MUNI

Abstract
This is a web app that prototypes a secondary market and listing for municipal bonds that anyone can easily access and participate in. Users can look at municipal bond data and then follow specific issues and securities. It consists of a Meteor app and also includes a Python scraper.

The Meteor app supports both the frontend client and the backend server. The web app communicates to the backend by querying a GraphQL API. The backend server and stores data in a local MySQL database. A Stripe component was used to add a credit card to enable future payment processing.

The scraper is written in Python and scrapes through municipal bonds listed on EMMA. Data from EMMA was then used to generate securities, issuers, and trade data. This data was then used to populate the local MySQL database.

This project has challenged me to learn new skills and tackle new areas of software engineering. Before this project, I was primarily a frontend engineer who mainly worked on React components and had done little backend development. I had no exposure to web scraping and had little practical experience with databases. Through this project, I was challenged to build core parts of websites that I had never encountered. This included authentication to support login/sign-up functionality, relational schema design, working with 3rd party APIs such as Stripe, and being exposed to web-scraping and common web-scraping Python packages. In the past, I had been working on much larger codebases with already set-up tooling and infrastructure or projects set up by peers that also were already well developed. This was a valuable experience in experience what it's like to build an app from the ground up for the first time.
Motivation

Municipal bonds are considered a popular asset because they allow buyers to generate tax-free income and are considered to be low risk. Currently, municipal bonds can be purchased on the primary market—at the time of issue—or on the secondary markets, a market for trading bonds after the bond has already been issued on the primary market.

When you purchase a bond on the primary market, there are no fees or markups. However, it tends to be difficult for a retail investor to buy new issue municipal bonds. Generally, retail investors are individual investors who buy and sell securities for their personal account, and not on behalf of another company or organization. (as opposed to institutional investors). Non high-net worth individuals tend to be unable to easily purchase a bond during this process.

The goal of this app would be to increase access to the municipal bond primary market for all individuals, whether through exposing more users to municipal bond offers or making the secondary market more widely available.

Functionality

The app allows one to make a user account and login. Users can then view previous municipal bonds and issuers that have been already loaded into the databases. For each issuer, a user can view the issuer's securities. And for each security, a user can view that security's previous trades. Users can choose to “follow” certain issues or securities. This information will be stored in the database and will be used to customize the user’s page. The user can also add a Stripe credit card for potential future buys and sales once that functionality exists.
Technical Details
Backend, client and scraper code available at https://github.com/klairetan/cpsc490

Database Schema
For this project, I chose to use MySQL, a relational database. Conceptually, the schema contains six table: users, issuers, securities, trades, issuerFollowers, securityFollowers.
The issuerFollowers and securityFollowers tables are used to store the issuers or securities that a user has decided to "follow" when customizing their homepage. Trades are used to represent trades between users of a security.

**API Design**

The frontend client communicates with the backend server via a GraphQL API. GraphQL serves as a query language for APIs and can be seen as a data layer between the client and the backend. A core part of GraphQL is defining a schema including a type system for the data that can be queries. Data is grouped into typed objects with queryable fields on them.

In the GraphQL schema, I have defined types **Issuer, Security, User, UserInput, AuthPayload, Trade**. An example of this has been included below.

```graphql
input UserInput {
  firstName: String!
  lastName: String!
  email: String!
  password: String!
  token: String
}

type AuthPayload {
  token: String!
  user: User!
}

type Trade {
  id: ID!
  buyer: User!
  seller: User!
  tradeDate: String!
  amount: Int!
  yield: Float!
  securityCUSIP: String!
  security: Security!
}

type User {
  id: ID!
  firstName: String!
  lastName: String!
  fullCountry: String!
  email: String!
  password: String
  buys: [Trade]!
  sales: [Trade]!
  followedIssuers: [Issuer]!
  followedSecurities: [Security]!
}
```
Part of easing the development process is also exposing a Graphiql, a visual interface for GraphQL that enables you to easily review your GraphQL schema and test queries against the backend server.

Requests to the API also need to include a valid access token if querying fields that require authentication. This is included in the request as a header using the bearer authentication schema.

**App Architecture, Technologies**

Because a lot of the server setup and development was new to me, app architecture was extremely challenging. Often times, the focus was on figuring out just how to build something rather than the best way to build something.

I chose to use Meteor JS as my framework because I hoped it would decrease time dealing with tooling and just setting up the app development environment, as Meteor comes with configuration for webpack and Babel and app bundling already taken care of. Meteor also bundles the frontend and the backend together into one codebase. Most of the code—backend and frontend—lives within the imports folder.

**Frontend**

Even though some fields on the graphql API can be queried without authentication, we require authentication on all routes in the app. Authentication is checked using a meQuery to the graphql API that returns the user if the request includes a valid token. Apollo Client is used for handling making graphql requests to the server. Apollo Client also comes with a local cache built in. This enables some level of offline support. If the app goes offline and the user makes a query that has already been made before, such as for all Issuers with a certain Id, we can use ApolloClient to fetch that information from the cache instead of making a network request.
Getting Data

To get municipal bond data, I wrote a Python scraper to scrape ‘https://emma.msrb.org/Search/Search.aspx?hlt=search’, a publicly accessible database of all past municipal bonds. I ran into several obstacles while doing so. First, the page required submission of a disclaimer form that actually prohibited scraping of data. Second, searches are done by filling out a form, and moreover, search results wouldn’t load if the search terms were too general. For example, only including ‘San Francisco’ for an issuer name.

Eventually, we didn’t use this data directly but instead used it to auto-generate municipal bond data of our own which I then populated my databases with using mySql files.

Future

While the current app allowed me to experience building an app from scratch for the first time, it’s clear to me that is a very simple prototype compared to the municipal bond app that users and investors would like to use. However, this is a product I would like to continue to work on, and I think there are three important areas to focus on in the future.

Data

On initial design of this project, it wasn’t obvious to me that the municipal bond dataset would be this difficult to acquire. A lot of time was sunk into figuring out what data I needed, different scraping alternatives, and how to reformat the data into the form I wanted for the table. I now believe that acquiring this data could be a whole project on its own and really should be given better emphasis. A valuable direction I think would be working with websites such as EMMA to build a developer friendly API for fetching data on municipal securities.

Scalability

Currently, the codebase lacks unit testing and integration testing. This will make it difficult to continue to add features without breaking other working parts of the app. In addition, error logging and handling could be a lot better, especially in alerting the users of backend errors. Error logging will also help once more users interface with the app and uncover edge cases that weren’t properly handled before.

Product

I think there is more basic product value that can be added in even if data is lacking. For example, perhaps a newsfeed on securities using an existing API such as Google News API to provide the user with more general content in the app.