Northern Michigan University, Winter 2019

Fiesta MARKET

Melissa Farrell

https://fiestamarket.app

Table of Contents

Project Overview

Fiesta Market is a React web app that enables players of the MMORPG Fiesta Online to coordinate the buying and selling of in-game items outside of the game. The current system requires players to set up a shop in-game and leave the game client open while vending their items. This system has multiple flaws including that the players must leave the game running for long periods of time as well as there is no good way to take offers on items - if you leave it up to potential buyers to message you, you may disconnect from the server while “AFK” and never see the offer messages.

I chose this project because I felt that it would provide a challenge while being something fun that solves a problem that I myself have had to deal with. By choosing to create a web app, I did have some advantage in having taken courses in client and server-side development, experience working with DigitalOcean, and creating a Wordpress site from the ground up. However, one of the major factors in my choice of this project was the opportunity to work with the React framework, which I went into this project with zero experience with. This project also required creating a website with an API backend, user accounts, authentication, and user roles from the ground up - something I had not done before.

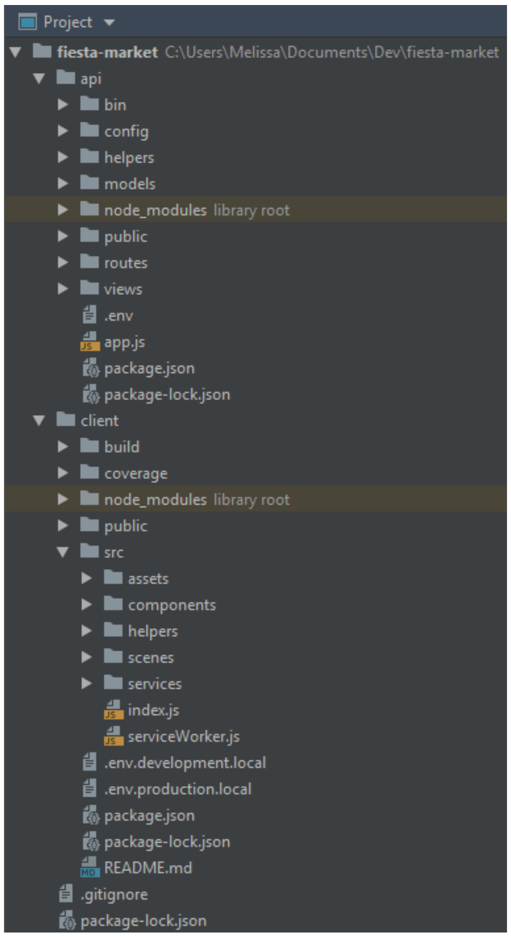
Ultimately, the goal with this project was to get something as close to full-featured as possible and that relieves some of the biggest pains points of selling and buying items for Fiesta Online. I have learned an immense amount from this project and I’m looking forward to expanding on this project further.

Hosting

For the database, I chose MySQL as I have some familiarity with it. I worked with MySQL Workbench for much of my direct interaction with the database. I opted to host through Amazon Lightsail, because of affordability and how simple the process was for set up. Hosting it on Lightsail also meant that I could host the back-end Node.js API alongside it and add some further security through limiting database access to only my other Lightsail resources. And so, Lightsail is also where the backend is hosted. I did later consider the option of AWS Elastic Beanstalk but ultimately determined that leaving my set-up on Lightsail would be my best option for now when considering where my time would be best spent.

For the React app front-end, I opted to use Amazon's Simple Storage Service (S3) for hosting. This process involved building my React app and simply adding it into the designated bucket for the website, and it proved the simplest option. I picked up the website domain through Google domains and opted for .app for the top-level domain (TLD). An interesting note here is that all .app sites require HTTPS by default. Since HTTPS was something I was going to be implementing for the site anyway, this wasn't an issue for me. Domain management runs through Amazon's Route53 and CloudFront.

Being familiar with Digital Ocean and not wanting to shell out a lot of cash, Amazon Web Services seemed like the best option for me to learn something new as well as keep it affordable to host the project. Further, many AWS services are made to work together nicely, which made it an even more appealing option. Overall, the set-up process for all of these services went as painlessly as this sort of thing can and I learned a lot about the specifics of AWS' services.

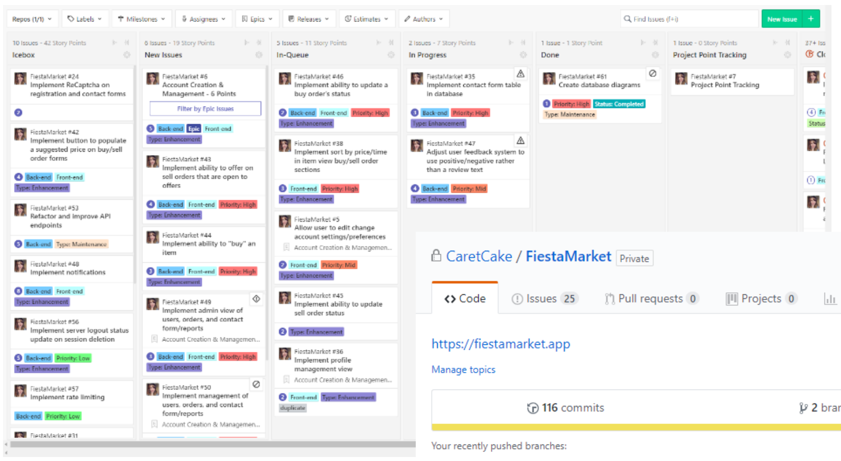


```
Project
▼ fiesta-market  C:\Users\Melissa\Documents\Dev\fiesta-market
  ▼ api
    ▶ bin
    ▶ config
    ▶ helpers
    ▶ models
    ▶ node_modules  library root
    ▶ public
    ▶ routes
    ▶ views
      .env
      app.js
      package.json
      package-lock.json
  ▼ client
    ▶ build
    ▶ coverage
    ▶ node_modules  library root
    ▶ public
    ▼ src
      ▶ assets
      ▶ components
      ▶ helpers
      ▶ scenes
      ▶ services
        index.js
        serviceWorker.js
      .env.development.local
      .env.production.local
      package.json
      package-lock.json
      README.md
    .gitignore
    package-lock.json
```

Project Structure and DevOps

The entire project structure consists of two main directories: api and client. The api directory consists of the Node.js backend API that handles the database interaction and API requests from the client-side. The client directory consists of, well, the React client. Each directory is split into various folders to improve readability and ease of navigating the project.

My work throughout the course of the project was all done with the use of a private Github repository. While this was not new to me, I did get experience working with branches, which is a new concept for me, as well as with connecting commits to issues and issue tracking with Zenhub. I also made use of environment variables both in my React client and Node.js API to streamline the process of going from development to production settings.



10 Issues - 42 Story Points
Icebox
- FiestaMarket #24 Implement ReCaptcha on registration and contact forms — 2
- FiestaMarket #42 Implement button to populate a suggested price on buy/sell order forms — 4 Back-end, Front-end, Type: Enhancement
- FiestaMarket #53 Refactor and improve API endpoints — 5 Back-end, Type: Maintenance
- FiestaMarket #48 Implement notifications — 8 Back-end, Front-end, Type: Enhancement
- FiestaMarket #56 Implement server logout status update on session deletion — 5 Back-end, Priority: Low, Type: Enhancement
- FiestaMarket #57 Implement rate limiting — Back-end, Priority: Low
- FiestaMarket #31

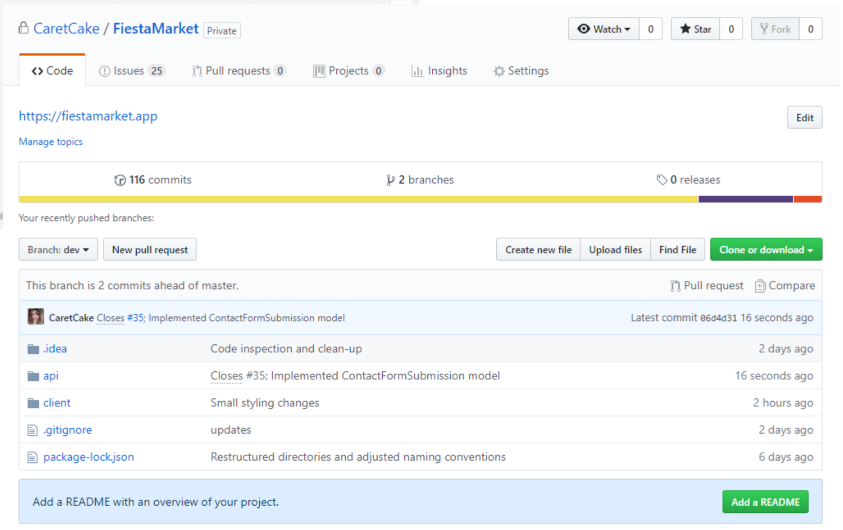
6 Issues - 19 Story Points
New Issues
- FiestaMarket #6 Account Creation & Management - 6 Points — Filter by Epic Issues — 5 Back-end, Epic, Front-end, Type: Enhancement
- FiestaMarket #43 Implement ability to offer on sell orders that are open to offers — 4 Back-end, Front-end, Priority: High, Type: Enhancement
- FiestaMarket #44 Implement ability to "buy" an item — 3 Back-end, Front-end, Priority: High, Type: Enhancement
- FiestaMarket #49 Implement admin view of users, orders, and contact form/reports — Account Creation & Managemen... — 5 Back-end, Front-end, Priority: High, Type: Enhancement
- FiestaMarket #50 Implement management of users, orders, and contact form/reports — Account Creation & Managemen...

5 Issues - 11 Story Points
In-Queue
- FiestaMarket #46 Implement ability to update a buy order's status — 2 Back-end, Front-end, Priority: High, Type: Enhancement
- FiestaMarket #38 Implement sort by price/time in item view buy/sell order sections — 3 Front-end, Priority: High, Type: Enhancement
- FiestaMarket #5 Allow user to edit change account settings/preferences — Account Creation & Managemen... — 2 Front-end, Priority: Mid, Type: Enhancement
- FiestaMarket #45 Implement ability to update sell order status — 2 Type: Enhancement
- FiestaMarket #36 Implement profile management view — Account Creation & Managemen... — 2 Front-end, Type: Enhancement, duplicate

2 Issues - 7 Story Points
In Progress
- FiestaMarket #35 Implement contact form table in database — 3 Back-end, Priority: High, Type: Enhancement
- FiestaMarket #47 Adjust user feedback system to use positive/negative rather than a review text — 4 Back-end, Priority: Mid, Type: Enhancement

1 Issue - 1 Story Point
Done
- FiestaMarket #61 Create database diagrams — 1 Priority: High, Status: Completed, Type: Maintenance

1 Issue - 0 Story Points
Project Point Tracking
- FiestaMarket #7 Project Point Tracking



CaretCake / FiestaMarket Private

https://fiestamarket.app

Manage topics

116 commits | 2 branches | 0 releases

Your recently pushed branches:

This branch is 2 commits ahead of master.

CaretCake Closes #35; Implemented ContactFormSubmission model — Latest commit 06d4d31 16 seconds ago

.idea	Code inspection and clean-up	2 days ago
api	Closes #35; Implemented ContactFormSubmission model	16 seconds ago
client	Small styling changes	2 hours ago
.gitignore	updates	2 days ago
package-lock.json	Restructured directories and adjusted naming conventions	6 days ago

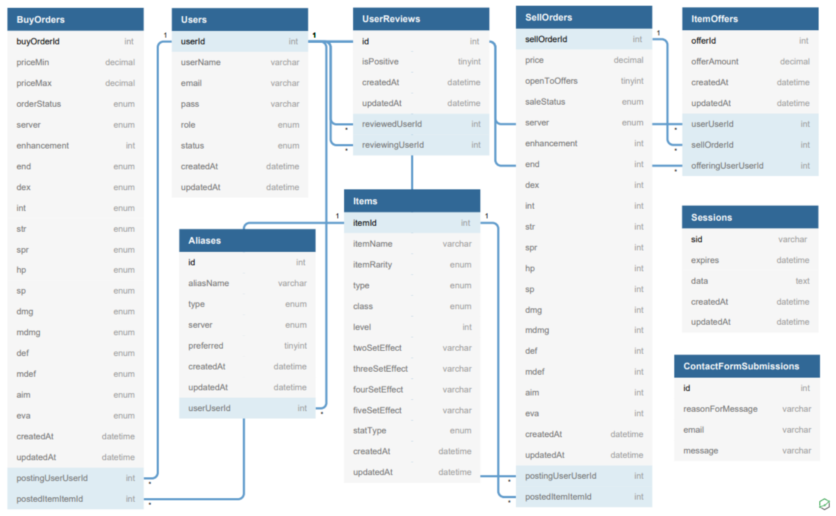
Add a README with an overview of your project.

Languages and Libraries

JavaScript	React - UI library
Node.js	React-Router - navigational components
Express.js	RXJS - event tracking
Express-Session	Axios - promise based HTTP client
CORS	Formik - form building
Passport.js - authentication middleware	Yup - form validation
Bcrypt - password hashing and salting	Sass - stylesheet language
MySQL - database	
Sequelize - MySQL ORM	

Database and DB Security and Validation

Taking from past course experience, I knew that database security and input validation is vital to protecting your application on all fronts, so I made sure to put time into researching my options.



To start out, I opted to use the Sequelize ORM. Implementing and learning Sequelize was one of the more difficult portions of the project, especially at first before I fully embraced it. I had created the Fiesta Market MySQL database myself when I first started trying to integrate Sequelize, which meant building the models to fit how I had originally set up the tables. After a while of struggling, I took the leap and opted to delete my database and allow Sequelize to create the tables for me based on the models and relationships that I had defined and refined by this point. Sequelize's model and relationship defining approach made it a breeze as I could focus more on the relationships and design of my tables, rather than the details of getting the tables and foreign keys just right and then trying to get my ORM to understand it.

Now, Sequelize helps avoid the problem at the heart of SQL injection attacks, which is unescaped strings. Sequelize escapes replacement strings. However, this isn't enough on its own. The extra step of object-relational mapping is a great first step, but these packages can still have vulnerabilities and ultimately at some point SQL is being composed and run against the database.

So, the next thing I considered for further protection was data validation both client and server side. Server-side validation can be implemented in a way that provides feedback to the user, which I did to ensure that the user would receive feedback in situations such as registering a new account where their username or email is taken. Implemented like this, server-side validation can be enough for input validation on client-side as well, but I didn't like the idea of that. Adding client-side validation as well would add another layer of validation and security, create a better user experience with more immediate feedback on invalid form input, and prevent the client from sending obviously invalid data to the server so I opted to do it on both sides.

Fortunately, my choice to use the Sequelize ORM paid off again as it not only allowed for creating custom validators for my data models but also contains built-in validators implemented by validator.js which cover individual attributes up to the entire model. This was perfect, as it allowed me to easily add validation for things like isEmail, isIn, len, and max/min while also implementing my own validation checks for things such as ensuring that a buy order's price range was a valid range all with my own custom error messages. Sequelize automatically performs these validation checks on every database query and throws a ValidationError which includes the type of error, the field it failed on, and the data instance it failed with.

On the client side, I made us of the Formik form and Yup validation libraries which I discuss more in the React section.

Authentication, Privileges, and Security

User authentication turned out to be quite a hassle, which was not a surprise to me. There was a lot of false starts on this aspect of the project, and ultimately it took me a while and a lot of reading and Google searching to determine how I wanted to approach it. As with much of web development, and software development in general, there's a lot of conflicting and confusing information and opinions out there that must be filtered through, compared, and weighed.

After looking into it, I ended up choosing to implement Passport.js, an authentication middleware for Node.js. Passport is made to be extremely easy to drop into Express-based web applications, which was exactly what I needed. Passport also allows you to create your own authentication strategy, which meant I was able to integrate it with my database and Sequelize ORM to track sessions and logins with my own site's usernames and passwords. Along with Passport, I made us of bcrypt. Bcrypt is vital here as it is the portion that hashes and salts passwords before storing them in the database for a new user, and for checking credentials on login. It's especially cool because first, it requires a salt by default, increasing the security of your user's right off the bat, and second due to bcrypt's design that makes it intentionally slow for password hashing, there's theoretically extra security provided against something like a dictionary attack.

API endpoints are protected through isAuthenticated and isAdmin middleware that use Passport to check the user's session ID.

Another portion of the security of the site involved setting up HTTPS. To start out, I had been running the API and react servers working on my own PC using localhost which meant I needed to make adjustments to get the live versions running and communicating. This process turned out to be one of the most time-consuming portions of the project, and in hindsight, I think I would have benefitted from doing it earlier on in the project. Fortunately, AWS S3 made handling the front-end hosting extremely easy, and by running it through CloudFront, AWS' CDN, I was able to use the AWS Certificate Manager service to generate a custom SSL certificate for my domain. From there, it was a matter of convincing my backend set-up to work and then work together with my client-side.

The process of setting up HTTPS for the Lightsail backend ended up being the most frustrating portion of the project. I toyed with the idea of running HTTPS in my backend Node.js application and even tried it at one point, only to find after some Google searches that that wasn't necessary. Essentially, since my front-end React client is served over HTTPS and on a .app TLD where that is required, all content and requests need to be HTTPS as well. Rather than change my backend code, the answer was Apache mod_proxy, and a LetsEncrypt cert. Apache mod_proxy is an optional module which implements a proxy/gateway for the Apache HTTP Server. Essentially, by making use of mod_proxy, Apache receives the requests from the Fiesta Market React client as HTTPS, forwards them along to my backend Node.js API as HTTP, and then upon receiving the HTTP result from the API, it forwards that back along to the client as HTTPS. This forwarding is all hidden from the client, keeping things running smoothly and happily in HTTPS. While the set-up process for this portion was frustrating, I learned some invaluable lessons about Apache, proxies, and troubleshooting issues with these things.

API

The Express application that runs the API for the backend of Fiesta Market was an extremely involved portion of the project. I learned a lot from the process of creating my own API backend from scratch and spent a lot of time trying to perfect and improve my endpoints and queries to make more sense and be more efficient, and I still would like to take time to go back and do more of that. There were so many moving parts with this project, I know that there are aspects of this portion of the project that I could improve.

That said, this portion of the project runs on Node.js and Express.js and it has the bulk of the work put into it.

React

The choice to use the React JavaScript library for the front-end was one that I had been set on quite early on. I wanted to make a single page application and React is one of the most popular libraries for that purpose. Learning React proved to be one of the most time-consuming portions of the development process which luckily a learn-as-I-go approach worked, but even after getting this far into this project I feel like I still have plenty to learn. Every time I moved on to working on a new portion of the React client, I would think of a better way to arrange components and modularize the code. Ultimately, I really enjoyed working with React and look forward to doing more with it.

On top of using vanilla React, I did also make use of some additional libraries which made the development process immensely easier. On the other hand, this also immensely increased the number of things I needed to learn and forced me to get a better understanding of some aspects of vanilla React as well.

The first library I picked up was react router. It is undoubtedly the library that had the most noticeable effect on the front-end application and my experience and enjoyment developing. React router allows you to make use of the browser URL to render different components depending on the URL, all while never actually changing the page in the browser. This meant I could maintain my single page application while still allowing bookmarkable URLs and adding logic to the rendering of components throughout the app. For example, I set the Header and Footer components to appear when the path is ‘/’, which causes them to render for every possible URL while the ContactForm component is set to render only when the path is ‘/contact.’

```
<Route path ='/' component={Header} />
<Route path='/contact' component={ContactForm} />
<Route path='/' component={Footer} />
```

The ease and simplicity of both the implementation of it and the logic behind it made for a great development experience and it made a huge difference in the structuring and readability of my React application.

The next set of libraries I opted to use were Formik and Yup. Formik is a small library which streamlines the process of building forms in React. It covers everything from tracking values, errors, visited fields, handling validation, and submission, but in a way that cuts out the extra steps while still like normal React forms - just less of a hassle. Yup is a simple library for object schema validation and it was a big deciding factor in my choice to use Formik. I wanted to make sure I covered input validation on the client side very thoroughly and given that much of Fiesta Market involves filling out forms with information to be added to the database, it seemed like the right call to take advantage of a couple of libraries that would streamline and standardize the process for me.

There were a few React libraries that I considered, and even started to implement, but ultimately opted not to use. The biggest of those was Redux. React Redux is an extremely popular library that allows you to create an overall state for your application, allowing you more freedom in how you set up your components and data flow. Without Redux, the data flow is limited to parent/child components and working around that can get extremely messy. Redux seems like an extremely powerful tool and I was excited to learn it and work with it, however, my main use for the application state was to track whether the user was logged in in order to display the correct components. After working with Redux for a day or so, I found that it was overkill for my use case, at least at that point in the project. So, I opted to use a library called rxjs and added a service to my app that would track login state which I could subscribe components to.

For communicating with the backend Node.js API, the React front-end makes use of Axios promise-based XMLHttpRequest client JavaScript library. Using Axios allowed writing more clear API calls, as well as better error handling on the client-side as well as use of ES6 promises.

Code Testing

I took a couple of different approaches for testing. For the API side, I used Postman to test API endpoints. For the React client, I used the Jest JavaScript testing framework. Learning how to test a front-end client and user interface, as well as working with an unfamiliar testing framework made for a challenge. That said, it was a great learning experience.

SASS and Browser Compatibility

I made use of SASS (syntactically awesome style sheets) to make the process of styling the site easier. SASS allows the use of variables in your style sheets, easier and more powerful logical nesting, loops, mixins, and more, which make the development and design process with CSS more like working with a full programming language. This made the process of styling and designing the site significantly easier. That said, browser compatibility is always a struggle with any website. Most of my time was spent working with and testing in Chrome, and then checking other browsers and compatibility and adjusting as needed. I found that this approach made the design process a lot more efficient rather than trying to get everything perfect in every browser from the start.

User Accounts, Account Management, and Registration

A registration form allows for creating a new account, at which point the user can then login and make use of the site. User accounts can edit their own user information, change their login status, post buy orders and sell orders for items.

Administrator Account

An administrator account can access admin-only pages and API endpoints, which allow for general site management. This includes viewing and managing users, viewing contact form and bug report submissions, and orders.

Admin only routes on the client-side are made invisible to users that do not have the ‘admin’ role but are also protected and will redirect a user without the ‘admin’ role upon any attempt to access it, in case they were to somehow find the link. The API endpoints which are designated for admin purposes are also protected through an isAdmin check on the API side and will reject any request from a user without proper permissions.

Site Interaction

Searching + Filtering

- Main Page Search - search items in the database and receive suggestions based on input. Selecting item brings up item page.
- Item Page - sort orders on item page by price/post time
- Secondary Main Page Search - filter most recently posted buy and sell orders by filters such as class, item type, etc.

Buy + Sell Orders

- Buy Order - post order from item page, set min and max for the price range, desired stat values, and server
- Sell Order - post listing from item page, set server, set a price, set if open to offers (other users can then place a bid on the item), set exact stats

Profile + Order Management

- Profile Management - change status, add and remove aliases
- Order Management - remove your posted buy/sell orders, edit buy/sell orders, and update the status of buy/sell orders

User Testing

I was fortunate to be able to recruit some trusted people to test the website for me after I got the live version up and running. This proved to be invaluable for multiple reasons. First, it was a great experience being able to talk to users and get feedback and ideas, especially for the interface design. It was also invaluable for catching small bugs and errors that I had not noticed myself and may not have. For example, I had set the form validation up for posting an item for sale in such a way that both the item's gold and gem price had to be greater than zero. This meant you could not post an item for, say, 15 gems (game currency) but would have to do something like 15 gems and 1 gold. This was reported to me by one of my test users, and even led me to be able to find and fix another issue with pricing in the database.

Retrospective: Major Problems, their Solutions, and Adjusting the Trajectory

Fortunately, most of my issues throughout the course of this project felt minor. Though, as with any large project, there were a few bumps in the road and general struggles along the way.

The biggest trajectory change I had from my original project proposal was opting not to use GraphQL. Once I started digging into working on the API, I found that GraphQL was going to be overkill for my project, as well as a lot for me to learn on top of getting familiar with writing API endpoints and learning Sequelize. This did in fact turn out to be the right call, as I feel like the difficulty of learning Sequelize and writing my own API was worth as many, if not more, points that I had originally allocated for GraphQL and I think adding that in as well would have been too much. I am happy with my choice to drop GraphQL in favor of Sequelize and not implementing further libraries for that. Overall, I think I generally covered most parts of the project well with my point estimates, but the one thing I would change would be adding points for creating the API and points for using Sequelize rather than GraphQL.

The most common issues were small things such as adjusting SQL queries and fixing SASS styling. There were multiple times over the course of the project where I needed to go back and adjust tables and relationships in the database, add an API endpoint, change an API endpoint, improve something as I figured out a better way to do it, etc.

The more I worked on the client-side, the more I found I needed to reconsider my approach to some aspects of the site. For example, I had originally created the UserReviews table with the intention of tracking the reviewed user's id, the reviewing user's id, and a block of text with their written feedback. However, as I started working on profile views and editing on the client-side, I realized that it made a lot more sense to simply track a simple positive or negative vote rather than just a written review.

Conclusion

I came into this project with the intention to learn as much as possible while making something to solve a real problem. Over the course of working on Fiesta Market, I learned how to use React as well as multiple React libraries, how to write a

Node.js API server from the ground up, and how to implement user authentication and securely track user accounts in your own database. In addition, I gained a lot from the experience of deploying my React client and API backend on AWS, including a better understanding of DNS, proxies, HTTPs, and how to troubleshoot server problems. I'm looking forward to continuing my work on the site and improving upon what I've done so far.