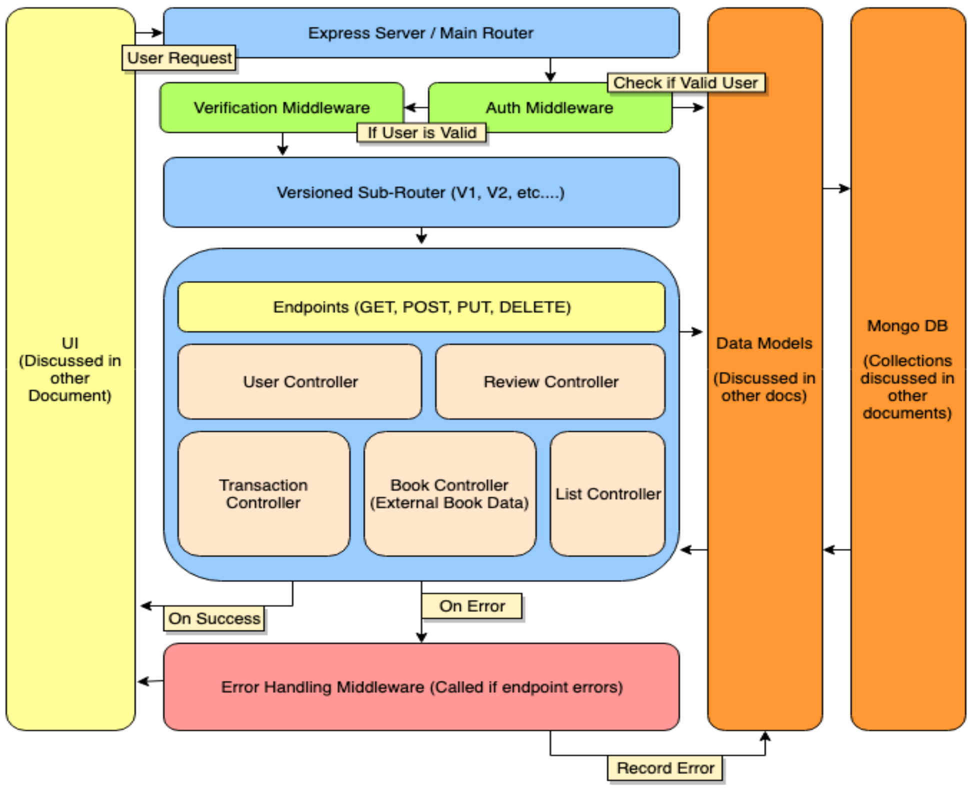
The Company (Curtis & Co.)

Texticles

2nd October 2019

OVERVIEW

The general structure of this application is shown below where each of the 3 major parts are shown. The application follows a standards Module-View-Controller architecture where the view must go through the controller in order to access any data, which is all located in the database. This allows for the system to have proper encapsulation where each part of the system serves its own purpose without affecting the other parts The decision to choose an MVC architecture was made because it will allow us to make more informed decisions about what models should exist and how the models should function or interact with each other and what type of modifications can be applied to the database.


UI (Discussed in other Document)
Express Server / Main Router
User Request
Check if Valid User
Verification Middleware
Auth Middleware
If User is Valid
Versioned Sub-Router (V1, V2, etc....)
Endpoints (GET, POST, PUT, DELETE)
User Controller
Review Controller
Transaction Controller
Book Controller (External Book Data)
List Controller
On Success
On Error
Error Handling Middleware (Called if endpoint errors)
Record Error
Data Models (Discussed in other docs)
Mongo DB (Collections discussed in other documents)

System Specifications

Deployment

The application will be deployed to fit the required criteria using a Docker container. This is the easiest way to ensure that the application can be deployed on any system while being able to provide the same runtime environment that you would expect on a given system. The container will use the basic “node-alpine” base environment. Furthermore, The system will be built and tested using Travis CI. Eventually, Travis will be configured to construct the container as well. The MongoDB instance will be held in the MongoDB official Docker container which can then be deployed alongside the application using Docker compose.

Testing

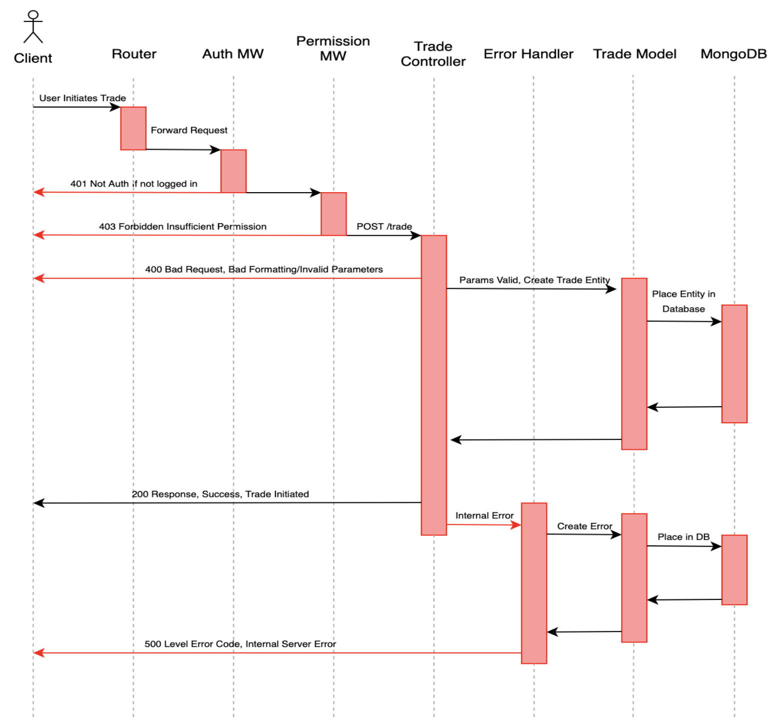
The testing platform being used is **Jest** because it is well supported (Maintained by Facebook) and it provides a simple API for managing a variety of tests and assertions. In order to fulfil the ASR for *testability* that was described in **Issue #14** we will be using the standard request library along with jest to make requests to the server. For testing, the configuration can be set to use a mock empty database through environment variables. This will prevent the program from filling the staging or production databases with unnecessary data slowing down the testing process. With this setup up and the ability to call each endpoint from the testing module, we should be able to reach the 70% test coverage that was projected.

Serviceability/Supportability (Issue #9)

In order to maintain a good idea of the reasons the application may be crashing or returning unexpected errors, an error catching middleware that can be called through the express “Next” function that is passed into endpoints as an argument. Any errors that occur will be caught and fed through this middleware and logged into a table in MongoDB. Implementing this functionality as a middleware allows the implementation to concern itself very little with the actual endpoints it will be called from since it will all be logged in the case of an error. It is also easy to chain multiple errors handlers together if it became necessary in the future.

Modularity (Issue #10)

To allow for our application to be modular, we have employed a Model-View-Controller design pattern to ensure that it is possible to change or remove models without affecting other components in the system. Our system includes various middlewares to help keep dependencies to a minimum within each component. After a user performs some sort of action such as initiating a trade or accepting one, the request will be sent through the router to the authentication middleware. If everything is going smoothly, the request will be passed along to the permissions middleware which will determine whether or not the user has the right credentials to be making a trade request. After confirming that the user has the correct credentials to be initiating a trade, a post request would be made to the trade controller which would govern whether or not the request has valid input parameters that match those of the trade models. If the parameters do match, then the request would be placed into the database and then and a trade will be initiated. Otherwise, if an internal error does occur before the trade is initiated, it will be passed through an error handling middleware which will then log the error into a table within the database and return a 500 level “Internal Server Error” code. The trade initiation is an example of a type or request that might occur but the majority of the requests would follow the exact same path to the database with the exception of the login, sign-out, signup, and public endpoints which will pass the authentication and permissions middlewares.


Client
Router
Auth MW
Permission MW
Trade Controller
Error Handler
Trade Model
MongoDB
User Initiates Trade
Forward Request
401 Not Auth if not logged in
403 Forbidden Insufficient Permission
POST /trade
400 Bad Request, Bad Formatting/Invalid Parameters
Params Valid, Create Trade Entity
Place Entity in Database
200 Response, Success, Trade Initiated
Internal Error
Create Error
Place in DB
500 Level Error Code, Internal Server Error

Performance (Issue #11)

In terms of keeping track of how well our application and its components are performing, we have decided on two different metrics. The first metric involves observing how many errors per minute our application is producing. These errors can be anything that is not within the 200 level error code family. We will keep track of these error codes by inserting them into the database. The other metric that we were planning to use is latency. Our goal is to be able to track how long it takes for users to accomplish certain tasks and also how long it takes for interactions between different components to complete. In order to keep the latency to a minimum we will try to keep the database interactions to a minimum and fetch the data we need in single requests to MongoDB. Additionally, to keep the query times to a minimum, we will index the columns that are being used often so queries will complete faster.

Security (Issue #12)

There are two major points to our security system. The first will be that after the user logs in, they will be provided with a token (JWT) token to store as a cookie in the browser and use this to make authorized requests. This is being used because it is a simple way to protect our endpoints and even though it is not a requirement it is a necessity for most real world applications. Additionally, to fulfill our ASR of locking users out after attempting to log in multiple times with incorrect passwords, we will store a counter in the database under the users name allowing the authorization firmware to lock out the account after a certain number of failed attempts.

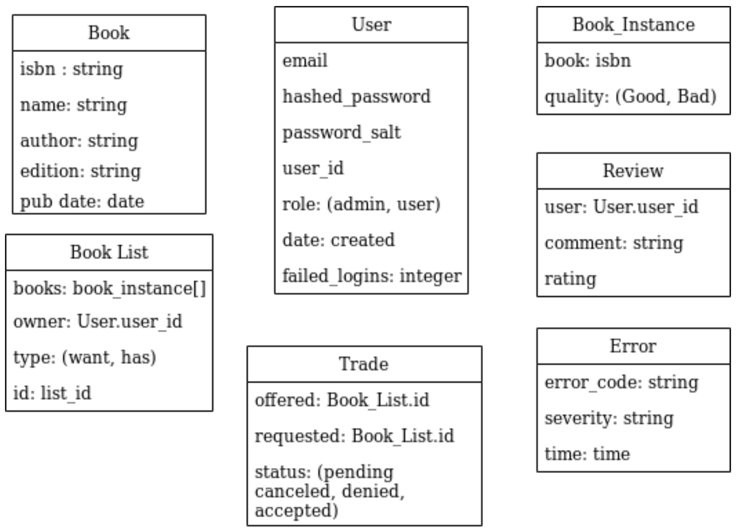
Roles

In order to delegate permissions to different users, users will have a role associated with their account. Regular users will have a “Standard” role and they are able to access and delete any data that they have created with their user ID. On the other hand there is an Administrator role which can be used to delete or create Users, Transactions, and override anything that other users can do. With a simple role based permission system we can easily delegate grouped permissions to the two user sets without spending large amounts of time implementing permissions systems. This will allow us to achieve the user stories set out in **Issue #6** and **Issue #7**. The implementation of the permission checking will be handled in a separate middleware after the user is authenticated. This way any changes in the future to the permission system can be handled separately without having to modify any of the endpoints or models.

User Interface

The user interface will be constructed using the Web Component technology. This is a standard for creating components in UI applications that are self contained. These components extend Typescript classes and also help encapsulate CSS into something called the "Shadow DOM" which will prevent different ID tags and styles from conflicting. This choice was made because it allows for us to still maintain clean and readable code and allows the complexity grow at a more manageable rate essentially allowing us to gain some of the benefits of frameworks without using them. In order to achieve our ASR of "Discoverability" (**Issue #8)** we will ensure that there is a simple menu on the top or side of the page at all times that allows users to access any other page. This will allow for easy navigation and make it simple for users to achieve their goal quicker. There will be a main page that allows users to access their trades, view recommended trades, as well as a place for them to edit the books on their wish lists. There is a search bar at the top of the page for users to do a more indepth search of the content they are looking for. The application will use client-side rendering due to its simplicity.

Models/Data Type Structure



All of the models held by the database will be reflected as classes in the typescript code. There will need to be a layer that loads each entity from the database and populates an object of that type to be used by the server. Forcing the data from a json object into a typescript class will allow us to leverage typescript static typing abilities to eliminate bugs.

Book - The definition of a book, not a specific instance of one that someone wishes to trade. This will be information such as the title, edition, year, etc. Having this type of book will reduce the amount of redundant information in the database.

Book Instance - A specific instance of a book. References a book (The actual book information) in addition to information such as the owner and the quality of the book.

Book List - A list of books that a user either wants or has.

Trade - Tracks the books wanted and offered in a trade, and the status of the trade.

User - All data related to a user.

Review - Left on a users profile after a trade was done.

Error - A table for us to report and track errors generated by the server.

Endpoints and Validation

As stated previously we wish to take full advantage of the static typing supplied by typescript. To do this we need to enforce a strict gateway between the server and clients by carefully checking the types of all the data returned. This way we can operate in confidence knowing the request data is in the correct form.

The validation will operate by sending each json value through a validation function. If any of the validation functions fail, an error is sent back to the client which lists the fields that were invalid. The data must be valid before the server will carry out a request.

Stretch Goals

One feature we talked about but decided was beyond our requirements was a permission system for our endpoints that would allow more fine grained control. It would essentially consists of a list of endpoints that each user would be allowed to access. If time permits we may add this feature, however it is low priority.