Data-Driven Websites - Part 1: Setting Up the Project

Data-driven websites are everywhere online. From e-commerce websites to search websites to mega social media websites, data is the foundation of the dynamic, personalized experiences that users have come to expect of the Web.

You've learned all of the necessary skills—now it's time to bring it all together to create a data-driven website using Express!

Over the next three articles, you'll create a data-driven Reading List website that will allow you to view a list of books, add a book to the list, update a book in the list, and delete a book from the list. In this article, you'll set up the project. In the next article, you'll learn how to integrate Sequelize with an Express application. In the last article, you'll create the routes and views to perform CRUD (create, read, update, and delete) operations using Sequelize.

When you finish this article, you should be able to:

- Split the Express application and HTTP server into separate modules;
- Use the `morgan` npm package to log requests; and
- Add support for the Bootstrap front-end component library to your application's Pug layout template.

You'll also review the following:

- Setting up a new Express project;
- Stubbing out an Express application;
- Adding custom error handlers to an Express application; and
- Configuring environment variables.

Setting up the project

First things first, create a folder for your project. If you're using source control (and you are—right?), open a terminal, browse to your project folder, and initialize your Git repository by running the command `git init`.
You'll be using npm to install packages in just a bit, so be sure to add a `.gitignore` file to the root of your project. Then add the entry `node_modules/` to the `.gitignore` file so that the `node_modules` folder (where npm downloads packages to) won't be tracked by Git.

Pro Tip: *While configuring Git to not track the `node_modules` folder is important to do, it's not necessarily the only thing you want to configure Git not to track. For a more comprehensive `.gitignore` file for Node.js projects, you can use GitHub's .gitignore file for Node.js projects.*

Initializing npm and installing dependencies

Before you stub out the application, use npm to initialize your project and install the following dependencies:

```
npm init -y
npm install express@^4.0.0 pug@^2.0.0
```

Then install Nodemon as a development dependency:

```
npm install nodemon@^2.0.0 --save-dev
```

Stubbing out the application

Now it's time to stub out the application by writing the minimal amount of code to define the route for the default route (i.e. the "Home" page).

Start with adding a `routes` module by adding a file named `routes.js` to the root of your project containing the following code:

```
// ./routes.js

const express = require('express');

const router = express.Router();

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

module.exports = router;
```

The default route renders the `index` view which you'll create in just a bit.
Next, add the `app` module (`app.js`) to the root of your project containing the following code:

```
// ./app.js

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

const app = express();

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

app.use(routes);

// Define a port and start listening for connections.

const port = 8080;

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

To stub out the initial views for the application, add a folder named `views` to the root of the project. Then add two Pug templates to the `views` folder—`layout.pug` and `index.pug` containing the following code:

```
//- ./views/layout.pug

doctype html
html
  head
    title Reading List - #{title}
  body
    h1 Reading List
    div
      h2 #{title}
      block content
//- ./views/index.pug

extends layout.pug

block content
  p Hello from the Reading List app!
```

The `layout.pug` template provides the overall HTML for each page in the application while the `index.pug` template provides the HTML for the index or default page for the application.
All four of these files—`routes.js`, `app.js`, `layout.pug`, and `index.pug`—will evolve and change as you add features to the Reading List application.

Testing the initial application setup

It's time to test your initial application setup before you make any further changes.

Open the `package.json` file and replace the placeholder npm `test` script (that was generated by npm) with the following `start` script:

```
"scripts": {
  "start": "nodemon app.js"
}
```

From the terminal, run the command `npm start` to start your application, then browse to `http://localhost:8080/`. You should see the "Home" page displaying the message "Hello from the Reading List app!"

Now is a good time to commit your changes (if you haven't already)! *In general, making smaller commits more often where each commit contains a related set of changes is better than waiting until the end of the day to make one giant commit that contains all of the changes for the entire day.*

Splitting the application and server into separate modules

Now that you've created a simple, initial version of the application, it's time to start adding additional features and making general improvements to the overall design of the application.

Up until this point, you've created your Express application and started the HTTP server within the same module—the `app` module. A common practice is to separate the application and server into separate modules. Doing this has the following benefits:

- **Improved separation of concerns:** As much as possible, each module in your application should be responsible for doing one thing and only one thing. Separating out the server setup and startup into its own module improves the overall separation of concerns by allowing the `app` module to only be responsible for creating the Express application.
- **Improved testability:** Removing the startup of the HTTP server from the `app` module improves the testability of the Express application. While you won't be writing tests for your Express application in this project, establishing good coding practices will set you up to write tests for your application in the future.

Updating the `app` module

The last two statements in the `app` module (`app.js`) are responsible for defining a port and starting the server listening for HTTP connections:

```
// Define a port and start listening for connections.

const port = 8080;

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

Go ahead and remove that code. In its place, add this line of code to export the Express application from the module:

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

For reference, here's what the complete `app` module should look like at this point:

```
// ./app.js

const express = require('express');

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

const app = express();

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

app.use(routes);

module.exports = app;
```

Defining a new entry point for the application

As a reminder, the npm `start` script currently looks like this:

```
"scripts": {
  "start": "nodemon app.js"
}
```

Nodemon is being used to start the application and to restart the application when a change is made to any of the files in the project. The `app.js` file is provided as the entry point for the application—the module that's responsible for configuring and starting the application.

Now that the `app` module doesn't start the server listening for HTTP connections, it can no longer be used as the entry point for the application. To create a new entry point for the application, add a folder named `bin` to the root of the project. Then add a file named `www` (with no `.js` extension) containing the following code. Make sure `#!/usr/bin/env node` is on the first line of your file:

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

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

// Define a port and start listening for connections.

const port = 8080;

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

Then update the npm `start` script to pass the `www` file into Nodemon as the entry point:

```
"scripts": {
  "start": "nodemon ./bin/www"
}
```

To test your application's new entry point, run the command `npm start`, then browse to `http://localhost:8080/`. As before, you should see the "Home" page displaying the message "Hello from the Reading List app!"

Taking a closer look at the `./bin/www` file

The `bin` folder is a common Unix convention for naming a folder that contains executable scripts. Even though it's lacking the `.js` file extension, the `www` file is actually a JavaScript module that contains the code to start up the Express application.

You might have noticed that the first line of the `www` file isn't valid JavaScript:

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

This is an instance of a Unix shebang. The shebang has to be written on the first line of the `www` file. It tells the system what interpreter to pass the file to for execution. In this case, `node` is specified as the interpreter via the Unix `env` command.

The intention of the `./bin/www` file is for it to be an executable script—meaning that you could start the application by simply entering the file name in the terminal as a command:

```
bin/www
```

If you attempt to execute the script, you'll receive a "permission denied" error. Text files by default do not have the necessary permissions. You can use the `chmod` command in the root of your project to add the missing permissions:

```
chmod +x bin/www
```

With the proper permissions added, you can run the command `bin/www` to start your application.

Note: *The ability to run your application via a `bin` script is primarily useful for Node projects that are intended to be used as command line tools or utilities. The Sequelize CLI is an example of a Node.js command line tool that's executable via a `bin` folder script. For Express applications like the Reading List application, it's far more common to use an npm `start` script to run the application.*

Logging requests using Morgan

Currently, after the application starts up and displays the message "Listening on port 8080...", nothing else is written to the console to show activity. To assist with testing and debugging, you can install the `morgan` npm package, an HTTP request logger middleware for Node.js and Express:

```
npm install morgan
```

Aftering installing `morgan`, import it into the `app` module:

```
// ./app.js

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

Code organization tip: *Notice how the external modules are imported first and grouped together followed by the imported internal modules. While this isn't a hard requirement, it can help make it easier to read a module's dependencies.*

Then call the `app.use()` method to add `morgan` to the application request pipeline:

```
app.use(morgan('dev'));
```

The string literal "dev" is passed into `morgan` to configure the request logging format. The "dev" format is just one of the available predefined formats.

Now if you start your application and browse to `http://localhost:8080/`, you'll see the request logged to the console:

```
GET / 200 9.851 ms - 172
```

Here's a breakdown of the above output:

- `GET` - The request HTTP method
- `/` - The request path
- `9.851 ms` - The response time in milliseconds
- `172` - The `Content-Length` response header value that indicates the size of the response body in bytes

Adding custom error handlers

As you learned in a previous article, Express provides a default error handler, but for most applications you'll want to create a custom error handler to precisely control how errors are handled.

In the `app` module, start with adding a middleware function to catch unmatched requests and throw a "Page Not Found" error:

```
// ./app.js

// Code remove for brevity.

app.use(routes);

// Catch unhandled requests and forward to error handler.
app.use((req, res, next) => {
  const err = new Error('The requested page couldn\'t be found.');
  err.status = 404;
  next(err);
});

// TODO Add custom error handlers.
```

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

Next, add the following custom error handlers—an error handler to log errors, an error handler to handle "Page Not Found" errors, and a generic error handler:

```
// ./app.js

// Code remove for brevity.

app.use(routes);

// Catch unhandled requests and forward to error handler.
app.use((req, res, next) => {
  const err = new Error('The requested page couldn\'t be
found.');
  err.status = 404;
  next(err);
});

// Custom error handlers.

// Error handler to log errors.
app.use((err, req, res, next) => {
  if (process.env.NODE_ENV === 'production') {
    // TODO Log the error to the database.
  } else {
    console.error(err);
  }
  next(err);
});

// Error handler for 404 errors.
app.use((err, req, res, next) => {
  if (err.status === 404) {
    res.status(404);
    res.render('page-not-found', {
      title: 'Page Not Found',
    });
  } else {
    next(err);
  }
});

// Generic error handler.
app.use((err, req, res, next) => {
  res.status(err.status || 500);
  const isProduction = process.env.NODE_ENV === 'production';
  res.render('error', {
    title: 'Server Error',
    message: isProduction ? null : err.message,
    stack: isProduction ? null : err.stack,
  });
});

module.exports = app;
```

To complete your custom error handlers, add two views to the `views` folder—`error.pug` and `page-not-found.pug`:

```
//- error.pug

extends layout.pug

block content
 div
   p= message || 'An unexpected error occurred on the server.'
 if stack
   h3 Stack Trace
   pre= stack
//- page-not-found.pug

extends layout.pug

block content
 div
   p Sorry, we couldn't find the page that you requested.
```

Testing your custom error handlers

To test your custom error handlers, update the default route (`/`) in the `routes` module to temporarily throw an error:

```
router.get('/', (req, res) => {
  throw new Error('This is a test error!');
  res.render('index', { title: 'Home' });
});
```

Start your application and browse to `http://localhost:8080/` and you should see the "Server Error" page. Next, browse to an unknown path like `http://localhost:8080/asdf` and you should see the "Page Not Found" page.

You should also see the errors logged to the terminal:

```
Error: This is a test error!
    at [path to the project folder]/routes.js:7:9
    at Layer.handle [as handle_request] ([path to the project
folder]/node_modules/express/lib/router/layer.js:95:5)
```

```
    at next ([path to the project
folder]/node_modules/express/lib/router/route.js:137:13)
    at Route.dispatch ([path to the project
folder]/node_modules/express/lib/router/route.js:112:3)
    at Layer.handle [as handle_request] ([path to the project
folder]/node_modules/express/lib/router/layer.js:95:5)
    at [path to the project
folder]/node_modules/express/lib/router/index.js:281:22
    at Function.process_params ([path to the project
folder]/node_modules/express/lib/router/index.js:335:12)
    at next ([path to the project
folder]/node_modules/express/lib/router/index.js:275:10)
    at Function.handle ([path to the project
folder]/node_modules/express/lib/router/index.js:174:3)
    at router ([path to the project
folder]/node_modules/express/lib/router/index.js:47:12)
GET / 500 452.308 ms - 2070
Error: The requested page couldn't be found.
    at [path to the project folder]/app.js:16:15
    at Layer.handle [as handle_request] ([path to the project
folder]/node_modules/express/lib/router/layer.js:95:5)
    at trim_prefix ([path to the project
folder]/node_modules/express/lib/router/index.js:317:13)
    at [path to the project
folder]/node_modules/express/lib/router/index.js:284:7
    at Function.process_params ([path to the project
folder]/node_modules/express/lib/router/index.js:335:12)
    at next ([path to the project
folder]/node_modules/express/lib/router/index.js:275:10)
    at [path to the project
folder]/node_modules/express/lib/router/index.js:635:15
    at next ([path to the project
folder]/node_modules/express/lib/router/index.js:260:14)
    at Function.handle ([path to the project
folder]/node_modules/express/lib/router/index.js:174:3)
    at router ([path to the project
folder]/node_modules/express/lib/router/index.js:47:12) {
  status: 404
}
GET /asdf 404 15.648 ms - 223
```

Also notice that thanks to the request logging provided by the `morgan` middleware, you can see the `500` (Internal Server Error) and `404` (Not Found) HTTP response status codes returned by the server.

After confirming that your custom error handlers work as expected, be sure to remove the code that you temporarily added to your default route!

For a refresher on the custom error handlers, see the "Catching and Handling Errors in Express" article.

Configuring environment variables

Since the Reading List application will use a database for data persistence, your project needs to include a way to configure the database connection settings across environments. To do that, let's use environment variables to configure the application.

For a refresher on how to use environment variables within an Express application, see the "Acclimating to Environment Variables" article.

To start, install `per-env` as a project dependency the `dotenv` and `dotenv-cli` as development dependencies (i.e. dependencies that are only needed in development environments):

```
npm install per-env
npm install dotenv dotenv-cli --save-dev
```

As a reminder, the `per-env` package allows you to define npm scripts for each of your application's environments. The `dotenv` package is used to load environment variables from an `.env` file and the `dotenv-cli` package acts as an intermediary between npx and tools or utilities (like the Sequelize CLI) to load your environment variables from an `.env` file and run the command that you pass into it.

Adding the `.env` and `.env.example` files

Next, add two files to the root of your project—`.env` and `.env.example` with the following content:

```
PORT=8080
```

The `.env` file is where you define the environment variables to configure your application. At this point in the project, you just need to define the `PORT` environment variable. The `.env` file shouldn't be committed to source control as the environment variables it defines are specific to your development environment. Additionally, it might contain sensitive information.

To ensure that the `.env` file isn't committed to source control, add `.env` as an entry to your project's `.gitignore` file. If you're using GitHub's `.gitignore` file for Node.js projects, this has already been done for you.

Because the `.env` file isn't committed to source control, the `.env.example` file serves as documentation for your teammates so they can create their own `.env` files.

Adding the `config` module

Let's encapsulate all of the `process.env` property access into a single `config` module by importing all of the application's environment variables and exporting them to make them available to the rest of the application.

Add a folder named `config` to the root of the project. Then add a file named `index.js` to the `config` folder containing the following code:

```
module.exports = {
  environment: process.env.NODE_ENV || 'development',
  port: process.env.PORT || 8080,
};
```

Now you can update the `./bin/www` file to get the port from the `config` module:

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

```
const { port } = require('../config');

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

// Start listening for connections.

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

Updating the npm `start` script

To load the environment variables from the `.env` file in the local development environment while ignoring the `.env` file in the production environment, update the `package.json` file `scripts` section:

```
"scripts": {
  "start": "per-env",
  "start:development": "nodemon -r dotenv/config ./bin/www",
  "start:production": "node ./bin/www"
}
```

To review, if the `NODE_ENV` environment variable is set to `production`, then running the `start` script will result in the execution of the `start:production` script. If the `NODE_ENV` variable isn't defined (or set to `development`), then the `start:development` script will be executed by default.

At this point, running the command `npm start` should start your application just like it before.

Supporting debugging in Visual Studio Code

To use the debugger in Visual Studio Code, configure it to load your environment variables from your `.env` file. Open the `launch.json` file located in the `.vscode` folder and add the `envFile` property to your Node configuration:

```
{
  "version": "0.2.0",
  "configurations": [
    {
      "type": "node",
      "request": "launch",
      "name": "Launch Program",
      "runtimeExecutable": "npm",
      "runtimeArgs": [
        "run-script",
        "start"
      ],
      "skipFiles": [
        "${workspaceFolder}/node_modules/**/*.js",
      ],
      "restart": true,
      "console": "integratedTerminal",
      "program": "${workspaceFolder}/bin/www",
      "envFile": "${workspaceFolder}/.env"
    }
  ]
}
```

Note: *If you don't have* `.vscode/launch.json` *file in your project, see this page in the Visual Studio Code documentation for instructions on how to set up debugging.*

Setting up Bootstrap

One last bit of project set up work before installing and configuring Sequelize! Update the `views/layout.pug` view with the following Bootstrap template markup:

```
doctype html
html
  head
    meta(charset='utf-8')
    meta(name='viewport' content='width=device-width, initial-
scale=1, shrink-to-fit=no')
    link(rel='stylesheet'
href='https://stackpath.bootstrapcdn.com/bootstrap/4.4.1/css/b
ootstrap.min.css' integrity='sha384-
Vkoo8x4CGsO3+Hhxv8T/Q5PaXtkKtu6ug5TOeNV6gBiFeWPGFN9MuhOf23Q9If
jh' crossorigin='anonymous')
    title Reading List - #{title}
  body
    nav(class='navbar navbar-expand-lg navbar-dark bg-primary')
      a(class='navbar-brand' href='/') Reading List
    .container
      h2(class='py-4') #{title}
      block content
    script(src='https://code.jquery.com/jquery-
3.4.1.slim.min.js' integrity='sha384-
J6qa4849blE2+poT4WnyKhv5vZF5SrPo0iEjwBvKU7imGFAV0wwj1yYfoRSJoZ
+n' crossorigin='anonymous')
script(src='https://cdn.jsdelivr.net/npm/popper.js@1.16.0/dist
/umd/popper.min.js' integrity='sha384-
Q6E9RHvbIyZFJoft+2mJbHaEWldlvI9IOYy5n3zV9zzTtmI3UksdQRVvoxMfoo
Ao' crossorigin='anonymous')
script(src='https://stackpath.bootstrapcdn.com/bootstrap/4.4.1
/js/bootstrap.min.js' integrity='sha384-
wfSDF2E50Y2D1uUdj0O3uMBJnjuUD4Ih7YwaYd1iqfktj0Uod8GCExl3Og8ifw
B6' crossorigin='anonymous')
```

Adding support for the Bootstrap front-end component library will give the Reading List application a nice, polished look. The above markup was taken directly from the starter template published in the official Bootstrap documentation.

Adding Bootstrap won't change the look of the application much at this point. Later on, once you start adding forms to the application, it'll be easier to notice the benefits of using a library like Bootstrap.

What you learned

In this article, you learned how to:

- Split the Express application and HTTP server into separate modules;
- Use the `morgan` npm package to log requests; and
- Add support for the Bootstrap front-end component library to your application's Pug layout template.

You also reviewed the following:

- Setting up a new Express project;
- Stubbing out an Express application;
- Adding custom error handlers to an Express application; and
- Configuring environment variables.

Next up: integrating Sequelize with an Express application!