Creating a Single Page MEAN Stack Web Application

Darren Fitzpatrick

B.Sc (Hons) in Software Development

April 16th, 2018

Final Year Project

Advised by: John Healy

Department of Computer Science and Applied Physics

Galway-Mayo Institute of Technology (GMIT)

Abstract

This project was developed alone as my final and fourth year assignment to achieve a Level 8 B.Sc. (Hons) in Software Development in GMIT. The purpose of my project was to create and host a full stack web development application by using the MEAN stack technologies (MongoDB, Express.js, Angular 2, Node.js)[1]. This is a single page web application, consisting of a discrete, informal, diary style text entry that allows the user to sign in with a unique username and create online blog(s). The blogs can be interacted with by other users, commenting, liking or disliking the post that uses bcrypt and JWT tokens for a secure authentication. The application utilises a 3-tier structure, a front-end, back-end and middle tier. The front end is the angular 2 UI, the middle tier consists of the node server containing a RESTful API [9]. This node server acts as the bridge between the database back-end and the app, which is hosted online using Heroku. The database is connected to my mlab deployment, which is connected to Heroku. This ties the project together with the front-end connected to the hosted node in the middle tier so it can access and send data to and from my database. The RESTful API sends JSON data from client to database and back through the server. This is how my blog data is updated when deleted, created, liked, etc. This whole experience has been extremely informative and has exposed me to a completely new set of tools, software practices, and engineering principles. It has ultimately left me with a deeper understanding of what it means to be a software developer and an incredible appreciate for the amount of time and work that is involved in designing and implementing a full-scale software application.

Authors This project was developed by Darren Fitzpatrick from Galway-Mayo Institute of Technology in order to achieve a Level 8 B.Sc. (Hons) in Software Development.

Contents

1 Introduction

This is my dissertation for my final year project which I developed to achieve a B.Sc. (Hons.) in Software Development. I undertook this project solo, with the idea of creating a interactive blog page that allows users sign-in, create a profile, make posts and share information easily online.

1.1 What is it about?

The premise of this project was to develop something that utilized a collection of technologies that integrated well together and in turn create a full stack web site [6]. With this in mind, I decided the best approach would be to implement a 3-Tier architecture. It is best suited for numerous reasons, as it adds security, performance and reliability. Also, it is widely used as an industry standard these days and would surely meet the difficulty expectations of a Level 8 degree. During my initial exploration, I researched numerous applicable technologies to find the best option for my needs and after considerable research, I decide to use a MEAN stack approach. The MEAN stack incorporates MongoDB, Express.js, Angular 2, and Node.js, and all the components of this stack support programs written in JavaScript. MEAN applications can be written in one language for both server-side and client-side execution environments. This will all subsequently be hosted using Heroku [15], a platform as a service that enables developers to build, run, and operate applications entirely in the cloud. These technologies are the basis of what I used to create my blog page web application.

1.2 Scope

This project was developed single-handedly. So bearing this in mind, the scope couldn't be too large that I would be unable to complete it all, or too small that it wouldn't have been a worthwhile endeavour. The project has a three-tier architecture with a client, server, and database all communicating together to form my web application. The 3 tiers are as follows:

1.2.1 Frontend/ Angular 2 Client

The front-end is everything involving what the user sees and interacts with, including design. For this I used Angular 2 which is an open-source, front-end JavaScript framework that really shines when building those dynamic, single page applications [11]. It works by making calls to a server-side program(NodeJS), running on an application server and is written in any server-side language that connects to the database and performs the operation, then returns a result to the client in JavaScript. In Angular 2, I did that with the use of RESTful API's.

1.2.2 Server side/Node Server

This is my middle tier where Node.js and Express will be used [3, 5]. While Node.js is the engine that gets everything running, Express is the framework which I use to write Node.js applications more effectively and efficiently. I used it to write my Web RESTful API, as well as for communicating with MongoDB server. For the second part, communication with MongoDB server, library specially crafted for this purpose will be used – Mongoose [20]. In essence, the server itself is run directly from Node rather than being embedded in another server like Apache.

1.2.3 Backend/Database

This side of the application holds all of the sites information. The data includes usernames, passwords, blogs, comments, etc. It is all stored in a NoSQL database program called MongoDB that uses JSON-like documents with schemas. Essentially, the backend of my application is responsible for things like calculations, business logic, database interactions, and performance. This database also uses a mLabs connection which together with Heroku deployment hosts my node server on the web, with it's own Heroku url.

1.3 Objectives

The primary objective in developing this application was to learn and implement a new set of skills. I ultimately wanted to improve my knowledge of an area that I wasn't entirely familiar with. Using MEAN stack enabled me to understand how a web site fully functions, as I got the chance to develop each layer of the architecture. The following are the milestones I set out at the beginning for how I wanted my project to function:

- Understand the principles of creating an effective full stack web page, including an in-depth consideration of information architecture.
- To create a full stack web application that would require a user to be able to login to the website by creating their own unique, secure account with a username, email and password.
- Registration and login would incorporate authentication to check that the formatting of the email and passwords are correct, and the the username or emails are not already taken by a previous user.
- Provide guarded routes, so blog users can edit only their own post and not others.
- Each user then could create their own blog post on a blog page. It could be interacted with by the users, allowing them to comment, like, or dislike posts.
- Deploying the application so it can be accessed and used online.

2 Methodology

Initially a large part of my project involved a careful decision process, which was then followed by a considerable amount of research. Eventually, I decided on a web application, thinking it would be a suitable option given the time frame I had and taking into account that I would be working solo. To fully understand my projects limitations, I dedicated the first month to investigating some possible technologies I might implement. I knew I wanted to use Angular and then slowly incorporated the rest of the technologies with that pre-existing knowledge. I researched many options and had listed out some advantages and disadvantages of each in order to optimize my app, thus allowing it to fulfill its potential.

2.1 Project planning and meetings

All planning was done for this project in my own personal weekly meetings. Being a solo developer on this project, I sat down and drew up ideas, theorized some possible concepts and explored how I would go about developing that project. This all required a lot of brainstorming and writing down potential ideas. I knew I was leaning towards a blog development application but wasn't fully confident yet on which technologies to use.

After a few weeks and with a clearer idea of what I wanted to build, I set out researching the best suited technologies. I found there were many possibilities when it came to web development and choosing the relevant software was not going to be a simple or straightforward task. I was fairly unfamiliar with all of the architecture involving web development, so a lot of time was required to fully research the relevant software and make an informed decision before I proceeded.

I looked into LAMP (Linux, Apache, MySQL, and PHP, Perl, or Python.), Ruby on Rails, and compared it with MEAN stack[7, 12]. I played around with each and eventually picked the one I found most applicable towards my eventual goal.

After picking the technologies I began to slowly work on my project and set myself achievable goals along the way. I had meetings with my supervisor to report on my research findings and ensure my project was up to standard and that I was on schedule.

2.2 Agile

Despite being a solo developer, I still chose an Agile approach to this project[8]. There's no reason a lone developer implementing Agile can not continue to do (and benefit from) some or all of the following:

- Keep track of how long it takes you to do tasks and how many tasks get completed in a sprint (burndown rate/velocity); this is key to accurately estimating how long future tasks will take and future sprint planning.
- Hold daily "standups" before starting work each day (i.e., review what was completed the day before, what issues there were and what needs to be done today)
- Work in short code sprints and have frequent, incremental product releases. Or in my case, just push to GitHub regularly.

Agile also meant I could make more modifications as I progressed with the project. When setting out I knew I might regularly have to adapt to changing circumstances, because my project idea was evolving and not yet fully decided upon. Flexibility was important, so therefore, Agile was a suitable approach for those reasons.

2.3 GitHub development process

I used GitHub throughout my project to help communicate any outstanding issues[14]. GitHub has a pretty simple issue management system for bug tracking, but it is flexible enough to be a pretty powerful tool for managing my entire project. I utilized issues which can reference each-other; labels for attaching meta data to my issues; methods for attaching code to your issues; and even milestones for grouping and focusing your issues within time blocks.

Another beneficial aspect I availed of was the GitHub code review feature. This benefited me greatly, enabling me to work confidently and productively. It allowed me

to compare my source code side by side, highlighting any new, edited, or deleted parts. This way it was easy to then conclude on the changes I made. This means, I can have a version history of my code so that previous versions are not lost with every iteration.

GitHub also allowed me to benefit from its useful branching feature. This characteristic enabled me to create an environment where I could try out new ideas and meant I didn't have to touch the master branch containing all my master source code and potentially risk its integrity. I used this quite a number of times with my project and ended up merging most of these ideas. I found it to be an efficient and effective way of working which ultimately helped to boost my productivity and was the main reason for utilizing GitHub as much as I did. It provided a clever work environment and I gained from it greatly.

2.4 Testing

Everyone agrees that writing tests are important, but not everyone does it. As you introduce new code, tests ensure that your API is working as intended. During the development stage it is important to write and run tests in Postman for each request. As my codebase grew, it was vital to ensure that I was not breaking anything that was previously working. The higher your test coverage, the more flexible and bug-resistant your code will be, and therefore less time I will be spend debugging hot fixes in production.

Postman has been created to support all aspects of API development, and it was what I chose to test my project with [17]. This tool makes it possible to quite easily and conveniently test sent requests and responses, without using the GUI of the tested application. The interface gives you a request editor where I can provide the different parts of a request like the URL, the HTTP verb, and any params I want to pass along. I then have an option to send the request or save it for later. After sending the request, I get a lot of information back, including the status code, headers, cookies, how long the request took, the size of the files returned, as well as the raw HTML, JSON, XML, etc.

3 Technology Review

In this section, I wanted to explore some of the different technologies I decided on incorporating into my project, thus allowing for an informed insight into the rational behind my choices. All the technologies I implemented were chosen with compatibility in mind. I wanted an architecture that wouldn't impede my productivity and cause a lot of problems along the way and this is the reasoning behind opting for the MEAN stack amalgamation. This quartet looked promising and I had confidence that they would work amazingly well together to seamlessly produce the end result with high performance and functionality. I felt it was more relevant than the long-established LAMP (Linux, Apache, MySQL, PHP) environment, that I had previously researched.

3.1 Why I chose a Stack

One of the first and most important decisions that I had to make when starting this project was choosing the appropriate language and toolset to use. It is an incredibly important decision as choosing the wrong language for the job can result in many wasted hours trying to make something work when it could have been done more easily in another language. When I started out, I was initially interested in writing the application in pyton with Django, it was a language I was unfamiliar with but had always wanted to have the opportunity to explore it in more detail. Also, I had heard several positive things about it from colleagues. This is not the ideal way to choose a programming language for a project, but at the time python with Django seemed like an exciting, and viable option. As I stated earlier, compatibility is an integral aspect of this assignment and is why I then leaned towards implementing a well tried and tested web stack instead.

My application architecture was going to consist of a backend RESTful API (Representational State Transfer Application Programming Interface). I researched having the project built in Node.js with a front-end user interface built in Angular, and a MongoDB database. This is more commonly known as the MEAN stack (MongoDB

Express.js AngularJS Node.js), a term first coined by MongoDB developer Valeri Karpov [4]. I liked this infrastructure because these tools are very useful on their own but can become even more powerful when used together.

One of the main benefits of using this software stack is that engineers only have to work with one programming language from the database up [4]. Querying the database is done in JavaScript and all of the data in the database is stored in JavaScript Object Notation (JSON), the backend API is in JavaScript and serves data to consumers in the form of JSON (without needing to convert it), and the entire frontend runs on JavaScript and consumes the JSON data. As Karpov puts it, "By coding with JavaScript throughout, we are able to realize performance gains in both the software itself and in the productivity of our developers. [4]" JavaScript is the de facto language of the web, so it makes sense to use it throughout a web application.

Another sometimes overlooked benefit of the MEAN stack is the large and vibrant development community for each of the individual tools. Learning a stack is a lot easier when there is a large number of tutorials on how to get started and some of the common issues that developers have encountered in the past are highlighted. Each of the tools are open source with a friendly open source community to boot, so getting support is a simple GitHub issue or Stack Overflow post away. Django is also open source but the development community is a lot smaller, so finding solutions to some of the early problems that I experienced with 9 the framework while going through the official tutorials was a lot more difficult and made the idea of using MEAN stack much more desirable.

3.2 MEAN-Stack

Mean Stack is a combination of four widely popular and highly efficient Javascript libraries, namely MongoDB, Express.js, Angular 2 and Node.js. (Shortened down to M for Mongo, E for Express, A for Angular and N for Node). Since all the components of MEAN Stacks are products of a Javascript library, it shouldn't surprise anyone that they integrate well. Mongo DB can be used to store documents on the JSON format – and these JSON queries are handled with utmost ease by Express JS and Node.js on the server side. Angular JS on the frontend is then fed these JSON documents, with slim chance of any errors or obstructions occurring. With the same

language on both the client side and the server side, the integration between these two environments is seamless and very subtle. Having all the components of MEAN stack written in JavaScript, MEAN application can be written in one language for both server-side and client-side executions environments. In order to completely understand the reasons why MEAN stack is so widely used, I will briefly discuss each of its components below.

3.2.1 MongoDB

I decided to use MongoDB because all of the components in my networked application should be designed to be fast and fluid. Using this database meant my data wouldn't be bottle-necked. MongoDB is able to meet new data challenges that are difficult to accomplish well with a relational database. MongoDB allowed my project to:

1. **Store large volumes of data that often have little to no structure.** Relational databases store structured data like a phonebook. But for growing, unstructured data—for example, a customer's preferences, location, past purchases, and likes—a NoSQL database sets no limits, and allows you to add different types of data as your needs change. Because MongoDB is flexible and document-based, you can store these JSON-like binary data points in one place without having to define what "types" of data those are in advance. This is more suitable for my project.

2. **Make the most of cloud computing and storage.** Cloud-based storage is an excellent cost-saving solution but requires data to be easily spread across multiple servers to scale up. MongoDB can load a high volume of data and give you lots of flexibility and availability in a cloud-based environment, with built-in sharing solutions that make it easy to partition and spread out data across multiple servers. MongoDB is a good choice in case our website will need to store a large amount of data in the future. It allows me to ustilise a fully managed cloud database service mLab to host my MongoDB database, which can be incorporated relatively easily[16].

3. **Develop and release quickly.** Due to the fact I was developing within two-week Agile sprints, cranking out quick iterations, or needing to make frequent updates to the data structure without a lot of downtime between versions, modifying a relational database would slow me down. Therefore, MongoDB was an adequate choice, allowing for dynamic schemas, where you can try new

things whilst being fast. Out data didn't need to be prepped ahead of time, and I could incorporate new changes, quickly and efficiently.

4. **Scale database architecture efficiently and effectively.** With MongoDB, it's easy to spread data out across commodity hardware on-site or in the cloud without needing additional software. Despite this, if you did needed to implement a service like I did with mLabs, it was reasonably simple.

How MongoDB works in my application.

In my application, I used MongoDB to store information like a new blog post. I utilize Mongoose, which provides a straight-forward, schema-based solution to model my application data. It includes built-in type casting, validation, query building, business logic hooks. Here is how I set up the database:

```
mongoose.connect(config.uri, {useMongoClient: true}, (err) => {
    if(err){
        console.log('Couldn't connected to database', err);
    }else{
        console.log('Connected to database: ' + config.db);
});
```

This is an example of how I configured the database to connect to mLabs cloud [16]. Before I used a local host connection - uri:mongodb://localhost:27017/web-db

```
Module.exports={
uri: 'mongodb:[username]:[password]@ds239309.mlab.com:39309/my-web-db',
db: 'my-web-db'
}
```

3.2.2 Express

Express is the web application framework that runs my back-end application (JavaScript) code. It runs as a module within the Node.js environment. Express can handle the routing of requests to the right parts of your application (or to different apps running in the same environment). You can run the app's full business logic within Express

and even generate the final HTML to be rendered by the user's browser. At the other extreme, Express can be used to simply provide a REST API – giving the front-end app access to the resources it needs e.g., the database. For my blog application, we will use Express to perform two functions:

1. Express acts as a light-weight web application framework to help organize my web application into an component based MVC architecture on the server side.

2. Provide a REST API that the front-end can access using HTTP network calls, in order to access the database.

3.2.3 Angular 2

Angular 2 has been developed by the same team who developed the famous angularJS, it brought the hard learned lessons of AngularJS into consideration this time round and developed a much cleaner more useable software. I find Angular 2 is more of an ALL IN ONE framework and it gives me just about everything to avoid myself getting trapped in a twisted nest of different JS frameworks, when I just want to create a simple single page website. Lets have look at Angular 2's improved features:

1. **TypeScript:** TypeScript is developed by Microsoft and is supported by Angular 2. It introduces static types which makes things like IDEs and builders/compilers more effective. In some scenarios it also makes writing testing your code easier.

2. **Modular:** Angular apps are quite modular i.e. organizing your code into independent chunks or buckets such that each of such bucket offers similar kind of functionality. This makes testing, upgradation and maintenance of the app a lot more convenient.

3. **Improved *ngIf and *ngFor:** if/else style syntax has been introduced where you can assign local variables such as when unrolling an observable. I used this feature heavily for validation.

4. **Flat ES Modules:** Modules are shipped as flattened version that helps in tree-shaking and reduce the size of generated bundles. It also speeds up the building process, transpilation and loading in the browser in certain scenarios.

5. **Angular CLI:** Angular command line interface (idea derived from ember) is pretty cool and makes your app development process easier. It provides you with all the boilerplates you want. So, adding basic boilerplate for new components, directives, services, etc. is not a big deal.

6. **Native App Development support:** With Ionic integration, you go can ahead and build native mobile apps without even requiring any more extra knowledge other than angular. I didn't use ionic, but it's nice to have the option should I further develop this project.

3.2.4 Node.js

Node.js is a JavaScript runtime environment that is used to runs your back-end application (via Express). It is worth noting that Node.js is based on Google's V8 JavaScript engine which is used in the Chrome, which is one of the most used browsers worldwide. It is very practical as it includes a number of modules that provide features that are essential for implementing web applications – including networking protocols such as HTTP. Node uses npm, which makes it easy to install packages like bcrypt with relative ease using the command line interface. The website is simple, clean, and provides helpful stats. Also, all packages link to the github repo, which greatly improves the experience.

Node.js is an asynchronous, event-driven engine where the application makes a request and then continues working on other useful tasks rather than stalling while it waits for a response. On completion of the requested task, the application is informed of the results via a callback. This enables large numbers of operations to be performed in parallel which is essential when scaling applications. MongoDB was also designed to be used asynchronously and so it works well with Node.js applications.

3.3 Restful API

A RESTful API is an application program interface (API) that uses HTTP requests to GET, PUT, POST and DELETE data. A RESTful API, which is also referred to as a RESTful web service is based on representational state transfer (REST) technology. An API is essentially code that allows two software programs to communicates with each other. In my case it was use to deal with the blogs data, to alter, retrieve, edit and remove blogs in the database. The authentication API was used to create and

deal with the users.

3.3.1 How Restful API Works with relation to my application

RESTful API best practices come down to four essential operations:

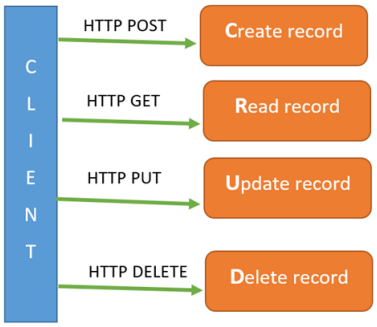
- Receiving data in a convenient format.
- Creating new data
- Updating data
- Deleting data.

An RESTful API is used in my project for a dealing with blogs and user authentication/authorization. Authentication means correctly ensuring that you are who you say you are and authorization is asking if you have access to a certain resource. When working with REST APIs you must remember to consider security from the start. RESTful API often uses the GET, POST, PUT and DELETE function to achieve this. Not all of these are valid choices for every single resource collection, user, or action. I made sure that the incoming HTTP method is valid for the session token/API key and associated resource collection, action, and record.

For example, if you take my RESTful API for my blogs, it's not okay to allow any users to DELETE or EDIT a post entry, but it's fine for them to GET all blogs entries to view them. On the other hand, for the user that created that blog, all of these are valid uses.

- GET - getting.
- POST - creation.
- PUT - update (modification).
- DELETE - removal.

All these methods (operations) are generally called CRUD. They manage data, "create, read, update and delete" it. The fact that REST contains a single common interface for requests and databases is a great advantage.



3.4 Heroku

After testing my web application locally for any errors, it was time to deploy it. After extensive research and experimenting, I concluded that Heroku would be an appropriate choice[15]. It was well suited to the projects infrastructure. Heroku makes it easy to deploy and scale Node.js applications in the cloud. It runs any recent version of Node.js and deploys apps in seconds utilizing dependency caching. Easily install third-party a add-ons like our MongoDB.

The Heroku deployment is connected to mLabs, a fully managed cloud service that hosts MongoDB databases and connects to our Heroku service[15]. This was a challenging aspect of the project to get working, as I will refer to 4.2.3. It required me to point the node.js server to the build of the application and add the URL of my Heroku instance to replace the local host. This required a little bit of research and persistence to eventually solve what should have been a relatively easy task.

3.5 Github

Using Github for my project was an easy decision. I was very confident and familiar with it, having used it throughout my past 4 years in college. It is an easy and safe way to store this project and has so many perks with very few drawbacks.[14]

Have your code reviewed by the community. My project is a skeleton - it does what you want it to do, but I'm not always sure how the wider population will implement it, or if it even works for everyone. Fortunately for me, if I post my project on GitHub, the wider community of programmers and hobbyists can download it and as a result, evaluate the work[14]. This means they could give me a heads-up on possible issues such as conflicts or unforeseen dependency issues, if I needed it.

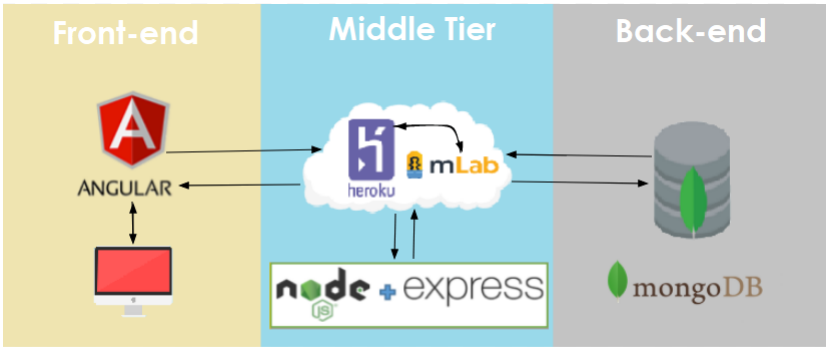
GitHub is a repository. It's important to note– GitHub is a repository [14]. What this means is that it allows your work to be viewed by the greater public or peers. Moreover, GitHub is great way of demonstrating your coding skill and easily displaying your work online for future prospective employers to inspect. I find it is beneficial having my project posted on GitHub where it can be accessed by other classmates, or lectures at any time. Allowing them to download it and assist should any problems arise.

Collaborate and track changes in your code across versions. Much like using Microsoft Word or Google Drive, you can have a version history of your code so that previous versions are not lost with every iteration. GitHub also tracks changes in a changelog, so you can have an exact idea of what is changed each time. (This is especially helpful for looking back in time.)

A ton of integration options. An import part of my project was having Heroku integrate with GitHub, making it easy to deploy code living on GitHub to apps running on Heroku. When GitHub integration is configured for a Heroku app, Heroku can automatically build and release (if the build is successful) pushes to the specified GitHub repo[14].

4 System Design

This section I want to explore the 3-tier architecture I implemented and demonstrate how the MEAN stack is all connected. Looking at the front-end, it consists of Angular 2 which is responsible for the overall user interface of the web page. Then the middle tier that consists of Node and express, which serves HTTP request for the web pages. This layer deals with routing, which determines how an application responds to our clients request to a particular endpoint that is a URI with a specific HTTP request method (GET, POST, and so on). It essentially deals with our REST API. The middle tier also deals with our Heroku deployment which serves our node server online. Our backend is our data tier which contains the MongoDB database where all our data for blogs, username, passwords, etc. are stored. The database is finally connected to mLabs which links to my Heroku deployment. A diagram of the architecture can be seen below:



4.1 Frontend

Although it was possible for me to write Angular 2 applications in ECMAScript 5 (the most common version of JavaScript supported by browsers), I prefered to write in Typescript[18]. Angular 2 itself is written in TypeScript and it helps us at the development stage and includes features that make it easier overall for me to define the Angular 2 components.

In particular, TypeScript supports decorators (sometimes referred to as "annotations") which are used to declaratively add to or change an existing "thing". For example, class decorators can add metadata to the class's constructor function or even alter how the class behaves. As I will demonstrate, Angular 2 components are the key building block for Angular applications. They include a view, defined with HTML and CSS, and an associated controller that implements functionality needed by the view. The controller has three major responsibilities:

- Manage the model, i.e. the application data used by the view.
- Implement methods needed by the view for things like submitting data or hiding/showing sections of the UI.
- Managing data related to the state of the view, such as which item in a list is currently selected.

Because Angular 2 components aren't native JavaScript entities, Angular provides a way to define a component by pairing a constructor function with a view. I do this by defining a constructor function (in TypeScript it's defined as a class) and using a decorator to associate my view with the constructor. The decorator can also set various configuration parameters for the component.This magic is accomplished using the @Component decorator we saw in the first article in this series.

Angular 2 applications are actually made up of a hierarchy of components – they begin with a root component that contains as descendants all the components used in the application. Angular 2 components are intended to be self-contained, because I want to encapsulate my component functions and I don't want other code to arbitrarily reach into our components to read or change properties. At the same time, components do need to exchange data. Angular 2 components can receive data from their parent as long as the receiving component has specifically said it's willing to receive data. Similarly, components can send data to their parents by triggering an event the parent listens for.

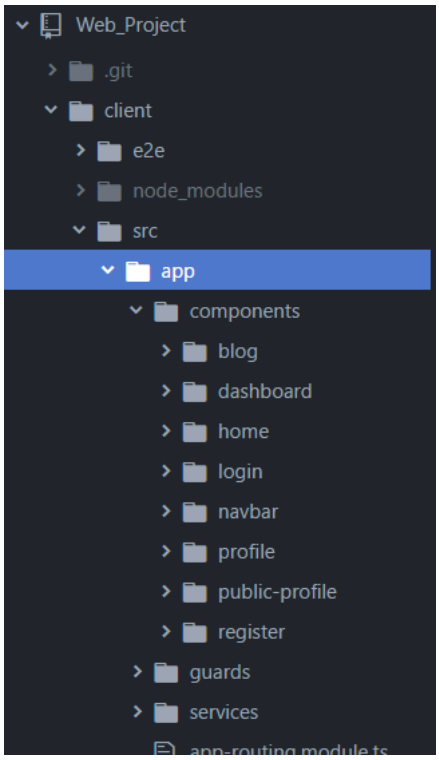
The @Input() decorator defines a set of parameters that can be passed down from the component's parent. The point of making components is not only encapsulation, but also re-usability. Inputs allow us to configure a particular instance of a component.

4.1.1 Serve the Application

To serve the front end locally I need to go into the project folder on my machine, using the windows command-line interface to navigate into the client directory. This is where the application can be launched by running the angular cli command > *ng serve*. The ng serve command builds the app, servers development server, where I can now develop and rebuild the app as you make changes to those files. Note: I have to have my database running with > *mongod* and the index.js running with > *node index.js* at he same time for everything to communicate an work.

Understanding Our Application structure

Using our preferred text editor(in my case I am using Atom) in the project directory, below is what we will find in it.



In the client/src folder is where you will find several files and components, including HTML files that makes up my app. This is the directory I use node to ng serve.

4.1.2 Angular Components

Components are the building blocks of Angular applications. In my project they can be located in the client side in the ../src/app directory. They help display data, listen on user inputs and react on this inputs. A component is similar to directives in Angular 1. It is built with features of Web Components. Every component has a view and a piece of logic. It can interact with services to achieve its functionality. The services can be "Dependency Injected" into the component. Anything that has to be used in view of the component has to be a public member on the instance of the component. The components use property binding to check for changes in the values and act on the changes. The components can handle events and event handlers are the public methods defined in the component's class.

This folder contains 4 main files:
blog.component.css— the component's private CSS styles.
blog.component.html— the component template, written in HTML.
blog.component.spec.ts— The spec files are unit tests for your source files.
blog.component.ts— the component class code, written in TypeScript.

In the . . . /app folder we have AppModule, this is the place where we'll declare all our components, services, and other modules. The generate command already added our components into the appModule when creating a component using:

> **ng generate component blog**

AppModule is where all the components like, LoginComponent, EditBlogComponent, RegisterComponent, etc. We also need to import FormsModule and HTTPModule and declare them as imports. FormsModuleis needed to create the form for our application and HTTPModule for sending HTTP requests to the server.

Routes tell the router which view to display when a user clicks a link or pastes a URL into the browser address bar. A typical Angular Route has two properties:

1. path: a string that matches the URL in the browser address bar.
2. component: the component that the router should create when navigating to this route.

For my project I added the following third property called canActivate, which allows me to help authorize which pages a user can navigate to. This means that somebody can not access a route or url like ../delete-blog/12345 without proper authorization. This further add extra security to the web page allowing only registered users to access the sites content.

3. canActivate: Interface that a class can implement to be a guard deciding if a route can be activated.

If you intend to navigate to the BlogComponent then the URL is something like localhost:4200/blog. Import the BlogComponent so you can reference it in a Route. Then define an array of routes with a single route to that component.

This is how I set it up, so now the router will match that URL to path: 'blog' and display the BlogComponent.

```
import { BlogComponent }  from './blog/blog.component';

const appRoutes: Routes = [
  {path: 'blog',
  component: BlogComponent,
  canActivate: [AuthGuard]
}];
```

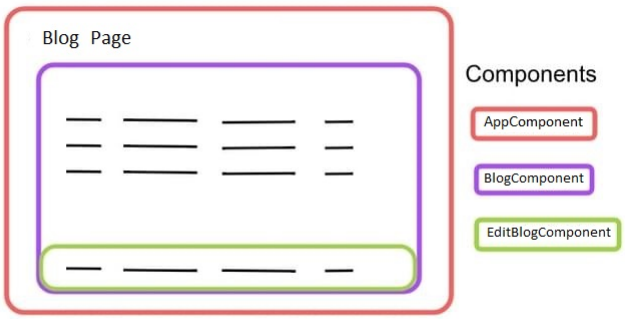
This is how I set it up and now, the router will match that URL to path: 'blog' and display the BlogComponent.

4.1.3 Blog Component

Here we have an example of the app hierarchy with relation to the blog components — the AppComponent at the top, followed by BlogComponent and then EditComponent. For the blog component to work I create a model with mongoose and called it blog.js in the models folder. This is where i'll store the model for the Blog.

To understand how the components work in the front end of my application, we look at blogComponent. This logic in the component.ts file includes:

createNewBlogForm() – Used to create the new blog with a title and body form.



createCommentForm() – Creates form for creating comments.

enableCommentForm()/disableCommentForm() - disable/enable blog comment forms.

disableFormNewBlogForm()/enableFormNewBlogForm() – disable/enable blog title and body forms.

alphaNumericValidation() - Checks the froms input in the valid format.

newBlogForm()– Function to display new body form.

reloadBlogs() – reloads all blogs and locks button for 4 seconds to prevent spamming.

draftComment(id) - Reset the comment form each time users starts a new comment, clearing array so only one post can be commented on at a time, add the post that is being commented on to the array.

cancelSubmit(id) - Check the index of the blog post in the array, remove the id from the array to cancel post submission, reset and enable the form after cancellation, enable any buttons that were locked.

onBLogSubmit() - Submit a new blog post, whilst disabling submit button and locking the form, and creates blog object from form fields. In this field we check if blog was saved to database or not, and return a success or error message depending on if successful.

getAllBlogs() - Function to get all blogs from the database.

likeBlog() / dislikeBlog() – Likes or dislike post and refreshes after doing so.

postComment() - Posts a new comment and saves the comment to the database, whilst adding it to the top of the array.

4.1.4 Edit/Delete Component

Inside the blog I can implement the features, in my project I had an edit blog and delete blog feature. They were used as mechanisms to edit/delete the blogComponent when the blog is created. This was carried out in the .html and .ts files. Inside our files, you can see several instances of [(ngModel)] being used. The unique looking syntax is a directive that implements two-way binding in Angular. Two-way binding is particularly useful when you need to update the component properties from your view and vice versa. I used an event-binding mechanism (Submit) to call the onSubmit() method when the user submits the form. The method is defined inside the component.

4.1.5 Reactive Forms

While reactive forms can be a bit complicated to work with in the beginning, I used them as they allowed for much more flexibility and also helped keep my logic in the component class and my templates simple.

Here are some examples of things that are easy to do with reactive forms:

- Using custom validators
- Changing validation dynamically
- Dynamically adding form fields

To work with reactive forms, i'll be using the ReactiveFormsModule instead of the FormsModule, so I imported it in my app module file.

```
import { ReactiveFormsModule } from '@angular/forms';
```

I used reactive forms a lot in my project to add front end validation, in addition to the already existing back-end validation for extra security. The validateForm method accepts the form to validate, a form errors object and a boolean on weather to check dirty fields or not. The function then loops over all the form controls and checks if there are errors on that control. If there are any, we find the correct error message that came from the validationMessages method and pass back the form errors object.

4.2 Middle Tier

This layer is basically a function that has access to the request and response objects of my application. It can be seen as a series of 'checks/pre-screens' that the request goes through before it is handled by the application. For e.g, I use my middle-ware to determine if the request is authenticated before it proceeds to the application.

4.2.1 Node Server

The Node server is the most crucial part in this tier. I use it for creating a file called index.js and it's were I get the API endpoints. I also defined modules necessary for the application there too. The node server connects to my mLabs deployment and translates the data into JSON so it can be sent back and fourth between tiers.

Module definition

```
Const express = require(‘express’)
Const app = require();
Const mongoose = require(‘mongoose’);
Const bodyParser = require(‘body-parser);
Const port = process.env.PORT || 8080;
```

The API endpoints were located in the routes directory, called authentication.js and blogs.js. They are accessed in index with the folowings commands:

```
app.use(‘/authentication’, authentication);
app.use(‘/blogs, blogs);
```

Some API Endpoints Used

```
router.post('/newBlog', (req, res) => {...}
router.get('/singleBlog/:id, (req, res) => {...}
router.put('/updateBlog/:id, (req, res) => {...}
router.delete('/deleteBlog/:id ', (req, res) => {...}
router.use((req, res, next)) => {...}
```

Connection to mLab

```
Mongoose.connect(config.uri, { useMongoClient: true}, (err) => {
    If(err){
        console.log(‘‘Can’t connect to db: ’’, err);
```

```
    }else{
        console.log(‘‘Conntected to db: ’’ + config.db);
}
});

Module.exports={
uri: ‘mongodb:darren:wegians12@ds239309.mlab.com:39309/my-web-db’,
db: ‘my-web-db’
}
```

4.2.2 Express

Express is used in my Application in conjuncture with our node server. I implemented Express as a utility belt for creating web applications with Node.js. It provides functions for pretty much everything you need to do in order to build a web server. If I were to write the same functionality with vanilla Node.js, I would have to write significantly more code. Here are a couple of simple examples of what Express does:

1. REST routes are made simple.
2. A middle-ware system that allows me to plug in different synchronous functions that do different things with a request or response, ie. authentication or adding properties.
3. Functions for parsing the body of POST requests
4. Cross site scripting prevention tools
5. Automatic HTTP header handling

4.2.3 Herouku and mLabs (Cloud Hosting)

In order to use our application on a web browser I need to find a way of deploying it online. I use mLabs combined with Heroku[16, 15], one as the cloud database service that hosts my MongoDB and the other to deploy my the application.

For Heroku I had a few problems when it came to deploying it. I set the path for connecting server to index.html in a folder called 'dist' that github ignores when it is uploaded to herokus git branch. Another issue was Angular had its own dependencies in its package.json, and because the back-end was outside of that visible

structure none of those dependencies would build, therefore ng build wouldn't work. I fixed this by changing "outDir": "dist" to "outDir": "../public", creating a new public folder to be used with the server. Then I connect the server to angular 2 index.html to this public folder instead.

```
app.get('*', (req, res) => {
    res.sendFile(path.join(__dirname + '/public/index.html'));
});
```

I got heroku working through the command line tool. I used the cmd to navigate to the directory of my project. I used 'heroku status' to check that I had it installed on my machine. I also checked all my files were committed with 'git status' and if not I would use 'git add'. Then I executed it with 'git commit -m "deployment"'. I ran heroku to create it it and then it automatically generated its own name for me. Next, with 'heroku open' command I could navigate to my application url. This was only a blank template and it was where I would eventually push all my code to via Heroku's GitHub link. Lastly this was done by entering 'git push heroku master' into the cmd. My project was then live and could subsequently be found at https://mighty-island-46941.herokuapp.com

4.3 Backend

My backend entails creating a MongoDB database and having my application connect to it[20]. I will also want to create mongoose models so I could use mongoose to interact with my database. The mongoDb is where I stored all the data about my blogs and the users information. This mongoDB instance was connected to mLabs deployment in order to serve it online where it could be accessed. Otherwise our database is stored locally and would not work when the application is hosted on heroku[15]. For this project I created two models blog and user, wherI have a userSchema to deal with the users credentials like email, username and password. And a blogSchema to deal with propertiees seen below:

```
const blogSchema = new Schema({
  title:  {type: String, required: true, validate: titleValidators},
  body:  {type: String, required: true, validate: bodyValidators},
  createdBy:  {type: String },
  createdAt:  {type: String, default: Date.now() },
```

```
  likes:  {type: Number, default: 0 },
  likedBy:  {type: Array },
  dislikes:  {type: Number, default: 0 },
  dislikedBy:  {type: Array },
  comments:[
    {
      comment:{type: String, validate: commentValidators},
      commentator:{type: String}
    }
  ]
});
```

This is how a Schema can be defined to be utilised by the database. I grabbed mongoose and mongoose.Schema. Then I can define the attributes on the blogSchema for all the things I need for my blog. I then need to export my mongoose database module/Schema, so I could use my defined models from every module in my program:

```
module.exports = mongoose.model('Blog', blogSchema);
```

Mongoose provided me with built-in and custom validators, and synchronous and asynchronous validators. It allows me to specify both the acceptable range or values and the error message for validation failure in all cases. I created numerous validation fields to be applied to my schema instances. The example below shows how I specified some of the validator types and error messages:

```
const titleValidators = [
  {
  validator: titleLengthChecker,
  message: 'Title must be between 5 and 50 characters'
  },
  {
    validator: alphaNumericTitleChecker,
    message: 'Title must be alphanumeric'
  }
];
```

This is then used in the blogSchema - title: type: String, required: true, validate: titleValidators

I created multiple validators to check that all the input would be valid. Email was checked to have the correct format using a regular expression and be of the correct length between 8 and 30 characters. Username and password also had to abide by these validation rules. The same concept was applied to all the blog field elements. Simply put, my models represent a collection of documents in the database that I can search, while the model's instances represent individual documents that you can save and retrieve.

4.3.1 Json Tokens

One of the key components of the API is the authentication system. I was very concerned about it early on in the application's development because I have found it to be one of the most difficult pieces to implement and it is critical to do so properly. An insecure authentication system puts user data at risk and seriously threatens the security of the application. After researching various methods of securing Node.js APIs, I determined that JSON Web Token (JWT) is a commonly used, supported, and effective solution to this problem.

JWT is an open internet standard (RFC 7519) that defines a methodology of securing and transmitting information between multiple parties [13]. It consists of a JSON object with a header, a payload, and a signature that is passed between the API and the API user and helps verify the identity of the API user [13]. The header of the object identifies that the object is a JWT and the specific encryption hashing algorithm that is being used. The payload contains statements about the entity like the username, and user ID and may sometimes contain information about when the token was issued and when it expires. These token issue and expiration dates enables the API to require users to authenticate on a regular basis for added security. The signature portion of the token is made up of a hash of the previously mentioned header and payload in addition to a secret – which is only held by the API and is used to sign the JWT – using the encryption algorithm that is specified in the header. This signature is joined to the Base64URL encoded header and the Base64URL encoded payload by periods that separate the components. Here is an example of a JWT that is fully encoded, taken from the official JWT website jwt.io.[13]

eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJzdWIiOiIxMjM0NTY3ODkwIiwibmFtZSI6IkpvaG4gRG9lIiwiaWF0IjoxNTE2MjM5MDIyfQ.XbPfbIHMI6arZ3Y922BhjWgQzWXcXNrz0ogtVhfEd2o

It is important to note that critical information like passwords should never be stored in the payload of a JWT because the payload is only encoded in Base64URL, which is easily decoded and is not secure. Instead, passwords should only be used to acquire the JWT from the API.

I decided used bcrypt to encrypt user passwords, as bcrypt is significantly slower than other commonly-used one-way encryption algorithms like MD5, SHA-256, SHA-512, and SHA-3 [10]. According to Coda Hale, a software engineer at Stripe, modern servers can crack passwords that were encrypted with MD5 using brute force in a matter of seconds [10]. Since bcrypt is a much slower algorithm, Hale asserts that a password encrypted with bcrypt will take over a decade to crack on modern servers, even if the password is lowercase, alphanumeric, and is only six characters long [10]. Additionally, the algorithm can be adjusted to suit the computational power of attackers in the future in keeping up with Moore's law because the computational expense of performing the algorithm can be adjusted.

5 System Evaluation

5.1 Mistakes and Future Development

Going forward, if I were to continue developing the web application, I would like to make several modifications to the API and UI layers. Most of these modifications are relatively minor in terms of scope, as there will always be a million little things that will need to be taken care of. The major changes generally relate to the almost-finished status of the minimum viable product, the current user experience, and fulfilling the overall vision of the application.

The minimum viable product is essentially complete but I feel it is possibly missing a few components. Users are unable to change their password or email address, which is something that I would like to have to seen incorporated. These issues are minor and should not be difficult to finish implementing; unfortunately there just was not enough time to do so.

The development experience, I feel could also be improved, particularly in the area of testing. A limited form of automated testing as already been set up but there is not full test coverage. The rest of the tests should be created in the future as a priority in order to reduce the amount of bugs that are introduced into the codebase. It would be ideal if the existing automated testing system could be expanded in the future to also be integrated with the production API server. As it currently stands, when a commit is pushed to the master GitHub repository Travis CI runs the automated tests. Travis CI also allows developers to deploy their working code after all the tests have passed so that the pipeline from developer to user is completely continuous and never stops flowing.

Again, there are a few additional enhancements that I would like to make to the application. One of these enhancements is tighter API security. The password hashing system and JWT authentication system are a good start but more can be done in this

area. In the current JWT implementation, the tokens do not become invalid until they expire. If a token became compromised, the attacker would have full access to the application and there would not be a solution to stop them. Some JWT libraries support the addition of a unique token identifier called a JTI to the token's payload, which allows system administrator to lock out tokens with the specific identifier.

Lastly, the most challenging obstacle was working as a lone developer. I underestimated the difficulty of working with such a intricate infrastructure without the aid of a like-minded team member. In particular, I found that not being able to bounce ideas off another person and having them provide mutual critique in order to get solid guidance and feedback inhibiting. This also proved difficult when hitting inevitable development dead-ends along the way. Being solely responsible for all sections of my project including the infrastructure, research, and documentation I found this to be a momentous task to have undertaken which might have benefited more from additional support. Definitely having two or more heads I feel would have allowed for a much smoother and more effective experience.

5.2 Conclusion

In conclusion, although the project has had its challenging moments I found that overall it has been an extremely interesting and a highly rewarding experience. I have successfully accomplished my ambition to learn more about full stack web development, this being one one of my main goals when I initially started the project. As were the following:

- Understand the principles of creating an effective full stack web page, including an in-depth consideration of information architecture.
- To create a full stack web application that would require a user to be able to login to the website by creating their own unique, secure account with a username, email and password.
- Registration and login would incorporate authentication to check that the formatting of the email and passwords are correct, and the the username or emails are not already taken by a previous user.
- Provide guarded routes, so blog users can edit only their own post and not others.

- Each user then could create their own blog post on a blog page. It could be interacted with by the users, allowing them to comment, like, or dislike posts.
- Deploying the application so it can be accessed and used online.

I managed to achieve most of the objectives I set out. The first objective was accomplished by being able to produce a fully functioning web application that utilized technologies like, MongoDB, Express, Angular, Node.js, Restful API, JSON web tokens, authGuard, and more. I became familiar with web development after researching various possible ways of developing my final product, and also having spent so much time developing it with the MEAN stack infrastructure. The second and third goals of allowing a user to create a secure profile or account was done by taking the input from forms in the front end and sending to backend to be validated and checked if already created in database. This was achieved with the userschema in the models folder, and also used API. Guarded routes were another objective, using the authGuard it allowed me to project deny or allow access to specific webpages. For instance, I only allowed users that were not logged in to view the home, login, and register pages. Once signed in authGuard grants access to blog, edit-blog, profile, etc. The blogs API is used to only allow a user to edit or delete their own blog post. The user can however leave likes and comments on other user's blog, by utilizing put and post methods in blogs API. Lastly, I was able to deploy the application online, so anybody could use it. The setup involved hosting my applications local database with mLabs, so when the application was used data could be stored and accessed. To deploy my app to Heroku, I used the git "push" command to push the code from my repository's master branch to my Heroku remote, where it was hosted. This part of the setup took a bit of research as I encountered a few problems, be with determination, and some trial and error I successfully managed to overcome it.

I feel I have learned more about software development over the past few months than I have from the previous three years of academic instruction. The hands-on, real-world experience cemented and exemplified the ideas of why Agile software development is so beneficial, why testing is so important, and why the architecture of applications matter. I was able to build a full-stack web application through copious amounts of research and experimentation and gain some DevOps skills in the process through hosting it on a Heroku. The skills that I have developed and the experience that I have gained has made me a better software developer and I will be able to use this knowledge moving forward in my career.

Bibliography

[1] MEAN Stack, https://www.sitepoint.com/introduction-mean-stack/

[2] MongoDB, https://www.mongodb.com/nosql-explained

[3] Nodejs, https://nodejs.org/en/

[4] V. Karpov. (2013, Apr. 30). The MEAN Stack: MongoDB, ExpressJS, AngularJS and Node.js [Online]. Available: http://blog.mongodb.org/post/49262866911/the-mean-stack-mongodbexpressjs-angularjs-and

[5] Express, https://expressjs.com/

[6] Full Stack, https://blog.udacity.com/2014/12/front-end-vs-back-end-vs-full-stack-web-developers.html

[7] LAMP vs MEAN, https://www.programmableweb.com/news/what-mean-stack-and-why-it-better-lamp/analysis/2015/12/22

[8] Agile for Solo Developer, https://www.techrepublic.com/article/agile-programming-works-for-the-solo-developer/

[9] RESTful API, https://mlsdev.com/blog/81-a-beginner-s-tutorial-for-understanding-restful-api

[10] C. Hale. (2010, Jan. 31). How to Safely Store a Password [Online]. https://codahale.com/how-to-safely-store-a-password/

[11] Angular, https://angular.io/

[12] http://www.methodsandtools.com/archive/archive.php?id=47

[13] JWT.IO JSON Web Tokens, https://jwt.io/ https://devcenter.heroku.com/

[14] GitHub, https://github.com/features

[15] Heroku, https://devcenter.heroku.com/articles/github-integration

[16] mLab, https://mlab.com/

[17] Postman, https://www.getpostman.com/products

[18] Typescript, https://blog.codewithdan.com/2017/08/26/5-key-benefits-of-angular-and-typescript/

[19] Angular 1 and 2, https://medium.com/@angularminds/comparison-between-angular-1-vs-angular-2-vs-angular-4-62fe79c379e3

[20] Mongoose, http://mongoosejs.com/