Web Applications Development

INITIAL DESIGN DOCUMENT

Joe Wemyss |Software Systems Development Year 4| 31/10/17

Contents

Project Overview

This project is to be a personal finance tracker application. The idea is that it will track the financial transactions of an individual, while being geospatially aware of the address of the individual, as well as where the transactions are made.

Currently, the application only consists of an authentication concept that will form the basis for the rest of the application.

Version Control

Version control for this application is handled by Git. The code is stored on GitHub, in this repository: https://github.com/JavaTheNutt/web_app_dev_project_backend. The application at the time of this submission is stored in the first_submission branch, located here: https://github.com/JavaTheNutt/web_app_dev_project_backend/tree/first_submission.

Building the Application

This application was built using NodeJS 8.6.0, which supports the use some ES2017(EcmaScript Standard 8) features. In order to ensure that this project can be built and run by anyone, it uses Babel(https://babeljs.io/) and the env preset (https://github.com/babel/babel-preset-env) for transpilation based on the current environment. For full details on building and running the application, refer to the project README(https://github.com/JavaTheNutt/web_app_dev_project_backend/blob/first_submission/README.md).

Architecture

This application is designed using a more modular structure than most standard Express application examples that I could find online. The base of the application is split into components, middleware, models and utilities.

COMPONENTS

The component directory may be poorly named, as I based this approach on the modularity section of the officially endorsed style guide for developing Angular applications (https://github.com/johnpapa/angular-styleguide/blob/master/a1/README.md#modularity), where the name components makes more sense. Resources would have been a better name, but I did not wish to cause

confusion with the more commonly used definition of resources in Express, which usually refers to static assets. This name will most likely be restructured in a later iteration.

Each subdirectory in this directory references a REST resource, which may be comprised of multiple models and services, as well as a set of routes to access these resources. The routes are "barrel exported" upwards to feed back into the main router file, which means that I need only add a new route to the local component router and it will be automatically available throughout the application without modifying the main router file, or the application entry point.

MIDDLEWARE

The middleware directory contains all global application middleware to be utilised. Currently the only global middleware is for Authentication. This follows a similar structure to the components directory, in that it is made up of models and services. The main difference is that subdirectories in the Middleware directory are not scanned for routes since there will be no routes attached, and there will also be a file to expose the actual functions that will be attached as middleware.

MODELS

This directory holds the mongoose models (and associated services) that will be used inside the components. Currently the only entry in this directory is the Address model, which is used for users, but may also be used for suppliers later on in the application.

UTILITIES

The Util directory holds application utilities that will be used throughout the application. Currently there are only two, error utils and the logger.

Error Utils

I created this module to deal with difficulties that I was having using JavaScripts new asynchronous mechanism, async/await. The issue that I was having is that since this is such a new technology, there is no standard in place for error handling, like the *reject* call when using Promises, or the error first method when using callbacks. The only way of handling errors is using *try/catch* blocks. I really didn't want to include a *try/catch* in every function up the call stack, and I also didn't want to throw exceptions every time I needed to stop the execution of a function early. This led to me simply returning a "falsey" value if an error occurred, which I then checked for in the calling function.

This was a terrible way of handling errors for several reasons. Firstly, it was very brittle, for example if a value such as *null* or *false* was an acceptable result of a function, I had to add checks the whole way up the stack to ensure that it wasn't treated as an error. Secondly, by the time the result reached the controller, which is attached to a specific route, I had

no idea where the error was thrown, and no error message to return to the user. This led to me just returning a status code of 500 and a generic error message for each request.

My solution was to create a utility module that could emulate the old-fashioned error first callback method that has always been inherent in NodeJS. It is simply a collection of "pure" functions that can be used to create and modify error objects. For example, if a function tries to write to the database, and an error is thrown, I could create an object which contains a custom message, as well as the error that was thrown. This object would be in the format of:

```
{
    error: {
        message: 'error occurred
        while saving user record',
        err: 'Error: Duplicate Key email'
    }
}
```

This allowed me to do checks along the lines of:

```
const result = await someAsyncOp();
if (result.error) return result;
```

I also added the capability to create an object that can be sent to the client by simply concatenating the custom message to the message that is attached to the actual error object.

This solution is not quite complete yet, as in certain cases I have to perform string comparisons on the message to determine the status code that should be sent to the user. I will have to make an adjustment to the error object so that it can also encapsulate an optional status code. Once that is done, I am considering extracting this to a separate module and publishing to NPM.

Logger

For logging in the application, I am using Winston (https://www.npmjs.com/package/winston), which is the Logger that I use most often in Express applications. I configured this Logger mainly by reading the docs, but the idea of exporting the configuration as a function, instead of an object, so that I can pass a module name to be prefixed to each log statement was found in this StackOverflow answer: https://stackoverflow.com/a/42966914/4108556.

This logger also uses the express-winston package (https://github.com/bithavoc/express-winston) to log every request and uncaught error.

The loggers level can also be configured using command-line arguments, or defaults based on the current value of *process.env.NODE_ENV*. i.e. in development the logger defaults to verbose and in production it defaults to error. This is handled in the config file

(https://github.com/JavaTheNutt/web_app_dev_project_backend/blob/first_submission/config/config.js).

Other Technologies Used

MODULE ALIAS

This module (https://www.npmjs.com/package/module-alias) can be used to apply aliases to certain paths in the application. For instance, I can add an alias "@user" to my *package.json* that points at "./api/components/User" so I can just use "@user/models/User" from anywhere in my application, instead of the full relative/absolute path. This came in extremely useful when I added Babel to the project, as I only had to change my aliases in the *package.json* and all my references pointed to the new, transpiled code rather than the original source code.

MONGOOSE

I used mongoose (http://mongoosejs.com/) as my Object Document Model (ODM) to abstract the details of dealing with MongoDB into a more Object-Oriented approach. I'm almost certain that everything that I did with mongoose is straight from the documentation.

I have one slight issue with mongoose that I can't find the solution to online. That is preventing certain models from being saved. In my case, currently it is the Address model. This model will only ever be saved as an embedded document within a parent collection, and never in its own collection. To solve this I am thinking of replacing the *save()* method on the prototype of the Address model with an empty function, but I can find no documentation online to suggest that this is the correct method. I will attempt this in the next iteration of the project.

EMAIL-VALIDATOR

I used this package(https://www.npmjs.com/package/email-validator), to validate email addresses, rather than attempting to write a custom regex. Everything I needed to use this package is included in the documentation at the above link.

FIREBASE

I used the Firebase Admin SDK (https://firebase.google.com/docs/admin/setup) to handle authentication, since I plan on using the Firebase Client API(https://firebase.google.com/docs/web/setup) on the client-side. Again, everything that I used from Firebase comes straight from the documentation at the above link.

One thing that might be considered a poor choice about how I utilized Firebase in this project was that I had to include the client-side API as a development dependency so that I could retrieve a token for integration testing. However, since it is included as a development dependency, it won't affect bundle size when the application is deployed.

EXPRESS

I used express(https://expressjs.com/) as my NodeJS framework. I didn't use any generators to create the application as I couldn't find an appropriate Yeoman template that set up the application the way that I wanted. I also find that Express is very quick and easy to configure manually. For the most part, everything I used to configure express is standard, such as the cors package(https://github.com/expressjs/cors) for fixing Cross-Origin Resource Sharing issues(currently just adding a wildcard to the Access-Control-Allow-Origin header until I know the domain of my client app), and body parser (https://github.com/expressjs/body-parser), to extract the contents of the Request Body, but there are a couple of exceptions.

The most noticeable of these exceptions was that I decided not to use the standard express logger(Morgan), as I prefer Winston (see above section on Logging for Winston configuration).

I also implemented a custom solution for handing routing. This solution can be found here: https://github.com/JavaTheNutt/web_app_dev_project_backend/blob/first_submission/api/router.js, and is a heavily modified version of this StackOverflow answer: https://stackoverflow.com/a/11923719/4108556, that has been adapted for my architecture. I had to heavily reference the Node FileSystem (fs) documentation(https://nodejs.org/api/fs.html) to create this solution. I do have an issue with this implementation, and that is that I must use the synchronous implementation of all the file system methods, since otherwise if a request came in when the server was just after starting, the routes would not exist yet. This is obviously slower, especically since there is a synchronous operation to read the contents of the specified directory and another to check the stats of each entry in that directory, but since CommonJS require calls are cached (https://medium.freecodecamp.org/requiring-modules-in-node-js-everything-you-need-to-know-e7fbd119be8), these blocking I/O operations should only happen when the application is initialised.

TESTING

The application is fully tested using Mocha (https://mochajs.org/) as a test framework/runner, Chai (http://chaijs.com/) as an assertion library, Sinon (http://sinonjs.org/) for stubs/mocks, Supertest(https://github.com/visionmedia/supertest) for API testing and NYC/Istanbul (https://github.com/istanbuljs/nyc) for code coverage.

I also used some plugins such as Chai-Things(https://github.com/chaijs/chai-things) for performing assertions on arrays and Sinon-Chai(https://github.com/domenic/sinon-chai) to add Sinon assertions (such as calledOnce, calledWith, calledBefore etc.) to Chai.

I was able to use a handy cheat-sheet(https://gist.github.com/yoavniran/1e3b0162e15450554296e) for all of the Chai and Sinon-Chai assertions that I needed. Aside from that, for the most part, each individual set of documentation was sufficient to learn how to use these tools, except in the case of Sinon, where I had to consult several blogs to piece together the parts to get a good idea of how to proceed with stubbing the parts of my application. The main blogs that I consulted were:

- https://semaphoreci.com/community/tutorials/best-practices-for-spies-stubs-and-mocks-in-sinon-js
- https://www.sitepoint.com/sinon-tutorial-javascript-testing-mocks-spies-stubs/

DOCUMENTATION

There is a full set of JSDocs included in the repository. These can be found here: https://github.com/JavaTheNutt/web_app_dev_project_backend/tree/first_submission/docs/jsdocs. They can also be generated using the command: *npm run docs*.

The docs are generated using the JSDoc package (https://www.npmjs.com/package/jsdoc)