo: ''  
    },  
    bio: {  
      type: 'string',  

```

```
    required: false,  
    defaultsTo: ''  
  }  
}  
};
```

I believe the above code is self-explanatory. By default, Sails.js uses a local disk database which is basically a file located in the `.tmp` folder. In order to test our app, we need to create some users. The easiest way to do this is to install the sails-seed package:

```
npm install sails-seed --save
```

After installing, you'll find that the file `config/seeds.js` has been created for you. Paste the following seed data:

```
module.exports.seeds = {  
  user: [  
    {  
      name: 'John Wayne',  
      email: 'johnnie86@gmail.com',  
      avatar:  
        'https://randomuser.me/api/portraits/men/83.jpg',  
      location: 'Mombasa',  
      bio: 'Spends most of my time at the beach'  
    },  
    {  
      name: 'Peter Quinn',  
      email: 'peter.quinn@live.com',  
      avatar:  
        'https://randomuser.me/api/portraits/men/32.jpg',  
      location: 'Langley',  
      bio: 'Rather not say'  
    },  
    {  
      name: 'Jane Eyre',  
      email: 'jane@hotmail.com',  
      avatar:  
        'https://randomuser.me/api/portraits/women/94.jpg',  
      location: 'London',  
      bio: 'Loves reading motivation books'  
    }  
  ]  
}
```

Now that we've generated an API, we should configure the migration policy in the file `config/models.js`:

```
migrate: 'drop'
```

There are three migration strategies that Sails.js uses to determine how to rebuild your database every time it's started:

- **safe** — don't migrate, I'll do it by hand
- **alter** — migrate but try to keep the existing data
- **drop** — drop all tables and rebuild everything

I prefer to use `drop` for development, as I tend to iterate a lot. You can set `alter` if you'd like to keep existing data. Nevertheless, our database will be populated by the seed data every time.

Now let me show you something cool. Fire up your Sails project and navigate to the addresses `/user` and `/user/1`.



The image shows a web browser window with the address bar displaying "localhost:1337/user". The browser's title bar includes the name "Michael" and standard window controls. The main content area displays a JSON array of three user objects. Each object contains fields for name, email, avatar, location, bio, createdAt, updatedAt, and id. The users are John Wayne, Peter Quinn, and Jane Eyre.

```
[
  {
    "name": "John Wayne",
    "email": "johnnie86@gmail.com",
    "avatar": "https://randomuser.me/api/portraits/men/83.jpg",
    "location": "Mombasa",
    "bio": "Spends most of my time at the beach",
    "createdAt": "2018-01-21T21:19:40.472Z",
    "updatedAt": "2018-01-21T21:19:40.472Z",
    "id": 1
  },
  {
    "name": "Peter Quinn",
    "email": "peter.quinn@live.com",
    "avatar": "https://randomuser.me/api/portraits/men/32.jpg",
    "location": "Langley",
    "bio": "Rather not say",
    "createdAt": "2018-01-21T21:19:40.474Z",
    "updatedAt": "2018-01-21T21:19:40.474Z",
    "id": 2
  },
  {
    "name": "Jane Eyre",
    "email": "jane@hotmail.com",
    "avatar": "https://randomuser.me/api/portraits/women/94.jpg",
    "location": "London",
    "bio": "Loves reading motivation books",
    "createdAt": "2018-01-21T21:19:40.474Z",
    "updatedAt": "2018-01-21T21:19:40.474Z",
    "id": 3
  }
]
```




Thanks to the Sails.js Blueprints API, we have a fully functional CRUD API without us writing a single line of code. You can use Postman to access the User API and perform data manipulation such as creating, updating or deleting users.

Let's now proceed with building the profile form.

PROFILE FORM

Open `view/profile.ejs` and replace the existing TODO line with this code:

```

<div class="ui grid">
  <form action="<%= '/user/update/' + data.id %>"
method="post" class="ui centered form">
    <div class="field">
      <label>Name</label>
      <input type="text" name="name" value="<%= data.name
%>">
    </div>
    <div class="field">
      <label>Email</label>
      <input type="text" name="email" value="<%=
data.email %>">
    </div>
    <div class="field">
      <label>Location</label>
```

```

        <input type="text" name="location" value="<%=
data.location %>">
    </div>
    <div class="field">
        <label>Bio</label>
        <textarea name="bio" rows="4" cols="40"><%=
data.bio %></textarea>
    </div>
    <input type="hidden" name="avatar" value=
    <%=data.avatar %>>
    <button class="ui right floated orange button"
type="submit">Update</button>
    </form>
</div>

```

We're using Semantic-UI Form to build the form interface. If you examine the form's action value, `/user/update/' + data.id`, you'll realize that I'm using a Blueprint route. This means when a user hits the Update button, the Blueprint's update action will be executed.

However, for loading the user data, I've decided to define a custom action in the User Controller. Update the `api/controllers/UserController` with the following code:

```

module.exports = {

  render: async (request, response) => {
    try {
      let data = await User.findOne({
        email: 'johnnie86@gmail.com'
      });
      if (!data) {
        return response.notFound('The user was NOT
found!');
      }
      response.view('profile', { data });
    } catch (err) {
      response.serverError(err);
    }
  }
};

```

In this code you'll notice I'm using the `async/await` syntax to fetch the User data from the database. The alternative is to use callbacks, which for most developers is not clearly readable. I've

also hardcoded the default user account to load temporarily. Later, when we set up basic authentication, we'll change it to load the currently logged-in user.

Finally, we need to change the route `/profile` to start using the newly created `UserController`. Open `config/routes` and update the profile route as follows:

```
...  
'/profile': {  
  controller: 'UserController',  
  action: 'render'  
},  
...
```

Navigate to the URL `/profile`, and you should have the following view:

Sails Chat App

Michael


localhost:1337/profile

Chat Room

Profile

Logout

Account Profile



Name

John Wayne

Email

johnnie86@gmail.com

Location

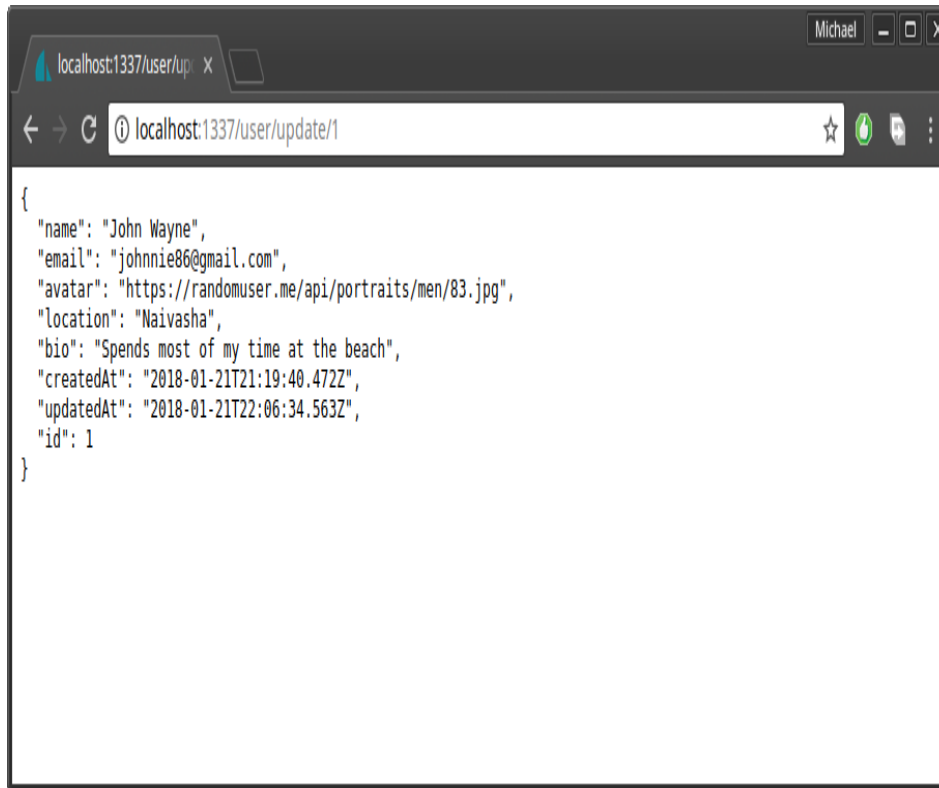
Mombasa

Bio

Spends most of my time at the beach

Update

Try changing one of the form fields and hit the update button. You'll be taken to this view:



You'll notice that the update has worked, but the data being displayed is in JSON format. Ideally, we should have a view-only profile page in `views/user/findOne.ejs` and an update profile page in `/views/user/update.ejs`. The Blueprint system will guess the views to use for rendering information. If it can't find the views it will just output JSON. For now, we'll simply use this neat trick. Create the file `/views/user/update.ejs` and paste the following code:

```
<script type="text/javascript">
window.location = '/profile';
</script>
```

Next time we perform an update, we'll be redirected to the /profile page. Now that we have user data, we can create the file views/partials/chat-users.js to be used in views/chatroom.ejs. After you've created the file, paste this code:

```
<div class="ui basic segment">
<h3>Members</h3>
<hr>
<div id="users-content" class="ui middle aligned
selection list"> </div>
</div>

// jsrender template
<script id="usersTemplate" type="text/x-jsrender">
<div class="item">
  
  <div class="content">
    <div class="header">{{:name}}</div>
  </div>
</div>
</script>

<script type="text/javascript">

function loadUsers() {
  // Load existing users
  io.socket.get('/user', function(users, response) {
    renderChatUsers(users);
  });

  // Listen for new & updated users
  io.socket.on('user', function(body) {
    io.socket.get('/user', function(users, response) {
      renderChatUsers(users);
    });
  });
}

function renderChatUsers(data) {
  const template = $.templates('#usersTemplate');
  let htmlOutput = template.render(data);
  $('#users-content').html(htmlOutput);
}

</script>
```

For this view, we need a client-side rendering approach to make the page update in real time. Here, we're making use of the jsrender library, a more powerful templating engine than EJS.

The beauty of `jsrender` is that it can either take an array or a single object literal and the template will still render correctly. If we were to do this in `ejs`, we'd need to combine an `if` statement and a `for` loop to handle both cases.

Let me explain the flow of our client-side JavaScript code:

1. `loadUsers()`. When the page first loads, we use the `Sails.js` socket library to perform a `GET` request for users. This request will be handled by the `Blueprint API`. We then pass on the data received to `renderChatUsers(data)` function.
2. Still within the `loadUsers()` function, we register a listener using `io.socket.on` function. We listen for events pertaining to the model `user`. When we get notified, we fetch the users again and replace the existing HTML output.
3. `renderChatUsers(data)`. Here we grab a script with the id `usersTemplate` using a `jQuery templates()` function. Notice the type is `text/x-jsrender`. By specifying a custom type, the browser will ignore and skip over that section since it doesn't know what it is. We then use the `template.render()` function to merge the template with data. This process will generate an HTML output which we then take and insert it into the HTML document.

The template we wrote in `profile.ejs` was rendered on the Node server, then sent to the browser as HTML. For the case of `chat-users`, we need to perform client-side rendering. This will allow chat users to see new users joining the group without them refreshing their browser.

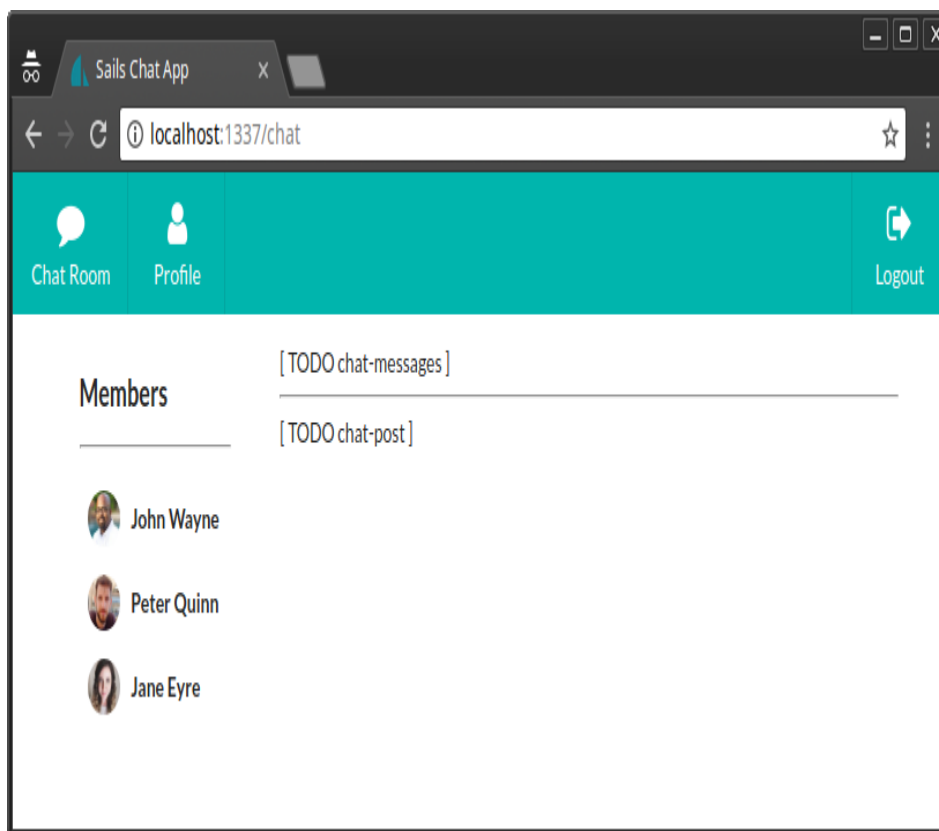
Before we test the code, we need to update `views/chatroom.ejs` to include the newly created `chat-users` partial. Replace `[TODO chat-users]` with this code:

```
...html
<% include partials/chat-users.ejs %>
...
```

Within the same file, add this script at the end:

```
<script type="text/javascript">
window.onload = function() {
  loadUsers();
}
</script>
```

This script will call the `loadUsers()` function. To confirm this is working, let's perform a `sails lift` and navigate to the `/chat` URL.



Your view should look like the image above. If it does, let's proceed with building the Chatroom API.

ChatMessage API

Same as before, we'll use Sails.js to generate the API:


```
sails generate api ChatMessage
```

Next, populate `api/models/ChatMessage.js` with these attributes:

```
module.exports = {  
  attributes: {  
    message: {  
      type: 'string',  
      required: true  
    },  
    createdBy : {  
      model: 'user',  
      required: true  
    }  
  }  
};
```

Notice that we've declared a one-to-one association with the User model through the `createdBy` attribute. Next we need to populate our disk database with a few chat messages. For that, we'll use `config/bootstrap.js`. Update the entire code as follows. We're using `async/await` syntax to simplify our code and avoid callback hell:

```
module.exports.bootstrap = async function(cb) {  
  sails.config.appName = "Sails Chat App";  
  
  // Generate Chat Messages  
  try {  
    let messageCount = ChatMessage.count();  
    if(messageCount > 0){  
      return; // don't repeat messages  
    }  
  
    let users = await User.find();  
    if(users.length >= 3) {  
      console.log("Generating messages...")  
  
      let msg1 = await ChatMessage.create({  
        message: 'Hey Everyone! Welcome to the community!',  
        createdBy: users[1]  
      });  
      console.log("Created Chat Message: " + msg1.id);  
    }  
  }  
}
```

```

    let msg2 = await ChatMessage.create({
      message: "How's it going?",
      createdBy: users[2]
    });
    console.log("Created Chat Message: " + msg2.id);

    let msg3 = await ChatMessage.create({
      message: 'Super excited!',
      createdBy: users[0]
    });
    console.log("Created Chat Message: " + msg3.id);

  } else {
    console.log('skipping message generation');
  }
} catch(err) {
  console.error(err);
}

// It's very important to trigger this callback method
when you're finished with Bootstrap! (Otherwise your
server will never lift, since it's waiting on Bootstrap)
cb();
};

```

The great thing is that the seeds generator runs before `bootstrap.js`. This way, we're sure `Users` data has been created first so that we can use it to populate the `createdBy` field. Having test data will enable us to quickly iterate as we build the user interface.

CHAT MESSAGES UI

Go ahead and create a new file `views/partials/chat-messages.ejs`, then place this code:

```

<div class="ui basic segment" style="height: 70vh;">
<h3>Community Conversations</h3>
<hr>
<div id="chat-content" class="ui feed"> </div>
</div>

<script id="chatTemplate" type="text/x-jsrender">
<div class="event">
  <div class="label">
    
  </div>
  <div class="content">

```

```

    <div class="summary">
      <a href="#"> {{:createdBy.name}}</a> posted on
      <div class="date">
        {{:createdAt}}
      </div>
    </div>
    <div class="extra text">
      {{:message}}
    </div>
  </div>
</div>
</script>

<script type="text/javascript">

function loadMessages() {
  // Load existing chat messages
  io.socket.get('/chatMessage', function(messages,
response) {
    renderChatMessages(messages);
  });

  // Listen for new chat messages
  io.socket.on('chatmessage', function(body) {
    renderChatMessages(body.data);
  });
}

function renderChatMessages(data) {
  const chatContent = $('#chat-content');
  const template = $.templates('#chatTemplate');
  let htmlOutput = template.render(data);
  chatContent.append(htmlOutput);
  // automatically scroll downwards
  const scrollHeight = chatContent.prop("scrollHeight");
  chatContent.animate({ scrollTop: scrollHeight },
"slow");
}

</script>

```

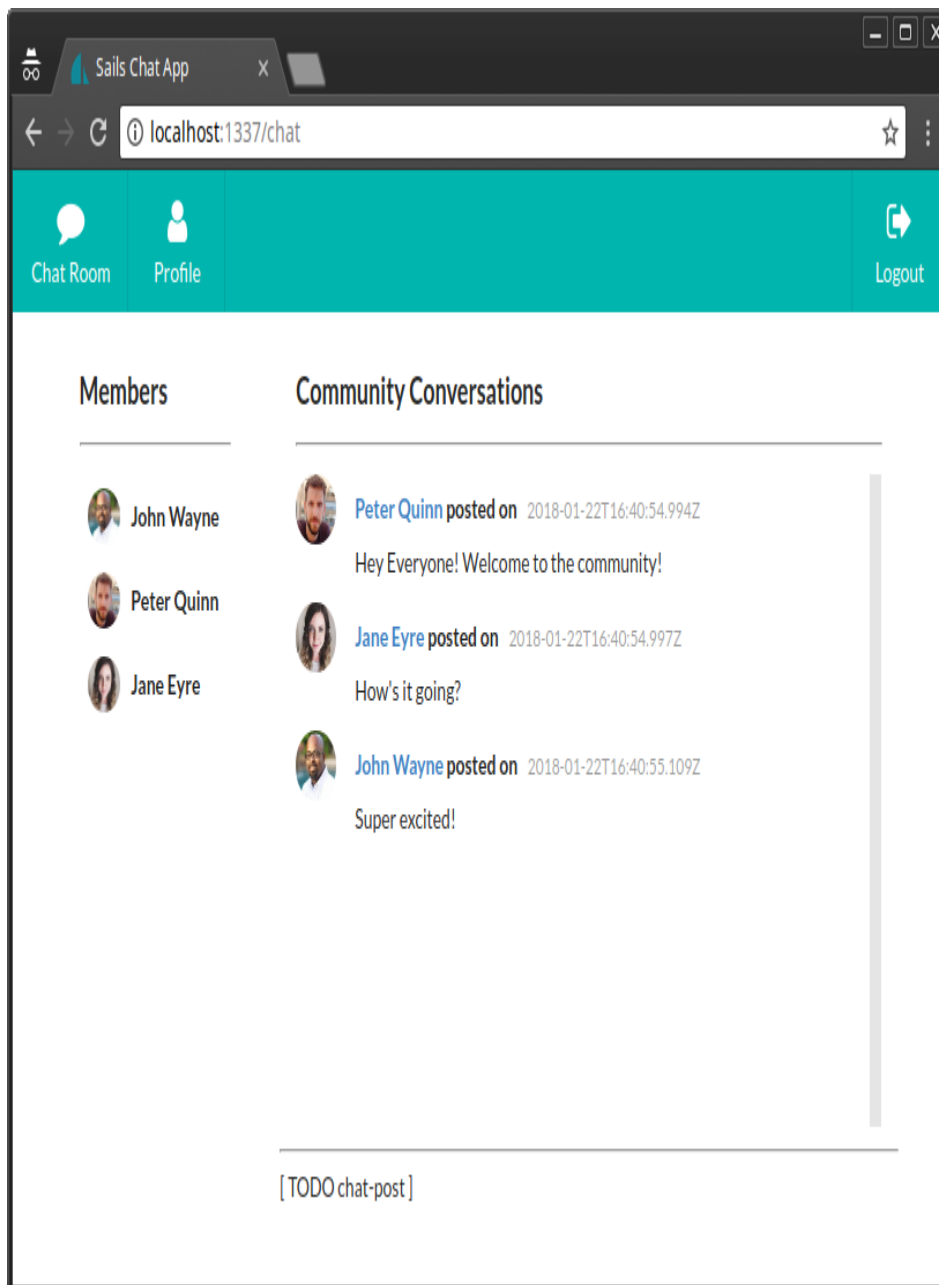
The logic here is very similar to chat-users. There's one key difference on the listen section. Instead of replacing the rendered output, we use append. Then we do a scroll animation to the bottom of the list to ensure users see the new incoming message.

Next, let's update chatroom.ejs to include the new chat-messages partial and also to update the script to call the loadMessages() function:

```
...
<!-- Chat Messages -->
    <% include partials/chat-messages.ejs %>
...

<script type="text/javascript">
...
    loadMessages();
...
</script>
```

Your view should now look like this:



Let's now build a simple form that will allow users to post messages to the chat room.

CHAT POST UI

Create a new file `views/partial/chat-post.ejs` and paste this code:

```

<div class="ui basic segment">
  <div class="ui form">
    <div class="ui field">
      <label>Post Message</label>
      <textarea id="post-field" rows="2"></textarea>
    </div>
    <button id="post-btn" class="ui right floated large
orange button" type="submit">Post</button>
  </div>
  <div id="post-err" class="ui tiny compact negative
message" style="display:none;">
    <p>Oops! Something went wrong.</p>
  </div>
</div>

```

Here we're using a using semantic-ui elements to build the form. Next add this script to the bottom of the file:

```

<script type="text/javascript">

function activateChat() {
  const postField = $('#post-field');
  const postButton = $('#post-btn');
  const postErr = $('#post-err');

  // Bind to click event
  postButton.click(postMessage);

  // Bind to enter key event
  postField.keypress(function(e) {
    var keycode = (e.keyCode ? e.keyCode : e.which);
    if (keycode == '13') {
      postMessage();
    }
  });

  function postMessage() {
    if(postField.val() == "") {
      alert("Please type a message!");
    } else {
      let text = postField.val();
      io.socket.post('/postMessage', { message: text },
function(resData, jwRes) {
      if(jwRes.statusCode != 200) {
        postErr.html("<p>" + resData.message + "
</p>")
        postErr.show();
      } else {
        postField.val(''); // clear input field
      }
    });
  }
}

```

```
}  
  
</script>
```

This script is made up of two functions:

- `activateChat()`. This function binds the post button to a click event and the message box (post field) to a key press (enter) event. When either is fired, the `postMessage()` function is called.
- `postMessage`. This function first does a quick validation to ensure the post input field is not blank. If there's a message is provided in the input field, we use the `io.socket.post()` function to send a message back to the server. Here we're using a classic callback function to handle the response from the server. If an error occurs, we display the error message. If we get a 200 status code, meaning the message was captured, we clear the post input field, ready for the next message to be typed in.

If you go back to the `chat-message` script, you'll see that we've already placed code to detect and render incoming messages. You should have also noticed that the `io.socket.post()` is sending data to the URL `/postMessage`. This is not a Blueprint route, but a custom one. Hence, we need to write code for it.

Head over to `api/controllers/UserController.js` and insert this code:

```
module.exports = {  
  
  postMessage: async (request, response) => {  
    // Make sure this is a socket request (not  
    traditional HTTP)  
    if (!request.isSocket) {  
      return response.badRequest();  
    }  
  
    try {  
      let user = await  
User.findOne({email:'johnnie86@gmail.com'});  
      let msg = await  
ChatMessage.create({message:request.body.message,  
createdBy:user });  
    }  
  }  
};
```

```

        if(!msg.id) {
            throw new Error('Message processing
failed!');
        }
        msg.createdBy = user;
        ChatMessage.publishCreate(msg);
    } catch(err) {
        return response.serverError(err);
    }

    return response.ok();
}
};

```

Since we haven't set up basic authentication, we are hardcoding the user `johnnie86@gmail.com` for now as the author of the message. We use the `Model.create()` Waterline ORM function to create a new record. This is a fancy way of inserting records without us writing SQL code. Next we send out a notify event to all sockets informing them that a new message has been created. We do that using the `ChatMessage.publishCreate()` function, which is defined in the Blueprints API. Before we send out the message, we make sure that the `createdBy` field is populated with a `user` object. This is used by `chat-messages` partial to access the avatar and the name of the user who created the message.

Next, head over to `config/routes.js` to map the `/postMessage` URL to the `postMessage` action we just defined. Insert this code:

```

...
'/chat': {
  view: 'chatroom'
}, // Add comma here
'/postMessage': {
  controller: 'ChatMessageController',
  action: 'postMessage'
}
...

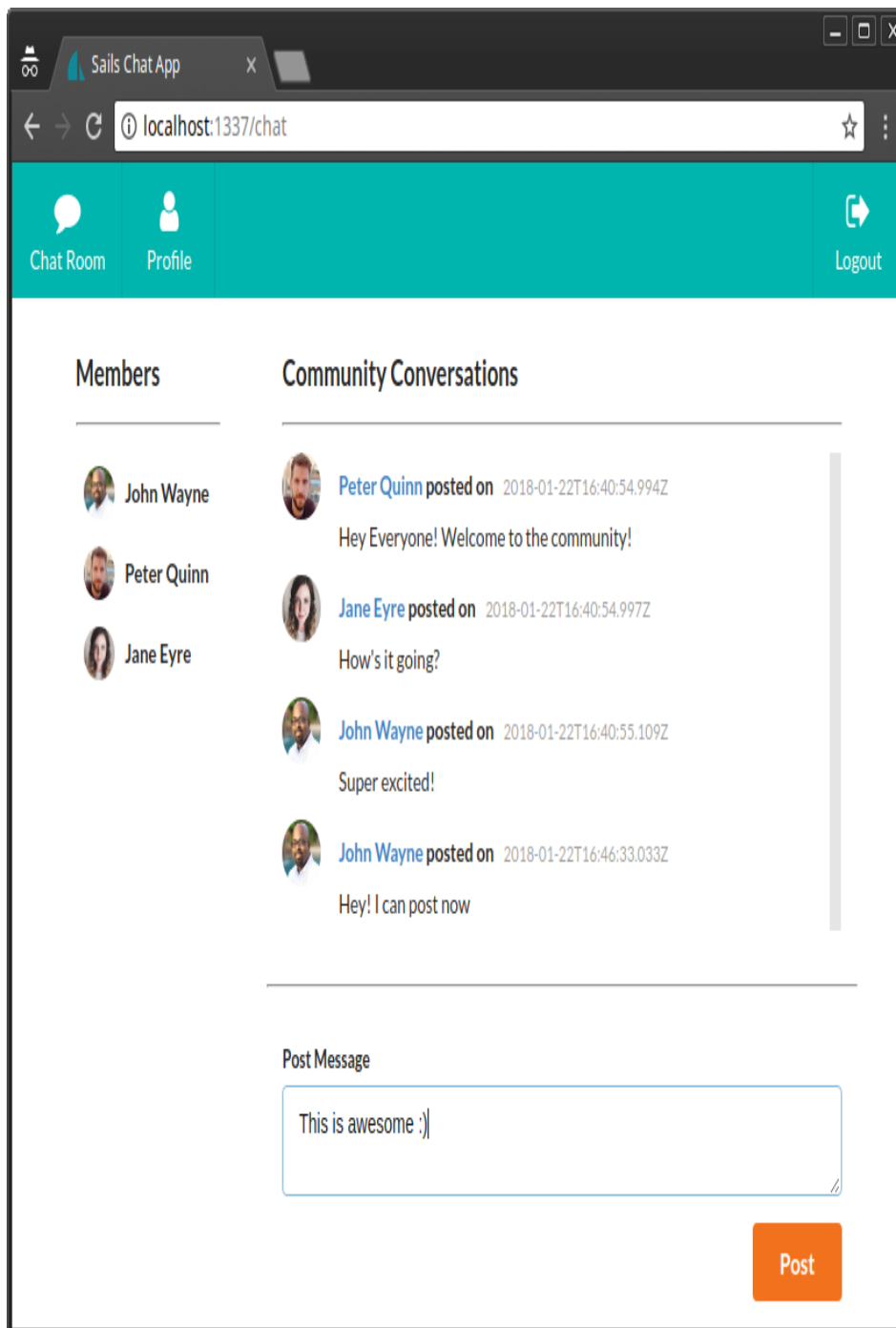
```

Open `views/chatroom.js` and include the `chat-post` partial. We'll also call the `activateChat()` function right

after the `loadMessages()` function:

```
...  
<% include partials/chat-messages.ejs %>  
...  
  
<script type="text/javascript">  
...  
    activateChat();  
...  
</script>
```

Refresh the page and try to send several messages.



You should now have a functional chat system. Review the project source code in case you get stuck.

Basic Authentication

Setting up a proper authentication and authorization system is outside the scope of this tutorial. So we'll settle for a basic password-less authentication system. Let's first build the signup and login form.

LOGIN/SIGN UP FORM

Create a new file `views/auth-form.ejs` and paste the following content:

```
<form method="post" action="/auth/authenticate" class="ui form">
  <div class="field">
    <label>Full Names</label>
    <input type="text" name="name" placeholder="Full Names"
    value="<%= typeof name != 'undefined' ? name : '' %>"
  </div>
  <div class="required field">
    <label>Email</label>
    <input type="email" name="email" placeholder="Email"
    value="<%= typeof email != 'undefined' ? email : '' %>"
  </div>
  <button class="ui teal button" type="submit"
  name="action" value="signup">Sign Up & Login</button>
  <button class="ui blue button" type="submit"
  name="action" value="login">Login</button>
  <p class="note">*Provide email only for Login</p>
</form>
<% if(typeof error != 'undefined') { %>
  <div class="ui error message">
    <div class="header"><%= error.title %></div>
    <p><%= error.message %></p>
  </div>
<% } %>
```

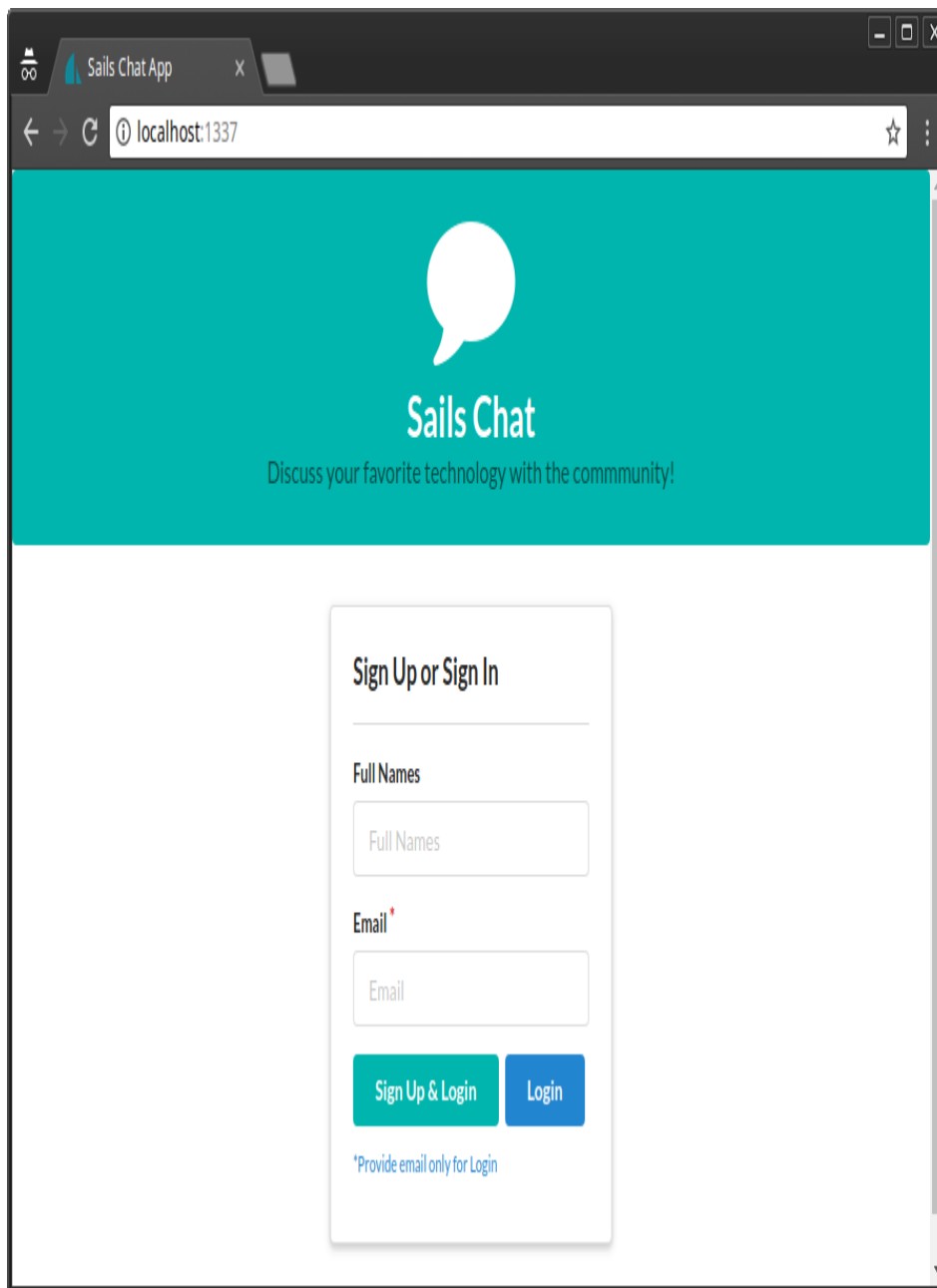
Next open `views/homepage.ejs` and replace the TODO line with this include statement:

```
...
<% include partials/auth-form.ejs %>
...
```

We've created a form that allows you to create a new account by providing an input for name and email. When you click Signup & Login, a new user record is created and you get logged in.

However, if the email is already being used by another user, an error message will be displayed. If you just want to log in, just provide the email address and click the `Login` button. Upon successful authentication, you'll be redirected to the `/chat` URL.

Right now, everything I've just said isn't working. We'll need to implement that logic. First, let's navigate to `/` address to confirm that the `auth-form` looks goods.



POLICY

Now that we're setting up an authentication system, we need to protect `/chat` and `/profile` routes from public access. Only authenticated users should be allowed to access them. Open `config/policies.js` and insert this code:

```
ChatMessageController: {
  '*': 'sessionAuth'
},

UserController: {
  '*': 'sessionAuth'
},
```

By specifying the name of the controller, we have also effectively blocked all routes provided by the Blueprint API for Users and Chat Messages. Unfortunately, policies only work with controllers. This means the route `/chat` can't be protected in its current state. We need to define a custom action for it. Open `api/controller/ChatroomController.js` and insert this code:

```
...
render: (request, response) => {
  return response.view('chatroom');
},
```

Then replace the route config for `/chat` with this one one `config/routes.js`:

```
...
'/chat': {
  controller: 'ChatMessageController',
  action: 'render'
},
...
```

The `/chat` route should now be protected from public access. If you restart your app and try to access `/profile`, `/chat`, `/user` or `/chatmessage`, you'll be met with the following forbidden message:



If you'd like to redirect users to the login form instead, head over to `api/policies/sessionAuth` and replace the

forbidden call with a redirect call like this:

```
...
// return res.forbidden('You are not permitted to perform
this action.');
```

```
return res.redirect('/');
...
```

Try accessing the forbidden pages again, and you'll automatically be redirected to the home page. Let's now implement the Signup and Login code.

AUTH CONTROLLER AND SERVICE

You'll need to stop Sails.js first in order to run this command:

```
sails generate controller Auth
```

This will create a blank `api/controllers/AuthController` for us. Open it and insert this code:

```
authenticate: async (request, response) => {

  // Sign up user
  if(request.body.action == 'signup') {
    // Validate signup form

    // Check if email is registered

    // Create new user
  }

  // Log in user
},

logout: (request, response) => {
  // Logout user
}
```

I've placed in comments explaining how the logic will flow. We can place the relevant code here. However, Sails.js recommends we keep our controller code simple and easy to follow. To

achieve this, we need to write helper functions that will help us with each of the above commented tasks. To create these helper functions, we need to create a service. Do this by creating a new file `api/services/AuthService.js`. Insert the following code:

```
/**
 * AuthService.js
 *
 */

const gravatar = require('gravatar')

// Where to display auth errors
const view = 'homepage';

module.exports = {

  sendAuthError: (response, title, message, options) => {
    options = options || {};
    const { email, name } = options;
    response.view(view, { error: {title, message}, email,
    name });
    return false;
  },

  validateSignupForm: (request, response) => {
    if(request.body.name == '') {
      return AuthService.sendAuthError(response, 'Signup
Failed!', "You must provide a name to sign up",
{email:request.body.email});
    } else if(request.body.email == '') {
      return AuthService.sendAuthError(response, 'Signup
Failed!', "You must provide an email address to sign up",
{name:request.body.name});
    }
    return true;
  },

  checkDuplicateRegistration: async (request, response) =>
  {
    try {
      let existingUser = await
User.findOne({email:request.body.email});
      if(existingUser) {
        const options = {email:request.body.email,
name:request.body.name};
        return AuthService.sendAuthError(response,
'Duplicate Registration!', "The email provided has
already been registered", options);
      }
      return true;
    } catch (err) {
```

```

        response.serverError(err);
        return false;
    }
},

registerUser: async (data, response) => {
    try {
        const {name, email} = data;
        const avatar = gravatar.url(email, {s:200}, "https");
        let newUser = await User.create({name, email,
avatar});
        // Let all sockets know a new user has been created
        User.publishCreate(newUser);
        return newUser;
    } catch (err) {
        response.serverError(err);
        return false;
    }
},

login: async (request, response) => {
    try {
        let user = await
User.findOne({email:request.body.email});
        if(user) { // Login Passed
            request.session.userId = user.id;
            request.session.authenticated = true;
            return response.redirect('/chat');
        } else { // Login Failed
            return AuthService.sendAuthError(response, 'Login
Failed!', "The email provided is not registered",
{email:request.body.email});
        }
    } catch (err) {
        return response.serverError(err);
    }
},

logout: (request, response) => {
    request.session.userId = null;
    request.session.authenticated = false;
    response.redirect('/');
}
}

```

Examine the code carefully. As an intermediate developer, you should be able to understand the logic. I haven't done anything fancy here. However, I would like to mention a few things:

- Gravatar. You need to install Gravatar. It's a JavaScript library for generating Gravatar URLs based on the email address.

```
```bash
npm install gravatar --save
```
```

- `User.publishCreate(newUser)`. Just like `ChatMessages`, we fire an event notifying all sockets that a new user has just been created. This will cause all logged-in clients to re-fetch the users data. Review `views/partial/chat-users.js` to see what I'm talking about.
- `request.session`. Sails.js provides us with a session store which we can use to pass data between page requests. The default Sails.js session lives in memory, meaning if you stop the server the session data gets lost. In the `AuthService`, we're using session to store `userId` and `authenticated status`.

With the logic in `AuthService.js` firmly in place, we can go ahead and update `api/controllers/AuthController` with the following code:

```
module.exports = {

  authenticate: async (request, response) => {
    const email = request.body.email;

    if(request.body.action == 'signup') {
      const name = request.body.name;
      // Validate signup form
      if(!AuthService.validateSignupForm(request,
response)) {
        return;
      }
      // Check if email is registered
      const duplicateFound = await
AuthService.checkDuplicateRegistration(request,
response);
      if(!duplicateFound) {
        return;
      }
      // Create new user
      const newUser = await
AuthService.registerUser({name,email}, response);
      if(!newUser) {
        return;
      }
    }

    // Attempt to log in
```

```
    const success = await AuthService.login(request,
response);
  },

  logout: (request, response) => {
    AuthService.logout(request, response);
  }
};
```

See how much simple and readable our controller is. Next, let's do some final touches.

Final Touches

Now that we have authentication set up, we should remove the hardcoded value we placed in the `postMessage` action in `api/controllers/ChatMessageController`. Replace the email code with this one:

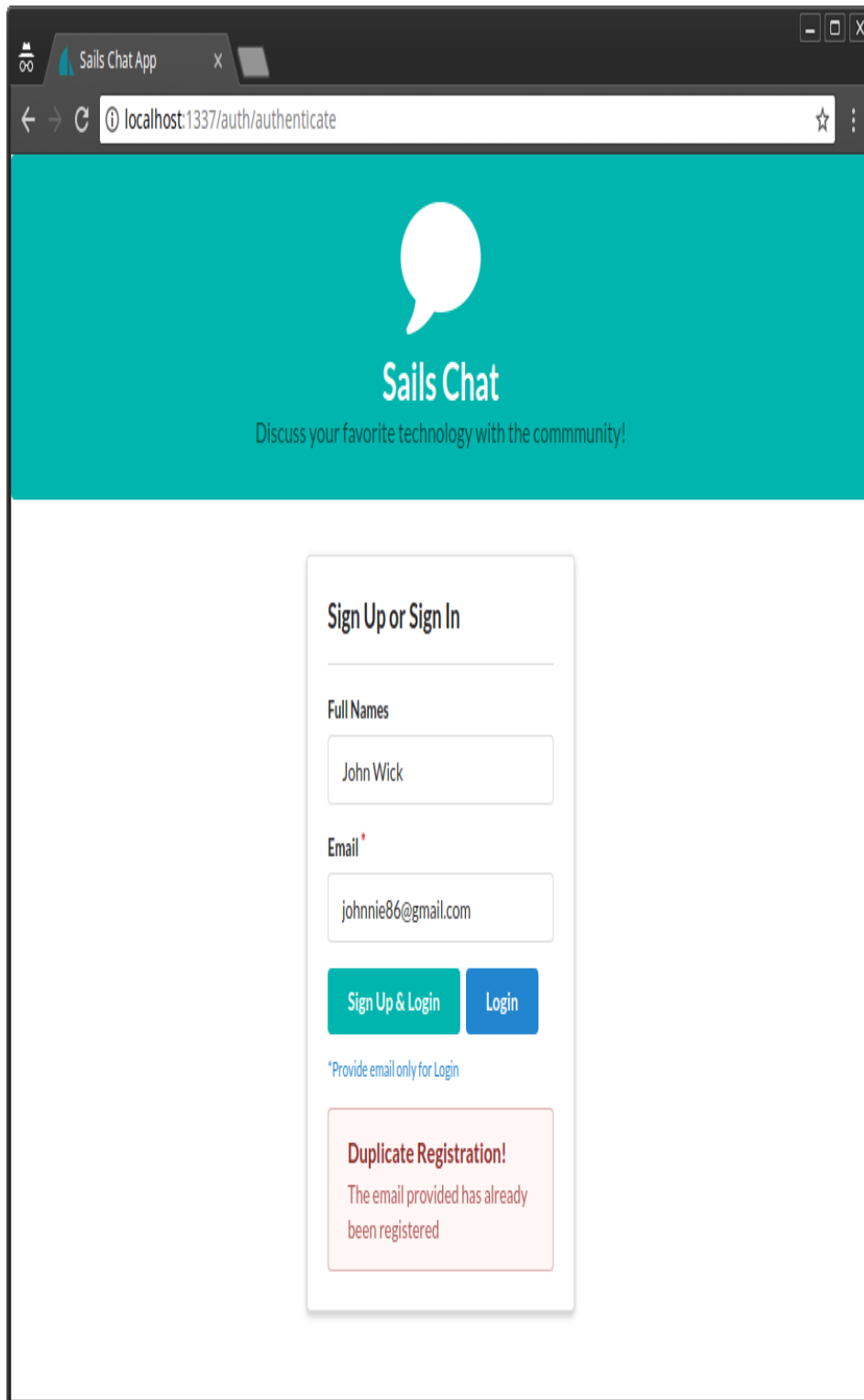
```
...
let user = await
User.findOne({id:request.session.userId});
...
```

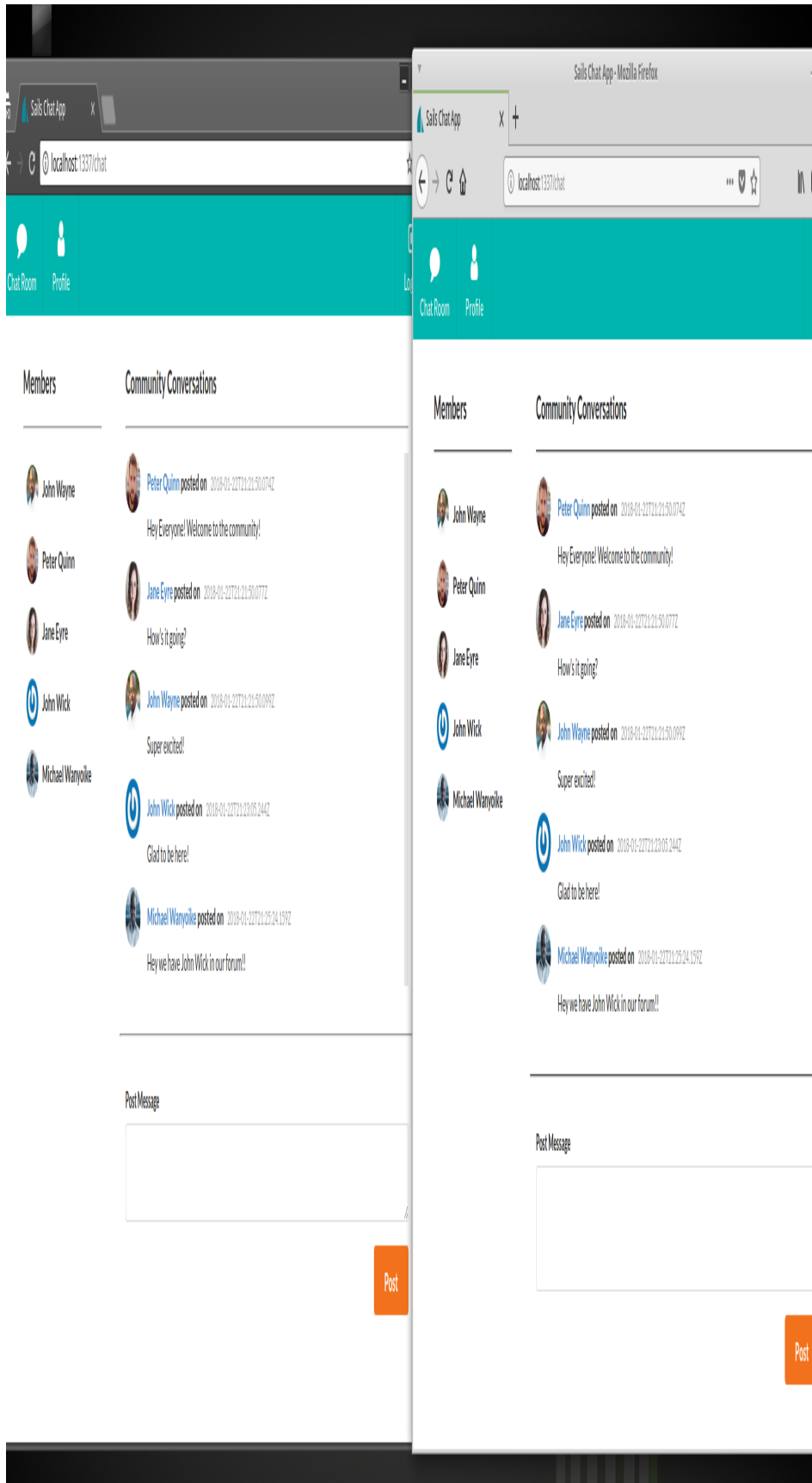
I'd like to mention something you may not have noticed, if you look at the `logout` URL in `views/partials/menu.ejs`, we've placed this address `/auth/logout`. If you look at `config/routes.js`, you'll notice that we haven't placed a URL for it. Surprisingly, when we run the code, it works. This is because Sails.js uses a convention to determine which controller and action is needed to resolve a particular address.

By now you should be having a functional MVP chat application. Fire up your app and test the following scenarios:

- sign up without entering anything
- sign up by filling only name
- sign up by only filling email

- sign up by filling name and a registered email — for example,
`johnnie86@gmail.com` or `jane@hotmail.com`
- sign up using your name and email
- update your profile
- try posting a blank message
- post some messages
- open another browser and log in as another user, put each browser
side by side and chat
- log out and create a new account.





Phew! That's a lot of functionality we've just implemented in one sitting and then tested. With a few more weeks, we could whip out a production ready chat system integrated with more features, such as multiple chat rooms, channel attachments, smiley icons and social accounts integration!

Summary

During this tutorial, we didn't put the name of the logged-in user somewhere at the top menu. You should be capable of fixing this yourself. If you've read the entire tutorial, you should now be proficient in building applications using Sails.js.

The goal of this tutorial is to show you that can come from a non-JavaScript MVC framework and build something awesome with relatively few lines of code. Making use of the Blueprint API will help you implement features faster. I also recommend you learn to integrate a more powerful front-end library — such as React, Angular or Vue — to create a much more interactive web application. In addition, learning how to write tests for Sails.js to automate the testing process is a great weapon in your programming arsenal.

Chapter 7: Passport Authentication for Node.js Applications

BY PAUL ORAC

In this tutorial, we'll be implementing authentication via Facebook and GitHub in a Node.js web application. For this, we'll be using Passport, an authentication middleware for Node.js. Passport supports authentication with OpenId/OAuth providers.

Express Web App

Before getting started, make sure you have Node.js installed on your machine.

We'll begin by creating the folder for our app and then accessing that folder on the terminal:

```
mkdir AuthApp
cd AuthApp
```

To create the node app we'll use the following command:

```
npm init
```

You'll be prompted to provide some information for Node's `package.json`. Just hit `enter` until the end to leave the default configuration.

Next, we'll need an HTML file to send to the client. Create a file called `auth.html` in the root folder of your app, with the following contents:

```
<html>
  <head>
    <title>Node.js OAuth</title>
  </head>
  <body>
    <a href=auth/facebook>Sign in with Facebook</a>
    <br></br>
    <a href=auth/github>Sign in with Github</a>
  </body>
</html>
```

That's all the HTML we'll need for this tutorial.

You'll also require Express, a framework for building web apps that's inspired by Ruby's Sinatra. In order to install Express, from the terminal type the following command:

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

Once you've done that, it's time to write some code.

Create a file `index.js` in the root folder of your app and add the following content to it:

```
/* EXPRESS SETUP */

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

app.get('/', (req, res) => res.sendFile('auth.html', {
  root : __dirname}));

const port = process.env.PORT || 3000;
app.listen(port, () => console.log('App listening on
port ' + port));
```

In the code above, we require Express and create our Express app by calling `express()`. Then we declare the route for the home page of our app. There we send the HTML file we've created to

the client accessing that route. Then, we use `process.env.PORT` to set the port to the environment port variable if it exists. Otherwise, we'll default to `3000`, which is the port we'll be using locally. This gives you enough flexibility to switch from development, directly to a production environment where the port might be set by a service provider like, for instance, [Heroku](#). Right below, we call `app.listen()` with the port variable we set up, and a simple log to let us know that it's all working fine, and on which port is the app listening.

Now we should start our app to make sure all is working correctly. Simply write the following command on the terminal:

```
node index.js
```

You should see the message: `App listening on port 3000`. If that's not the case, you probably missed a step. Go back and try again.

Moving on, let's see if our page is being served to the client. Go to your web browser and navigate to `http://localhost:3000`.

If you can see the page we created in `auth.html`, we're good to go.

Head back to the terminal and stop the app with `ctrl + c`. So remember, when I say start the app, you write `node index.js`, and when I say stop the app, you do `ctrl + c`. Clear? Good, you've just been programmed :-)

Setting up Passport

As you'll soon come to realize, Passport makes it a breeze to provide authentication for our users. Let's install Passport with

the following command:

```
npm install passport --save
```

Now we have to set up Passport. Add the following code at the bottom of the `index.js` file:

```
/*  PASSPORT SETUP  */

const passport = require('passport');
app.use(passport.initialize());
app.use(passport.session());

app.get('/success', (req, res) => res.send("You have
successfully logged in"));
app.get('/error', (req, res) => res.send("error logging
in"));

passport.serializeUser(function(user, cb) {
  cb(null, user);
});

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

Here we require Passport and initialize it along with its session authentication middleware, directly inside our Express app. Then, we set up the `'/success'` and `'/error'` routes, which will render a message telling us how the authentication went. It's the same syntax for our last route, only this time instead of using `[res.sendFile()]` (<http://expressjs.com/en/api.html#res.sendFile>) we're using `[res.send()]` (<http://expressjs.com/en/api.html#res.send>), which will render the given string as `text/html` in the browser. Then we're using `serializeUser` and `deserializeUser` callbacks. The first one will be invoked on authentication and its job is to serialize the user instance and store it in the session via a cookie. The second one will be invoked every subsequent request to deserialize the instance,

providing it the unique cookie identifier as a “credential”. You can read more about that in the Passport documentation.

As a side note, this very simple sample app of ours will work just fine without `deserializeUser`, but it kills the purpose of keeping a session, which is something you’ll need in every app that requires login.

That’s all for the actual Passport setup. Now we can finally get onto business.

Implementing Facebook Authentication

The first thing we'll need to do in order to provide *Facebook Authentication* is installing the `passport-facebook` package. You know how it goes:

```
npm install passport-facebook --save
```

Now that everything's set up, adding *Facebook Authentication* is extremely easy. Add the following code at the bottom of your `index.js` file:

```
/* FACEBOOK AUTH */

const FacebookStrategy = require('passport-
facebook').Strategy;

const FACEBOOK_APP_ID = 'your app id';
const FACEBOOK_APP_SECRET = 'your app secret';

passport.use(new FacebookStrategy({
  clientID: FACEBOOK_APP_ID,
  clientSecret: FACEBOOK_APP_SECRET,
  callbackURL: "/auth/facebook/callback"
},
function(accessToken, refreshToken, profile, cb) {
  return cb(null, profile);
}
));

app.get('/auth/facebook',
  passport.authenticate('facebook'));

app.get('/auth/facebook/callback',
  passport.authenticate('facebook', { failureRedirect:
'/error' })),
function(req, res) {
  res.redirect('/success');
});
```

Let's go through this block of code step by step. First, we require the `passport-facebook` module. Then, we declare the

variables in which we'll store our *app id* and *app secret* (we'll see how to get those shortly). After that, we tell Passport to use an instance of the `FacebookStrategy` we required. To instantiate said strategy we give it our *app id* and *app secret* variables and the `callbackURL` that we'll use to authenticate the user. As a second parameter, it takes a function that will return the profile info provided by the user.

Further down, we set up the routes to provide authentication. As you can see in the `callbackURL` we redirect the user to the `/error` and `/success` routes we defined earlier. We're using `passport.authenticate`, which attempts to authenticate with the given strategy on its first parameter, in this case `facebook`. You probably noticed that we're doing this twice. On the first one, it sends the request to our Facebook app. The second one is triggered by the callback URL, which Facebook will use to respond to the login request.

Now you'll need to create a Facebook app. For details on how to do that, consult Facebook's very detailed guide [Creating a Facebook App](#), which provides step by step instructions on how to create one.

When your app is created, go to *Settings* on the app configuration page. There you'll see your *app id* and *app secret*. Don't forget to change the variables you declared for them on the `index.js` file with their corresponding values.

Next, enter "localhost" in the *App Domains* field. Then, go to *Add platform* at the bottom of the page and choose *Website*. Use `http://localhost:3000/auth/facebook/callback` as the *Site URL*.

On the left sidebar, under the *Products* section, you should see *Facebook Login*. Click to get in there.

Lastly, set the *Valid OAuth redirect URIs* field to `http://localhost:3000/auth/facebook/callback`.

If you start the app now and click the *Sign in with Facebook* link, you should be prompted by Facebook to provide the required information, and after you've logged in, you should be redirected to the `/success` route, where you'll see the message `You have successfully logged in`.

That's it! you have just set up *Facebook Authentication*. Pretty easy, right?

Implementing GitHub Authentication

The process for adding *GitHub Authentication* is quite similar to what we did for Facebook. First, we'll install the `passport-github` module:

```
npm install passport-github --save
```

Now go to the `index.js` file and add the following lines at the bottom:

```
/* GITHUB AUTH */

const GitHubStrategy = require('passport-github').Strategy;

const GITHUB_CLIENT_ID = "your app id"
const GITHUB_CLIENT_SECRET = "your app secret";

passport.use(new GitHubStrategy({
  clientID: GITHUB_CLIENT_ID,
  clientSecret: GITHUB_CLIENT_SECRET,
  callbackURL: "/auth/github/callback"
},
function(accessToken, refreshToken, profile, cb) {
  return cb(null, profile);
}
));
```



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

app.get('/auth/github/callback',
  passport.authenticate('github', { failureRedirect:
    '/error' })),
  function(req, res) {
    res.redirect('/success');
  });
```

This looks familiar! It's practically the same as before. The only difference is that we're using the *GithubStrategy* instead of *FacebookStrategy*.

So far so ... the same. In case you hadn't yet figured it out, the next step is to create our *GitHub App*. GitHub has a very simple guide, [Creating a GitHub app](#), that will guide you through the process.

When you're done, in the configuration panel you'll need to set the *Homepage URL* to `http://localhost:3000/` and the *Authorization callback URL* to `http://localhost:3000/auth/github/callback`, just like we did with Facebook.

Now, simply restart the Node server and try logging in using the GitHub link.

It works! Now you can let your users log in with GitHub.

Conclusion

In this tutorial, we saw how Passport made the task of authentication quite simple. Implementing Google and Twitter authentication follows a nearly identical pattern. I challenge you to implement these using the [passport-google](#) and [passport-twitter](#) modules. In the meantime, the code for this app is available on [GitHub](#).

Chapter 8: Local Authentication Using Passport in Node.js

BY PAUL ORAC

In the last chapter, we talked about authentication using Passport as it relates to social login (Google, Facebook, GitHub, etc.). In this chapter, we'll see how we can use Passport for local authentication with a MongoDB back end.

All of the code from this article is available for download on GitHub.

Prerequisites

- Node.js — Download and install Node.js.
- MongoDB — Download and install MongoDB Community Server. Follow the instructions for your OS. Note, if you're using Ubuntu, this guide can help you get Mongo up and running.

Creating the Project

Once all of the prerequisite software is set up, we can get started.

We'll begin by creating the folder for our app and then accessing that folder on the terminal:

```
mkdir AuthApp
cd AuthApp
```

To create the node app, we'll use the following command:

```
npm init
```

You'll be prompted to provide some information for Node's `package.json`. Just hit `enter` until the end to leave the default configuration.

HTML

Next, we'll need a form with `username` and `password` inputs as well as a *Submit* button. Let's do that! Create a file called `auth.html` in the root folder of your app, with the following contents:

```
<html>
  <body>
    <form action="/" method="post">
      <div>
        <label>Username:</label>
        <input type="text" name="username" />
        <br/>
      </div>
      <div>
        <label>Password:</label>
        <input type="password" name="password" />
      </div>
      <div>
        <input type="submit" value="Submit" />
      </div>
    </form>
  </body>
</html>
```

That will do just fine.

Setting up Express

Now we need to install Express, of course. Go to the terminal and write this command:

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

We'll also need to install the body-parser middleware which is used to parse the request body that Passport uses to authenticate the user.

Let's do that. Run the following command:

```
npm install body-parser --save
```

When that's done, create a file `index.js` in the root folder of your app and add the following content to it:

```
/* EXPRESS SETUP */

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

const bodyParser = require('body-parser');
app.use(bodyParser.urlencoded({ extended: true }));

app.get('/', (req, res) => res.sendFile('auth.html', {
  root : __dirname}));

const port = process.env.PORT || 3000;
app.listen(port, () => console.log('App listening on
port ' + port));
```

We're doing almost the same as in the previous tutorial. First we require Express and create our Express app by calling `[express()]` (<http://expressjs.com/en/api.html#express>). The next line is the only difference with our previous Express setup. We'll need the body-parser middleware this time, in order for authentication to work correctly. Then we declare the route for the home page of our app. There we send the HTML file we created to the client accessing that route. Then, we use

`process.env.PORT` to set the port to the environment port variable if it exists. Otherwise, we'll default to 3000, which is the port we'll be using locally. This gives you enough flexibility to switch from development, directly to a production environment where the port might be set by a service provider like, for instance, Heroku. Right below we called

```
[app.listen()]
```

(<http://expressjs.com/en/api.html#app.listen>) with the port variable we set up and a simple log to let us know that it's all working fine and on which port is the app listening.

That's all for the Express setup. Now we have to set up Passport, exactly as we did the last time. I'll show you how to do that in case you didn't read the previous tutorial.

Setting up Passport

First, we install *passport* with the following command:

```
npm install passport --save
```

Then, add the following lines at the bottom of your `index.js` file:

```
/*  PASSPORT SETUP  */

const passport = require('passport');
app.use(passport.initialize());
app.use(passport.session());

app.get('/success', (req, res) => res.send("Welcome "+req.query.username+"!!"));
app.get('/error', (req, res) => res.send("error logging in"));

passport.serializeUser(function(user, cb) {
  cb(null, user.id);
});

passport.deserializeUser(function(id, cb) {
  User.findById(id, function(err, user) {
```

```
    cb(err, user);  
  });  
});
```

Here, we require passport and initialize it along with its session authentication middleware, directly inside our Express app. Then, we set up the `'/success'` and `'/error'` routes which will render a message telling us how the authentication went. If it succeeds, we're going to show the `username` parameter, which we'll pass to the request. We're using the same syntax for our last route, only this time instead of using

```
[res.sendFile()]  
(http://expressjs.com/en/api.html#res.sendFile)  
we're using [res.send()]  
(http://expressjs.com/en/api.html#res.send),  
which will render the given string as text/html on the  
browser. Then we're using serializeUser and  
deserializeUser callbacks. The first one will be invoked on  
authentication, and its job is to serialize the user instance with  
the information we pass to it (the user ID in this case) and store  
it in the session via a cookie. The second one will be invoked  
every subsequent request to deserialize the instance, providing  
it the unique cookie identifier as a “credential”. You can read  
more about that in the Passport documentation.
```

As a side note, this very simple sample app of ours will work just fine without `deserializeUser`, but it kills the purpose of keeping a session, which is by all means something you'll need in every app that requires login.

Creating a MongoDB Data Store

Since we're assuming you've already installed Mongo, you should be able to start the `mongod` server using the following command:

```
sudo mongod
```

From another terminal, launch the Mongo shell:

```
mongo
```

Within the shell, issue the following commands:

```
use MyDatabase;  
  
db.userInfo.insert({'username':'admin','password':'admin'  
});
```

The first command creates a data store named `MyDatabase`. The second command creates a collection named `userInfo` and inserts a record. Let's insert a few more records:

```
db.userInfo.insert({'username':'jay','password':'jay'});  
db.userInfo.insert({'username':'roy','password':'password'  
});
```

RETRIEVING STORED DATA

We can view the data we just added using the following command:

```
db.userInfo.find();
```

The resulting output is shown below:

```
{ "_id" : ObjectId("5321cd6dbb5b0e6e72d75c80"),  
  "username" : "admin", "password" : "admin" }  
{ "_id" : ObjectId("5321d3f8bb5b0e6e72d75c81"),  
  "username" : "jay", "password" : "jay" }  
{ "_id" : ObjectId("5321d406bb5b0e6e72d75c82"),  
  "username" : "roy", "password" : "password" }
```

We can also search for a particular username and password:


```
db.userInfo.findOne({'username':'admin','password':'admin'})
```

This command would return only the admin user.

Connection Mongo to Node with Mongoose

Now that we have a database with records in it, we need a way to communicate with it from our application. We'll be using Mongoose to achieve this. Why don't we just use plain Mongo? Well, as the Mongoose devs like to say on their website:

writing MongoDB validation, casting and business logic boilerplate is a drag.

Mongoose will simply make our lives easier and our code more elegant.

Let's go ahead and install it with the following command:

```
npm install mongoose --save
```

Now we have to configure Mongoose. You know the drill: add the following code at the bottom of your `index.js` file:

```
/* MONGOOSE SETUP */

const mongoose = require('mongoose');
mongoose.connect('mongodb://localhost/MyDatabase');

const Schema = mongoose.Schema;
const UserDetails = new Schema({
  username: String,
  password: String
});
const UserDetails = mongoose.model('userInfo',
UserDetail, 'userInfo');
```

Here we require the `mongoose` package. Then we connect to our database using `mongoose.connect` and give it the path to our database. Then we're making use of `Schema` to define our data structure. In this case, we're creating a `UserDetail` schema with `username` and `password` fields. After that, we create a `model` from that schema. The first parameter is the name of the collection in the database. The second one is the reference to our `Schema`, and the third one is the name we're assigning to the collection inside `Mongoose`.

That's all for the `Mongoose` setup. We can now move on to implementing our `Strategy`.

Implementing Local Authentication

It's time to configure our authentication strategy using `passport-local`. Let's go ahead and install it. Run the following command:

```
npm install passport-local --save
```

Finally, we've come to the last portion of the code! Add it to the bottom of your `index.js` file:

```
/* PASSPORT LOCAL AUTHENTICATION */

const LocalStrategy = require('passport-local').Strategy;

passport.use(new LocalStrategy(
  function(username, password, done) {
    UserDetails.findOne({
      username: username
    }, function(err, user) {
      if (err) {
        return done(err);
      }

      if (!user) {
        return done(null, false);
      }
    });
  }
));
```

```

        if (user.password !== password) {
            return done(null, false);
        }
        return done(null, user);
    });
}
));

app.post('/',
  passport.authenticate('local', { failureRedirect:
    '/error' }),
  function(req, res) {
    res.redirect('/success?username='+req.user.username);
  });

```

Let's go through this. First, we require the `passport-local` Strategy. Then we tell Passport to use an instance of the `LocalStrategy` we required. There we simply use the same command that we used in the Mongo shell to find a record based on the username. If a record is found and the password matches, the above code returns the `user` object. Otherwise, it returns `false`.

Below the strategy implementation is our `post`, with the `[passport.authenticate]` (<http://www.passportjs.org/docs/authenticate/>) method which attempts to authenticate with the given strategy on its first parameter, in this case `'local'`. It will redirect us to `'/error'` if it fails. Otherwise it'll redirect us to the `'/success'` route, sending the username as a parameter. That's how we get the username to show on the line `req.query.username` we saw earlier.

That's all we need for the app to work. We're done!

Restart your Node server and point your browser to `http://localhost:3000/` and try to log in with "admin" as username and "admin" as password. If it all goes well, you should see the message "Welcome admin!!" in the browser.

Conclusion

In this article, we learned how to implement local authentication using Passport in a Node.js application. In the process, we also learned how to connect to MongoDB using the Mongoose.

We only added the necessary modules for this app to work — nothing more, nothing less. For a production app, you'll need to add other middlewares and separate your code in modules. You can take that as a challenge to set up a clean and scalable environment and grow it into something useful.

Chapter 9: An Introduction to NodeBots

BY PATRICK CATANZARITI

Many web developers out there would love the chance to build an incredibly cool robot that they can control via JavaScript, right? I'm here to tell you that this is already possible today! Right now.

NodeBots have been around for a while, and the community around them is growing like wildfire. In this article, I'm going to explain what NodeBots are, how they work and how you can get started tinkering away at robot creation.

What is a Microcontroller?

Before I get too far into things, we'll be mentioning microcontrollers quite frequently. A microcontroller is a tiny and very simple computer. It has a simple physical programmable circuit board that can detect various inputs and send outputs. An Arduino is a type of microcontroller. It's actually one of the most common ones for newcomers to experiment with. There are other sorts of microcontrollers too that can be powered by Node, including Particle boards (my favorite!), BeagleBone boards, Tessel boards (the board itself runs on JS) and Espruino boards (also runs on JS). In this article, I'll be focusing on Arduinos, as they're the most common.

What are NodeBots?

NodeBots are (quite literally) robots of one kind or another that can be controlled via Node. They can have everything from wheels, movable arms and legs, motion detectors, cameras, LED displays, the ability to play sound and so much more. The only limits are your imagination and the components you can find and put together!

The whole idea of NodeBots evolved through the increasing capabilities of Node.js and the interest of a few developers like [Nikolai Onken](#), [Jörn Zaefferer](#), [Chris Williams](#), [Julian Gautier](#) and [Rick Waldron](#) who worked to develop the various Node modules we use in NodeBots today. The Node package called [node-serialport](#) by Chris Williams started it all, allowing access to real world devices via reading and writing to serial ports at a low level.

Julian Gautier then implemented the Firmata protocol, a protocol used to access microcontrollers like Arduinos via software on a computer, using JavaScript in his Node.js [Firmata](#) library.

Rick Waldron took it a massive step further. Using the Firmata library as a building block, he created a whole JavaScript Robotics and IoT programming framework called [Johnny-Five](#). The Johnny-Five framework makes controlling everything from LEDs to various types of sensors relatively simple and painfree. This is what many NodeBots now use to achieve some very impressive feats!

Where To Start

If you're completely new to the idea of building robots and any sort of real-world, JavaScript-controlled device, there are plenty

of incredible resources for you to get started with. The very first thing I'd recommend you do is find yourself a good Arduino kit that provides a good range of components and sensors to give you a range of items to play around with. Below, I've got a list of some of the Arduino starter kits that are available from various companies. If the below list looks overwhelming, don't worry! They all contain very similar components and are all a good choice for beginners.

STARTER KITS

- **SparkFun Inventors Kit.** This is the kit that started it all for me years ago! It comes with a range of standard components like colored LED lights, sensors, buttons, a motor, a tiny speaker and more. It also comes with a guide and sample projects you can use to build your skills. You can find it here: [SparkFun Inventor's Kit](#).
- **Freetronics Experimenter's Kit for Arduino.** This kit is by an Australian-based company called Freetronics. It has very similar components to the SparkFun one, with a few small differences. It also has its own guide with sample projects to try as well. For those based in Australia, these kits and other Freetronics parts are available at Jaycar. You can also order it online here: [Freetronics Experimenter's Kit](#).
- **Seeed Studio ARDX starter kit.** Seeed Studio have their own starter kit too, which is also very similar to the SparkFun and Freetronics ones. It has its own guide and such too! You can find it here: [ARDX - The starter kit for Arduino](#).
- **Adafruit ARDX Experimentation Kit for Arduino.** This kit is also very similar to the ones above with its own guide. You can find it here: [Adafruit ARDX Experimentation Kit for Arduino](#).
- **Arduino Starter Kit.** The guys at Arduino.cc have their own official kit that's available too. The starter kit is similar to the ones above but has some interesting sample projects like a "Love-O-Meter". You can find it here and often at other resellers too: [Arduino Starter Kit](#).

With all of the above kits, keep in mind that none of them are targeted towards NodeBot development. So the examples in

booklets and such are written in the simplified C++ code that Arduino uses. For examples using Node, see the resources below.

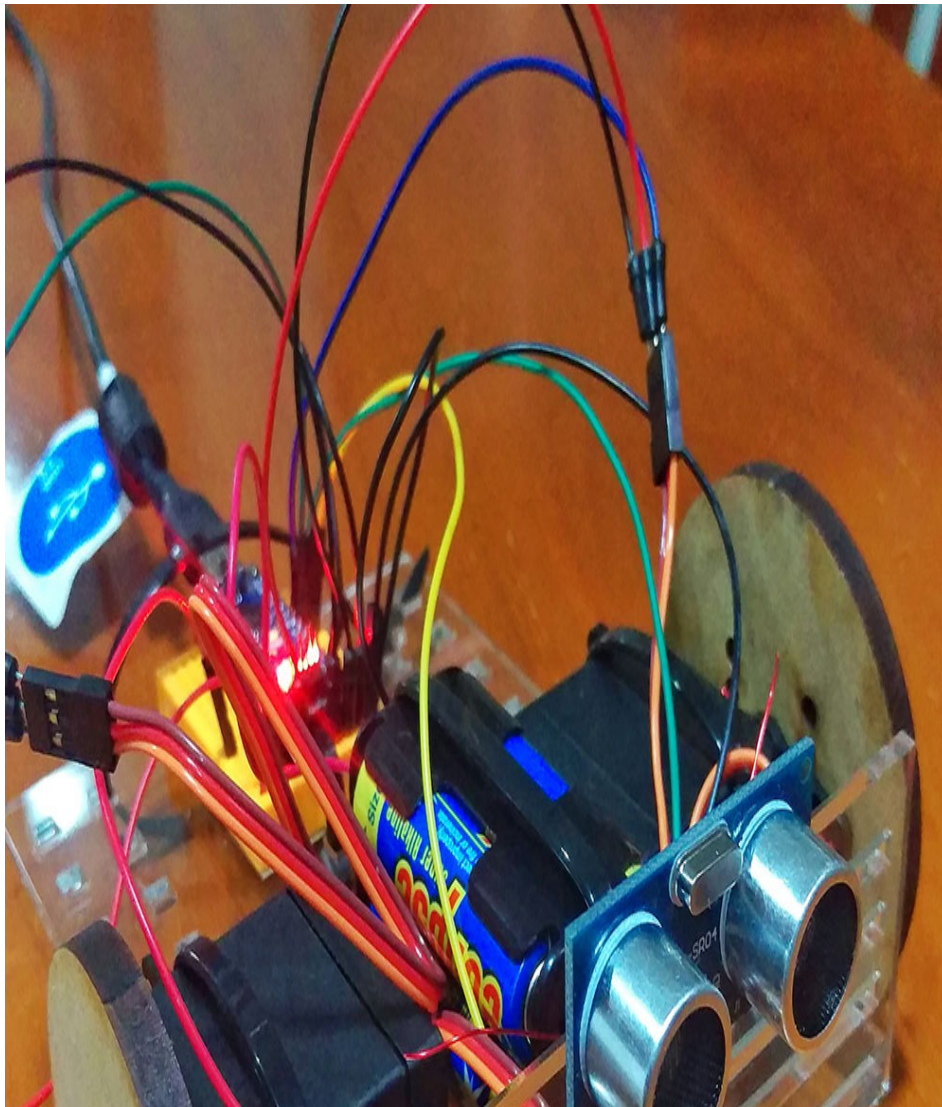
Resources for Learning NodeBots

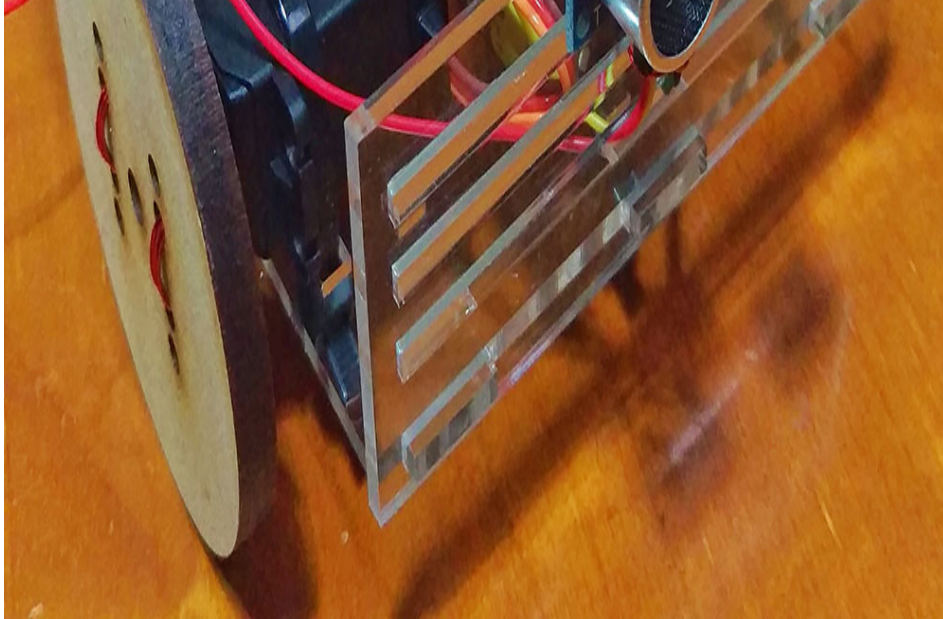
There are a few key spots where you can learn how to put together various NodeBot projects on the Web. Here are a few recommendations:

- [Controlling an Arduino with Node.js and Johnny-Five](#). This is a free SitePoint screencast I recorded a little while ago that introduces the basics of connecting up an Arduino to Node.js and using the framework to turn an LED light on and off.
- [Arduino Experimenter's Guide for NodeJS](#). An adaptation by Anna Gerber and other members of the NodeBots community from the SparkFun version of [.:oomlout.:'](#)s ARDX Guide. It shows how to do many of the examples from the kits mentioned above in Node instead of the simplified C++ code from Arduino.
- [The official Johnny-Five website](#). Not so long ago, the Johnny-Five framework had a whole new website released that has great documentation on how to use the framework on Arduino and other platforms too!
- [Make: JavaScript Robotics Book](#). A new book released by Rick Waldron and others in the NodeBot community that provides a range of JS projects using various devices. Great for those who've got the absolute basics down and want to explore some new projects!
- [NodeBots Official Site](#). Check this page out if you're looking for a local NodeBots meetup near you, or to read more about NodeBots in general.
- [NodeBots - The Rise of JS Robotics](#). A great post by Chris Williams on how NodeBots came to be. It's a good read for those interested.

THE SIMPLEBOT

Andrew Fisher, a fellow Australian NodeBot enthusiast, put together a rather simple project for people to build for their first NodeBot experience. It's called a "SimpleBot", and it lives up to its name. It's a NodeBot that you can typically build in a single day. If you're keen on getting an actual robot up and running, rather than just a basic set of sensors and lights going on and off, this is a great project choice to start with. It comes available to Australian attendees of NodeBots Day (see below) in one of the ticket types for this very reason! It's a bot with wheels and an ultrasonic sensor to detect if it's about to run into things. Here's what my own finished version looks like — which I prepared as a sample for NodeBots Day a few years ago:





A list of SimpleBot materials needed and some sample Node.js code is available at [the SimpleBot GitHub repo](#). Andrew also has a YouTube video showing [how to put the SimpleBot together](#).

Andrew also collaborated with the team at Freetronics to put together a SimpleBot Arduino shield that might also be useful to people who'd like to give it a go as a learning project without needing to solder anything: [SimpleBot Shield Kit](#).

Conclusion

That concludes a simple introduction into the world of NodeBots! If you're interested in getting involved, you've got all the info you should need to begin your NodeBot experience.

If you want to get more involved with NodeBots, keep an eye out for the annual International NodeBots Day. (It happens around July each year.) It's a day where all sorts of people get together at various events around the world to build JavaScript powered bots and have a great time.

If you build yourself a pretty neat NodeBot with any of the above resources, get in touch with me on Twitter ([@thatpatrickguy](#)), I'd love to check out your JavaScript powered robot!