Coopmo: A Friend-to-Friend Payments and Social Platform

Evan Bubniak, Do Hyung Kwon, Minh-Thai Nguyen, Shyam Paidipati
ECE 366: Software Engineering
Ethan Lusterman
https://github.com/thedavekwon/coopmo

October 8, 2020

Contents

1 System Architecture

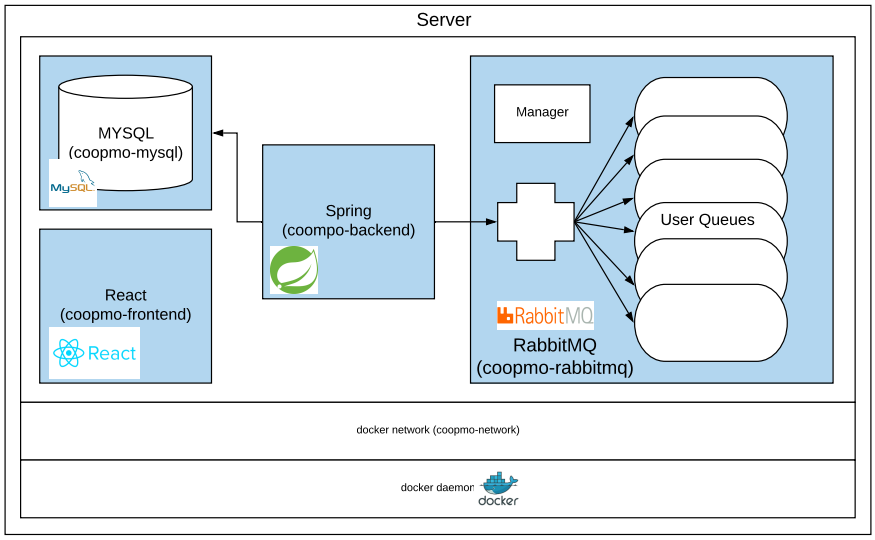

Figure 1: System Architecture Overview

Coopmo employs a general client/server architecture. Our server is entirely deployed in the docker container for scalability and ease of migration. It uses docker-compose to create docker network and containers that communicate each other and with the web client. Spring, React, MySQL, and RabbitMQ are the four main containers. We used Spring for its backend handling http requests and websocket connection, MySQL for persistance database, React for frontend, and RabbitMQ for message queuing. The general workflow is that user access web server through browser and retrieves web app using react server. react server communicates with Spring for backend and RabbitMQ for push notification. Websocket is connected to following RabbitMQ queue waiting for push notification as soon as user logs into the service through backend authentication. We use Simple (or Streaming) Text Orientated Messaging Protocol (STOMP) for communicating between backend and RabbitMQ (queue notifications) and between client and RabbitMQ (push notification). All requests and data are fetched through backend server.

2 Frontend Overview

We built the front-end in ReactJS, with help from Figma for front-end design modelling, Bootstrap for creating a responsive and clean interface, and Redux for state storage and persistence. For our frontend, we chose to make a single-page application because we didn't think our app had enough functionality to warrant a multi-page application and because a single-page application offered different problems that we wanted to tackle.

2.1 Figma

We initially used Figma as a tool for prototyping the frontend design. We initially used Figma's API to generate React components from our Figma prototype; however, this proved to cause more trouble rather than help. Instead, we decided to build our components from scratch making our components much more customizable.

2.2 Bootstrap

In our second checkpoint, we used a primarily in-house created component. Although this came with a lot of customizability, it took much longer to implement, most of the time didn't look as nice, and it would've taken longer to make it more interactive. This led us to use React-Bootstrap components as well as other libraries to implement our components. This made our application much more interactive and generally prettier.

2.3 Redux

One of the main reasons of using Redux is because it makes our organization much cleaner and makes it easier to share states between components. With Redux in conjuction with notifications, we were able to make a pseudo-live application updating components when relevant notifications appear (ie. updating balance when a new payment is made). Redux also allowed us to save a persistent state in the browser's local storage, allowing us to keep a user logged in, to keep the user's current page upon refresh, and to maintain the username in state to check if a user has already liked transactions on the feed.

2.4 Additional Features

A new feature that we added to the frontend is notifications. Through RabbitMQ we are able to send notifications to application by opening a web socket to receive them. Users are able to receive notifications in real time and navigate to the corresponding page through the notification. Notifications that are sent went the user is not logged in are queued and arrive when they log in. We also added profile picture support to our application. The user can now change their profile picture and see other people's profile picture. The feed also supports infinite scrolling so more entries are loaded as you scroll down. Finally, some quality of life changes were made for the user.

2.5 User Interface

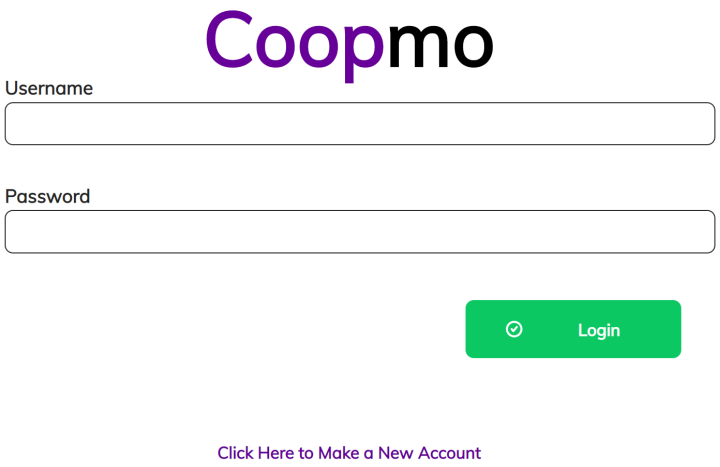

Figure 2: Landing page of Coopmo, prompting the user to log in or create an account.

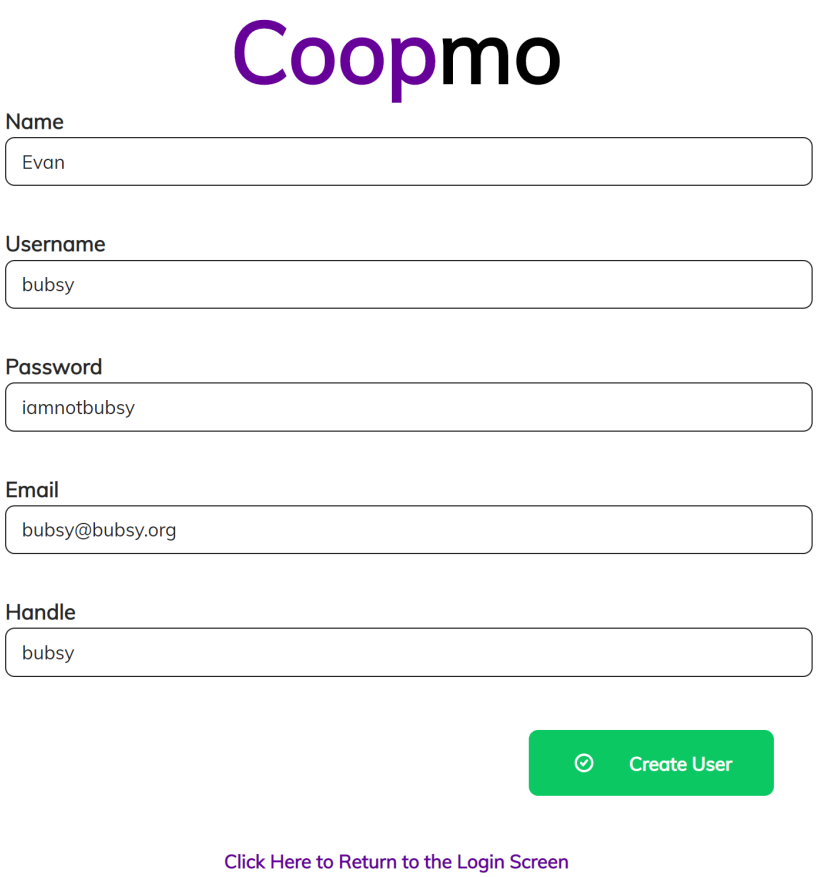

Figure 3: Create user page.

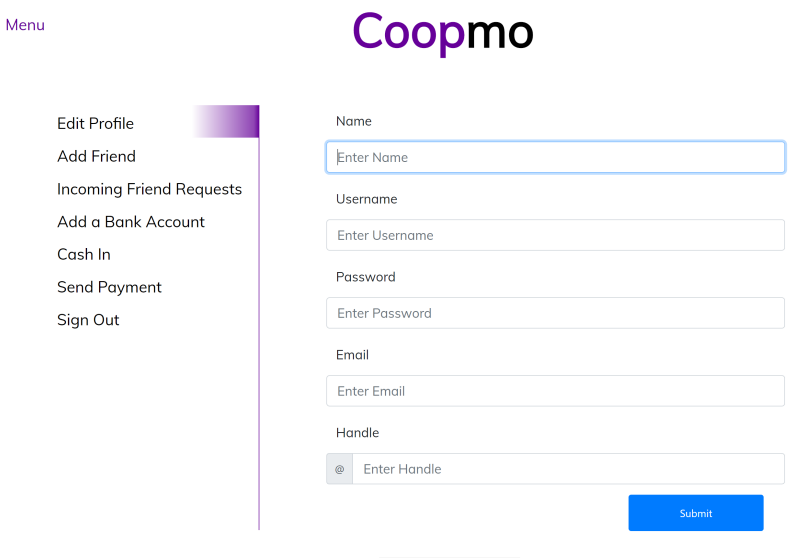

Figure 4: Edit profile page.

3 Backend Overview

3.1 Security

3.1.1 Spring Security Overview

Spring Security allows for flexible customization of security to allow for many variations to authorization and authentication. Spring Security uses a system of configuration files and filters to implement its version of security. Each configuration file details how each *Bean* defined for security handles a particular authentication or authorization issue. Filters catch every request sent to the server and does layered check based on the role of each filter. Details about configuration files and filters will be explained in future sections.

3.1.2 Authentication

Authentication is done with JPA and a back end MySQL database. The first step of authentication in Spring is to determine which *AuthenticationManager* it will use to verify input. For Coopmo, the input was credentials, so a credential based authentication method was implemented. The *AuthenticationManager* then passes on the Authentication request to on of the implemented methods. These methods will either return a Authentication object with a flag indicating that is has been verified, an *AuthenticationException* for an invalid principal, or a null if it can not process the request. For Coopmo, the credentials are passed in an Authentication object to the *MyUserDetailsService* which will load the corresponding credentials of the username passed to it. A check is done to see if the passed in credentials match the ones loaded from the back end database. If it does, the Authentication object is returned with a flag saying it has been Authorized. The principal is now set as the *SecurityContextHolder*. This is used later to prevent repeated authorization requests for a logged in user.

3.1.3 Authorization

Authorization is handle by the *AccessDecisionManager*, which decides whether a request has the authority to access a certain endpoint. Most Spring framework implementations uses the default behavior as it is robust and utilizes a simple implementation. Most of the flexibility of authorization comes from configuring which endpoints in the API require authentication and changing filters. Spring Security uses Servlet filters to catch HTTP request as they come in to the server. At most one servlet can handle one request, but the filters are chained so that a single filter can decide whether the request continues down the chain or if it will handle the request itself. Configuring these filters requires the modification of the *FilterChainProxy* which delegates which filters take which requests. In a configuration file, endpoints can be configured for authentication by matching the path in the header of the HTTP request to a condition that need to be passed in order for access to be allowed. In Coopmo, every endpoint requires authentication with the exception of the */user/createUser* and */login* endpoints.

3.1.4 Sessions

Sessions are largely handled by the default behavior of Spring Security. A *sessionID* is passed to the client after they successfully login. this cookie is stored in the clients browser and sent on every request from the client. This prevents the client from needing to log in on every subsequent request. Configuration of sessions is done in the *application.properties* file where characteristics such as the session expiry time and the database schema used to store the sessionIDs and the corresponding principal.

3.1.5 JWT Tokens

In order to use JWT tokens, Spring Security must be configured to be stateless so that the framework does not create sessions automatically for each client. To use JWT tokens, on a successful login, a token is generated for that user based on their unique userID. The token is then passed back to the client where it can be sent back to authenticate the identity of the client and allow access to endpoints. Several files must be create in order to use tokens such as *AuthenticationRequest* and *AuthenticationResponse* in order to properly load in the user credentials as a *Usernameand PasswordToken* for authentication and to send back a response containing a JWT. In addition, an extra filter must be added to the filter chain to catch every HTTP request to process the JWT sent with it. This filter checks to see if the JWT is valid and corresponds to the correct user that the JWT was made for. This can be done in a configuration file that uses a *Bean* to add a filter for this purpose. Th last file needed is to create JWT utility file that creates the template for every JWT and adds critical fields such as token expiration and the creation of the secret key encoded at the end of every JWT.

4 Team Dynamic/Challenges

After midpoint check one, the team split into two groups, the front end and back end. The front end group worked on using React and writing scripts to send request to the server while the back end group worked on security and other features such as docker build and cleaning up back end code. The two groups met with each other when required, such as communicating which endpoints to send certain request to and to determine in what form to send user input. Within each group, members

talked with on another frequently to tackle problems together or to relay ideas that impacted the architecture of the project. In addition to these meetings, the team would also meet as a whole is discuss and delegate goals to reach by certain dates.

On the front end, designing the user interface and getting the front and back ends to interoperate smoothly was a challenge. While constructing the feed, we needed to make separate API calls to the backend in order to get the most recent transactions - one call each for private transactions (which include transfers to and from bank accounts), friend transactions, and public transactions respectively. In addition, we wanted to implement infinite scrolling (or as infinite as the transaction history on the database would allow), making backend calls for new transactions as the user scrolled through their feed.

Designing the user interface proved tricky as well; we had started out with Figma, which generated a lot of boilerplate React and CSS that initially looked good but proved *extremely* difficult to manipulate without messing up the appearance or alignment of each object. (Almost every object was placed with absolute positioning and CSS styles were applied hard-positioning each object relative to the corners.) This was circumvented by carefully refactoring the boilerplate into flexboxes for positioning elements across the screen, which conveniently also made it easy to add and remove white space as needed.

The back end had a few challenges, including dealing with Cross-Origin-Resource-Sharing permissions and modifying features to adhere to the current authentication and authorization features. An example is when incorporating the notification system into Coopmo, the STOMP client in JS does not allow appending additional headers for authroization for JWT. Another issue was looking into work around of CORS as Spring documentation did not explain how to configure Spring security to adapt to the localhost using two separate ports to communicate. These problems were later resolved after extensive work in configuring the appropriate *Beans* in Spring to allow the intended behavior.

5 Future Direction

5.1 RabbitMQ

Generating queue for each user is not ideal for scalability. We should use RabbitMQ with MySQL for queuing message properly and more persistent queuing. SockJS and STOMP client does not support authentication through JWT token, so we need some work around for finding session for user.

5.2 Security

Future steps for the backend include incorporating both sessions and tokens for security. Sessions will keep track of a user and when they login while tokens will handle requests to authenticated endpoints. Token can also be split into refresh tokens and access tokens, each with a different purpose. Access tokens act as the key for the protected resource since they contain all of the information needed to authenticate the client. Refresh tokens contain the information that allows a client to get more access tokens.

5.3 Logging

We currently do not employ any logging tools to log information and errors. We believe adding a logging tool for automatically log will help us easily debug further issues.

5.4 Testing

We currently have only one test for the user service layer. We believe we need more testing in different layers as well as integration testing to ensure the quality of our codebase.