SET008101 Coursework 2 Report

Matthew DelSordo
40340571@live.napier.ac.uk
Edinburgh Napier University - Module Title (SET08101)

1 Introduction

The premise of this coursework was to design and implement a simple blog platform in Node.js comprised of a client element that displays the blog and a server-side CRUD API that the client uses to obtain functionality. My resulting blog page (Blegg, a blog about eggs) is able to handle multiple users who are able to create accounts, log in, and create, update, and delete blog posts. Also present are security features that prevent unauthorized users from editing or deleting others' posts and that encrypt login information to both preserve users' sessions and prevent unauthorized logins.

I decided to create my blog as a React app as I had some pre-existing experience using the library. My goal was to implement the required features at a robust, scalable level, and I decided to use a number of libraries such as Redux and MongoDB, as a data store, to this end. My hope was that these technologies would allow me to create a more powerful and extensible blog in less time, so I could then add support for multiple user accounts with sub-blogs. In practice, the added nuances and complications of the technologies I used led to more time being spent fiddling with the bones of the project, at some points, than was spent actually implementing features. I ended up with a blog that makes up for what it lacks in special features with handling those features very robustly.

2 Software Design

2.1 Use Case Diagram

My first step in planning out the design of my blog was creating a use case diagram to outline what sorts of functionality I wanted the blog to have (Figure 1). I wanted all visitors to be able to view blog posts and post comments on them as well as sign up for accounts. Signed-in users would then be able to log in and out, write, edit, and delete posts, and update their personal information. While I implemented most of these features, commenting and updating one's information had to be left out for lack of time.

2.2 Wireframes

I next created wireframes of the various pages of the client in order to have something to go off of when implementing the client-side HTML (Figures 2, 3, 4, 5, and 6).

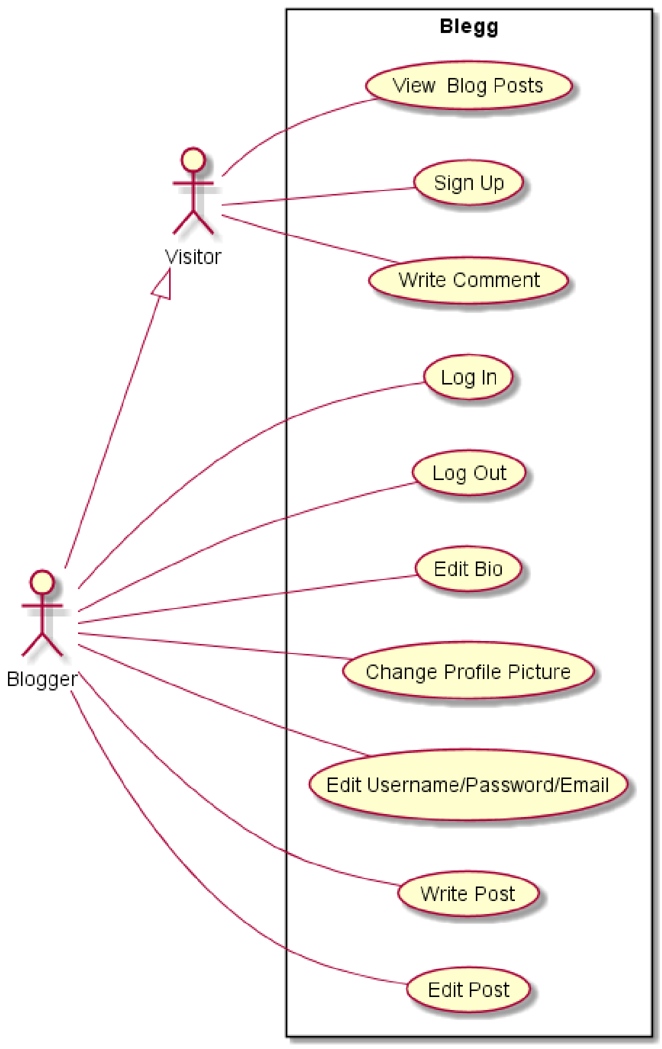


Figure 1: Use Case Diagram

2.3 API

While I did not do this next chronologically, I soon realized that specifying an API in advance would be crucial in order to keep track of what routes the server was serving data on so that I could work on one side of the project at a time rather than implementing one feature at a time, which involved keeping track of the complicated flow of data around the app (Figure 7).

2.4 Tech Stack

Going in to this project I failed to account for the potential complexity of setting up what is essentially a MERN (MongoDB, Express, React, Node.js) stack. The boilerplate for

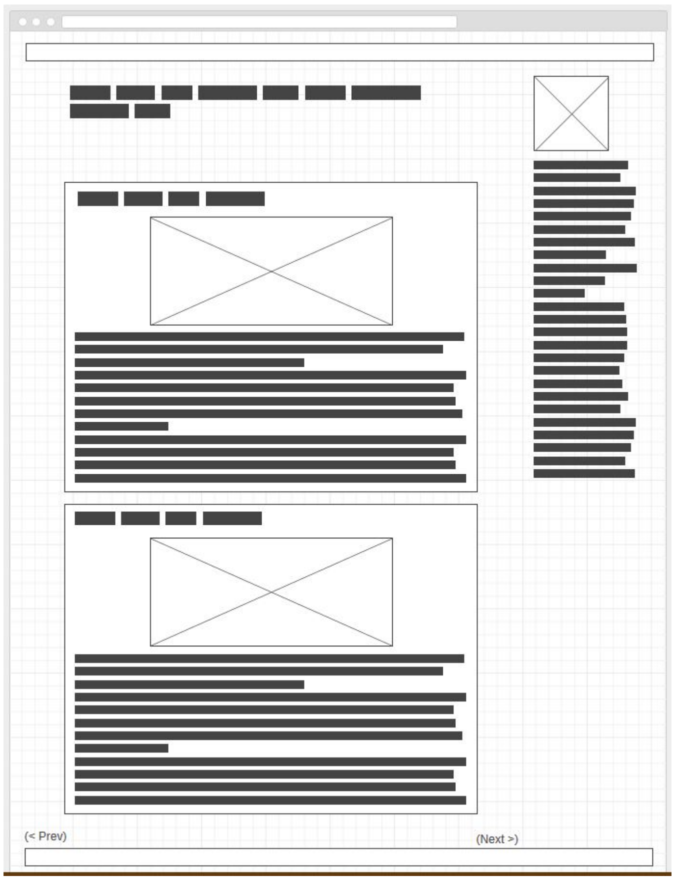


Figure 2: Blog Page Wireframe

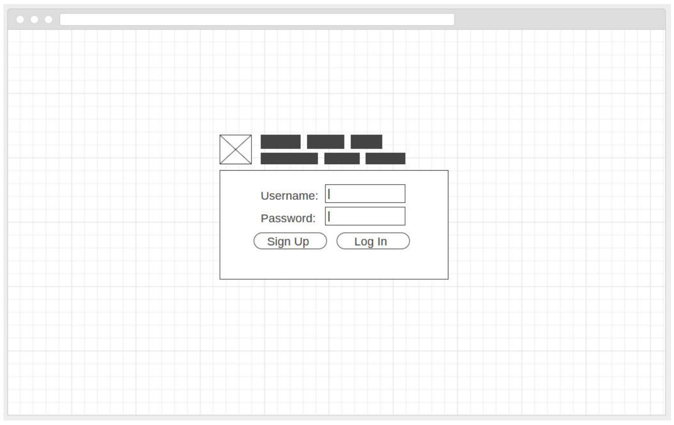


Figure 3: Login Page Wireframe

my project is based heavily off the tutorial created by GitHub user verekia.[1] While said tutorial contains a lot of useful information and helped me out a ton, some of the technologies referenced in the guide ended up being far too complex for what I needed them to do, and this slowed down my development process considerably. What follows are explanations of the major node modules that my project relies on, along with assessments of their in-practice usefulness for this project.

- **Babel** Translates the React code in my project into code that the server and browser can actually interpret. As React notation is not standard JavaScript, this package was absolutely essential. It also allowed me to use fancy ES6 JavaScript which is more concise, which is nice.

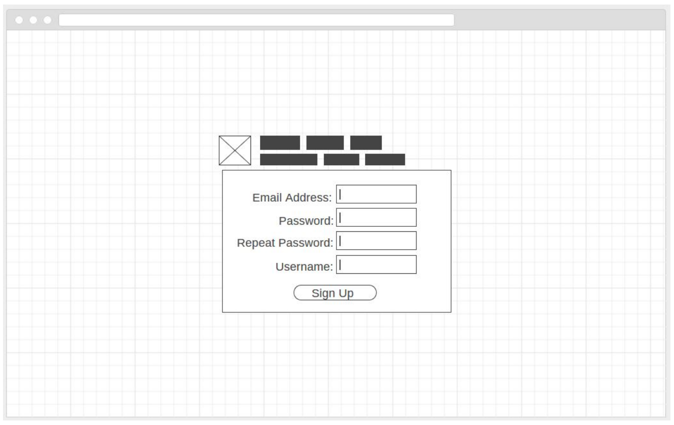


Figure 4: Signup Page Wireframe

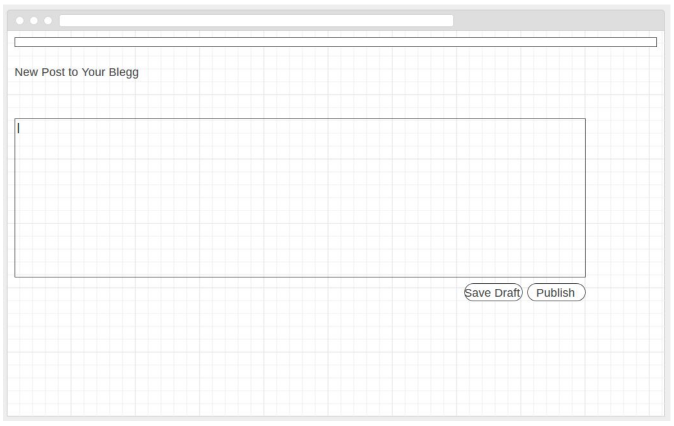


Figure 5: Post Page Wireframe

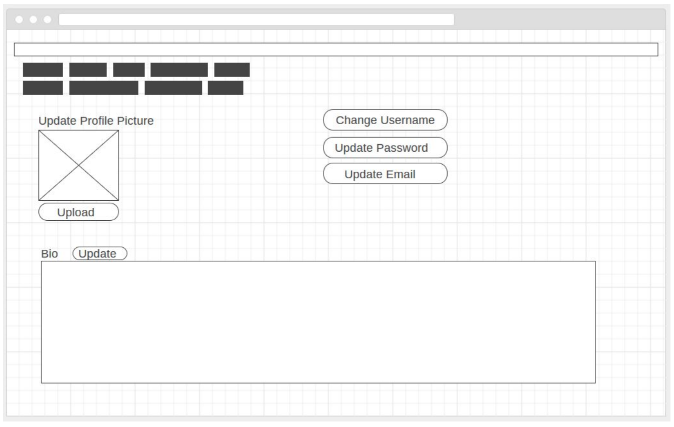
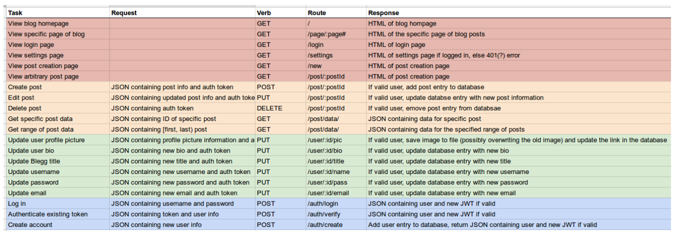


Figure 6: Settings Page Wireframe

- **ESLint** Used by my text editor to check code for validity and upgraded to work with React. Very useful to check mistakes but not strictly necessary.
- **Flow** Used by my text editor to type-check code. I can see this being useful if I actually knew how to use it properly, but I honestly ignored it in most cases and ran in to very few type problems (aside from those caused by Immutable.js).



Task	Request	Verb	Route	Response
View blog homepage		GET	/	HTML of blog hompage
View specific page of blog		GET	/page/:page#	HTML of the specific page of blog posts
View login page		GET	/login	HTML of login page
View settings page		GET	/settings	HTML of settings page if logged in, else 401(?) error
View post creation page		GET	/new	HTML of post creation page
View arbitrary post page		GET	/post/:postId	HTML of post creation page
Create post	JSON containing post info and auth token	POST	/post/:postId	If valid user, add post entry to database
Edit post	JSON containing updated post info and auth toke	PUT	/post/:postId	If valid user, update databse entry with new post information
Delete post	JSON containing auth token	DELETE	/post/:postId	If valid user, emove post entry from databsae
Get specific post data	JSON containing ID of specific post	GET	/post/data/	JSON containing data for specific post
Get range of post data	JSON containing [first, last) post	GET	/post/data/	JSON containing data for the specified range of posts
Update user profile picture	JSON containing profile picture information and a	PUT	/user/:id/pic	If valid user, save image to file (possibly overwriting the old image) and update the link in the database
Update user bio	JSON containing new bio and auth token	PUT	/user/:id/bio	If valid user, update database entry with new bio
Update Blegg title	JSON containing new title and auth token	PUT	/user/:id/title	If valid user, update database entry with new title
Update username	JSON containing new username and auth token	PUT	/user/:id/name	If valid user, update database entry with new username
Update password	JSON containing new password and auth token	PUT	/user/:id/pass	If valid user, update database entry with new password
Update email	JSON containing new email and auth token	PUT	/user/:id/email	If valid user, update database entry with new email
Log in	JSON containing username and password	POST	/auth/login	JSON containing user and new JWT if valid
Authenticate existing token	JSON containing token and user info	POST	/auth/verify	JSON containing user and new JWT if valid
Create account	JSON containing new user info	POST	/auth/create	Add user entry to database, return JSON containing user and new JWT if valid

Figure 7: Blog API

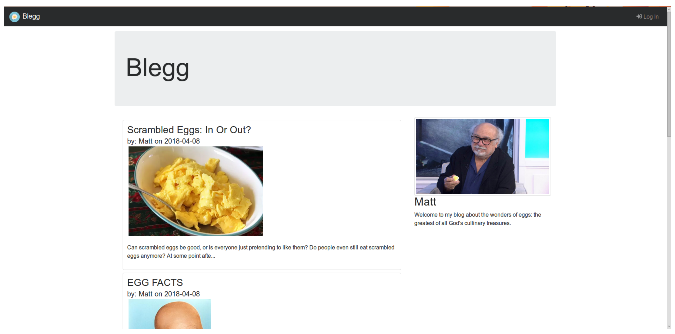
- **Nodemon** A utility that monitors changes in the code and restarts the server when code updates. Super useful for development but not necessary for the site to run.
- **PM2** A Node process manager that is used in "production" mode to manage the server's processes. This package is honestly way overkill for my tiny blog platform that definitely only ever has to serve one request at a time, and I could have not included it and been fine.
- **Webpack and webpack-dev-server** Webpack translates all the client code in my project into a single bundle that is served to the user in order to minimize the number of files that they have to download. It is plugged in to Babel in order to interpret everything properly and like Babel is essential for React to work properly. Webpack-dev-server runs in "development mode" in order to repack everything on the fly whenever Nodemon notices an update.
- **Bcrypt** Encryption library that I use to encrypt user passwords. Even though my platform will have few users and probably not be exposed to the wider internet, I wanted to have some practice with best safety practices for storing confidential user information.
- **Body-parser, CORS, express-fileupload, and compression** Middleware used by Express to decipher JSON information sent along with incoming API requests. Without these I would not be able to send new information from the client to the server.
- **Bootstrap** A CSS library that I used for styling. While it is not the most imaginative choice, Bootstrap saved me time fiddling with custom CSS and provides lots of professional looking and already responsive design elements.
- **Express** Provides server functionality and acts as the backbone of the entire project. The server listens to various routes and serves up data to any clients who connect to those routes using the proper protocols.
- **ImmutableJS** This package creates data types that cannot be modified except by their proprietary methods in order to ensure that the data in them cannot be mutated. This condition is apparently ideal to have for your Redux state as said state should only be modifiable by Redux actions, but in practice this package created so many more problems than it solved. The fact that the getters of these objects are different than in vanilla JavaScript caused me to use the wrong notation all over the place, that was not immediately apparent in the code. Not worth it in the long run.
- **JQuery** Used in a couple spots to handle small client UI updates.
- **JSONWebToken** Used to create cookies that store encrypted user data to protect the user's private information while keeping them logged in to the blog platform. The tokens act both as an authentication tool and also as sources of user info which minimizes the amount of times the server has to authenticate the user.
- **Mongoose** A standard library for interfacing with MongoDB. Although using MongoDB was probably overkill as far as data storage goes, I wanted to learn how to use it as it is a popular and widespread piece of software. I used mLab to host collections that store user and post information, and Mongoose made interfacing with them very straightforward.
- **React** A library for encapsulating HTML components that can render different structures based on their state and automatically update when their state changes. I really like React because it lets you do object oriented programming with HTML and do not have to worry about handling state change events or selecting specific HTML components to update at specific times.
- **React-Router** A library that works with React to basically circumnavigate server routing, so that from page to page the entire React app is not re-rendered and re-downloaded by the user. While this is very useful and I definitely needed it for the purposes of the React app, this library introduced the second-largest amount of headache to the coursework (after Redux), as it handles navigation with HTML tags in a way that I found somewhat unintuitive and confusing compared to routing purely with Express.
- **Redux** A library that creates a single central state for your app in which you can store data. This works very well with React (through react-redux) in order to condense the states of all of your react components into a single universal state that can then propagate down to your UI. While Redux is very powerful, this project left me wondering how much I really needed it and if that was worth the complications it introduced, which I will get in to in the Personal Evaluation section.

3 Implementation

My blog platform is made up of the following pages that have the following functionality:

- **Home Page** Displays a chronological listing of all posts to the site. Posts are displayed as minimized views that can be clicked to navigate to the post-specific page. (Figure 8)
- **Post-Specific Pages** Each post has its own page that shows the post in full. If the current logged-in user matches the author of the post, options to edit or delete the post are made available. (Figure 10)

- **Sign In/Sign Up Pages**These pages contain forms that accept user info and send it to the server to be authenticated. If the server finds a user that matches the login info it returns a new authorization token that gets stored in local storage to keep track of the user's session. This token is sent back to the server when the page is refreshed to determine which user is visiting the page. Sign-up information is used to create a new user in the database, after which an authorization token is also returned. (Figure 9)
- **Post Creation/Edit Page** The New Post page and each Edit Post page allows the user to edit or create a post. A post is made up of a title, an optional image, and some text content. If the logged-in user does not match the author of the post to which the edit request is made, the request is rejected. (Figures 11 and 12)



Can scrambled eggs be good, or is everyone just pretending to like them? Do people even still eat scrambled eggs anymore? At some point afte...

Welcome to my blog about the wonders of eggs: the greatest of all God's cullinary treasures.

Figure 8: Blog Home Page

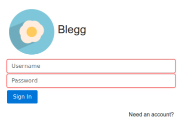
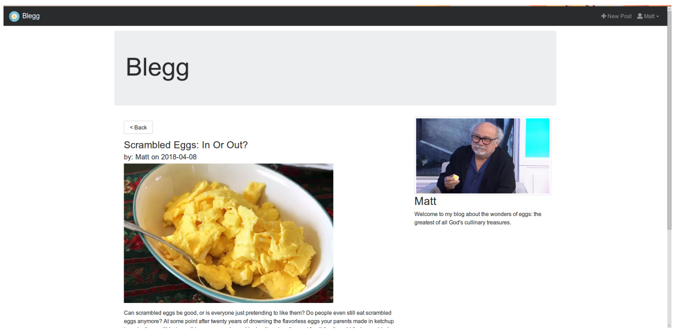


Figure 9: Blog Login Page



Can scrambled eggs be good, or is everyone just pretending to like them? Do people even still eat scrambled eggs anymore? At some point after twenty years of drowning the flavorless eggs your parents made in ketchup

Welcome to my blog about the wonders of eggs: the greatest of all God's cullinary treasures.

Figure 10: Blog Post Page

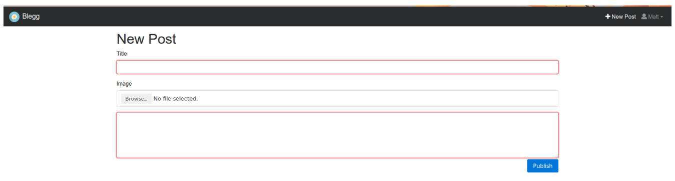
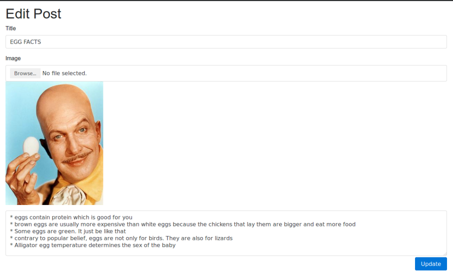


Figure 11: Blog New Post Page



* eggs contain protein which is good for you
* brown eggs are usually more expensive than white eggs because the chickens that lay them are bigger and eat more food
* Some eggs are green. It just be like that
* contrary to popular belief, eggs are not only for birds. They are also for lizards
* Alligator egg temperature determines the sex of the baby

Figure 12: Blog Edit Post Page

4 Critical Evaluation

Compared to popular blogging platforms, my implementation is very light on features, only allowing for the bare minimum of information-display functionality. This satisfies the requirements of the coursework, but I wish I had more time to implement a few more features. I intended to implement commenting and the ability to modify user information, like one's profile photo, bio, and blog title. Another feature I found myself wishing I had time to include during testing was the development of an improved text editor and support for a variable amount of images for posts. I had hoped that HTML in my post content would be injectable into my layouts, but the fact that it is not is far better for security anyway.

5 Personal Evaluation

My own major takeaway from this project is that I need in the future to identify the minimal quantity of technologies absolutely necessary in order to solve the problem at hand. That said, grappling with the libraries I used taught me how they work, and I can definitely identify which of them I would need to accomplish certain tasks as opposed to others, which I could not have done at the beginning of this coursework. Below, IâĂŹd like to highlight the pros and cons of my approach.

5.1 Pros

- I find React intuitive because it turns designing HTML into programming with objects. It makes it easy to composite together large, complex, and dynamic layouts from a number of smaller, independent files.

- Redux creates a single point of contact between the clientâĂŹs session and the server. Instead of having to send AJAX requests from individual scripts in individual pages, the client instance as a whole talks to the server.
- ReduxâĂŹs single state also makes it very easy to keep track of the active user from page to page, as long as the user does not manually refresh.

5.2 Cons

- Having to do all server-client communications through Redux middleware is a gigantic hassle. Each asynchronous call is made up of three Redux actions which all have to be dispatched and reduced separately in order to put the app in discrete loading, success, and failure states. I understand why doing it this way is a good idea (you would not want the page to hang while you wait for data) but it is a lot of code, not even counting the code on the server end.
- There are some cases where doing AJAX requests on behalf of the app as a whole was not ideal, for example, in the post displaying components. Only the display needs the post data from the database, not the whole app, but that is how it has to be rigged up if you do it through Redux (as far as I could gather).
- The entire project is gigantic. The amount of data being sent to the client is far more than if the server served conventional HTML, CSS, and JS, and the huge amount of Node dependencies puts the project file at something like 200MB. There are so many files in node modules that it is faster to re-download all dependencies with 'npm install' than it is to copy or move node modules. In one of the labs, it was said that if you have more layout files than pages, you're doing it wrong. By that metric I have done a really poor job (if you don't count the potentially infinite quantity of post pages, at least).
- In general, using fancy libraries to create desirable non-standard web page behaviors can create other undesirable behavior that can be hard to predict, interpret, and debug because they differ from that of a "normal" webpage.

References

[1] J. Verrecchia, "Javascript stack from scratch." https://github.com/verekia/js-stack-from-scratch/, 2017.

[2] reacttraining.com, "React router." https://reacttraining.com/react-router/web/guides/philosophy.

[3] J. Watmore, "React + redux - jwt authentication tutorial & example." http://jasonwatmore.com/post/2017/12/07/react-redux-jwt-authentication-tutorial-example, 2017.

[4] Facebook, "Immutable js docs." http://facebook.github.io/immutable-js/docs/#/.

[5] N. Foundation, "Express 4.x api reference." http://expressjs.com/en/api.html, 2017.

[6] O. Garuba, "5 steps to authenticating node.js with jwt." https://www.codementor.io/olatundegaruba/5-steps-to-authenticating-node-js-with-jwt-7ahb5dmyr 2017.

[7] Twitter, "Introduction - bootstrap." https://getbootstrap.com/docs/4.0/getting-started/introduction/.

[8] A. Erdeljac, "File upload with node & react." https://levelup.gitconnected.com/file-upload-with-node-js-react-js-686e342ad7e7, 2018.