SENG 350

Detailed Design

Deliverable #2 <Team 3-1>

Isaac C <V00841434>

Jamie S <V00868359>

Marina D <V00844643>

Braydon B <V00741364>

Sunday, October 13th, 2019

Table of Contents

Purpose

The following paper outlines the architectural design of the TindArt system. It contains a dynamic view explaining the runtime flow of the system, a static view explaining the class structures, and a record of architectural decisions.

Overview

TindArt provides a platform for artists to show their artwork to members of the local community. Users of TindArt who desire to sell their art are able to post an image of their artwork with an associated price and description. These artworks are then visible to users of TindArt in the same geographical location by browsing the home page or viewing them on the artist's profile page. The home page for each user shows a carousel of artwork photos, chosen algorithmically by TindArt, from the user's location. Locations are determined by the user; each user specifies the location they are in (for example, "Victoria, B.C, Canada"), and can see artwork from other users in the same location. A user can "like" an artwork, which allows them to save the artwork for later; view the artist's profile page, including their other artwork; and contact the artist for arranging a purchase.

Dynamic View

The Dynamic View section follows a "happy path" use case through TindArt, as a first-time user who uploads artwork to sell. This "happy path" is shown visually through a sequence diagram in Appendix A. User stories from Milestone 1 are used to define the path through the application and provide concrete use cases to which quality attributes scenarios are applied. Two quality attributes in particular, usability and security, pertain to quality attribute scenarios existing in the dynamic portion of the system.

First Time User

A first time user of the application must create an account before they can browse or upload art. To create an account, the user enters a registration portal which prompts them for the following information using a traditional form:

- Username [Alphanumeric, Unique]
- Password
- Email address
- Bio [500 characters or fewer]
- Location [Region and Country]
- Profile Picture [Optional]

- Phone Number [Optional]

This data is validated by the web server before a new account is created in the database under the users table. The account is given a universally unique identifier (UUID) for ease of lookup.

A session cookie is generated on the web server and sent back to the user so that the user is logged in as the newly created account. The user is then redirected to the home page, where a pop-up prompts them with the option to upload images of their artwork. If the user is an artist wanting to sell their art, they navigate to the artwork upload portal via the pop-up.

Artwork Upload Portal

The artwork upload portal allows a user to upload a new artwork to their account. Each artwork on TindArt has the following attributes, which are provided by the artist in a form on the artwork upload portal:

- Image
- Price
- Dimensions
- Title
- Description

Each artwork also has a UUID generated by the web server. Each uploaded artwork is displayed on the artist's account, and is sometimes displayed on the carousel on the home page of other users who share the same location.

When a user creates a new artwork by submitting the artwork upload form, the web server will validate the entered data, including the uploaded image. When the validation is successfully completed, the web server saves the uploaded image to the file server with the artwork's UUID as the file name, and a new artwork is added to the user's account in the database, with fields for price, dimensions, title, description, and the URL for the saved image on the file server.

Home Page

The home page of the application is where users browse local artwork. When a user navigates to the home page, a query to the database is executed which requests a paginated list of artworks from other accounts who share the same location as the browsing user. The home page is sent to the user with the queried data. The client then requests the URLs for the queried artworks' images, which are available via the file server.

Liking an Artwork

On a user's home page or an artist's account page, there is an option to "like" each artwork. When a user "likes" an artwork, a request is sent to the web server with the artist's UUID, artwork's UUID, and the user's UUID, which adds the artwork to the user's list of "liked" artworks. A user cannot "like" their own artwork.

Security

The security of this application is important so that each user's personal information is not accessible by other users or the public. To ensure that TindArt is a secure website, all user input is sanitized and object models is used to mitigate NoSQL injection attacks. Additionally, each time a user logs in, they are given a new session cookie to reduce cross-site scripting attacks.

Only the web server has write access to the file server to avoid artwork from being inadvertently deleted from or inserted into the file server, and to simplify the design of the file server. Due to the public nature of TindArt, no read security exists because all art pieces are available to view by all users.

Usability

To increase the speed at which the users can access the artwork while browsing or looking through their own "liked" artworks, a client-side pre-fetch approach is implemented. The pre-fetch is implemented by immediately getting and loading 10 artworks on the client-side when the home page is loaded. When the user has browsed through 5 of these in the home page carousel, an additional 10 artworks are requested from the web server, their images fetched from the file server, and loaded on the client-side to display to the user. This process is repeated each time 5 of the 10 new artworks are viewed.

Static View

A static view of TindArt is shown in the class diagram in Appendix B. This view gives an overview of the different components of the system by showing the typescript classes for each. These include the routes that use Pug to render the front-end pages to show to the users, the model classes that provide data to the routes, the system classes, and the relationships between each of these. We chose not to show details of our external libraries, Express and MongoDB, to avoid cluttering the diagram without adding valuable information.

The design of TindArt's class structure was informed by 5 quality attributes: testability, modifiability, scalability, front-end portability, and back-end portability. Here we discuss the effects of our architecture on each of these.

Testability

Our architecture has been designed to follow object-oriented best practices, and in particular, our functionality is well-encapsulated, allowing for straightforward testing. For example, our Database class abstracts away any interactions directly with a database, meaning that when testing the model classes which rely on the Database class, we could easily create a mock of the Database class which does not rely on an existing mongo database, but still allows the model class to be fully tested. This is also seen in the distinction between route and model classes, allowing routes to be tested with mocks of the model classes. Our code will of course have unit and integration tests with Travis CI, which will prove this testability.

Modifiability

In our design, we have a very small amount of coupling between classes. Only the Router and Database classes are connected directly to more than 3 other classes (not counting Express classes), meaning that changing any class other than those two classes will likely require changes to fewer than 3 other classes. This means that modifications to existing functionality will be straightforward in most cases.

Additionally, because the structure of the router, route, model, and database classes are consistent, clear, and sufficiently decoupled, adding new features via new routes and models will be conceptually and programmatically simple, giving the system a high degree of modifiability.

Scalability

By separating the image server from the web server, we have the ability to scale the two systems independently. For example, if the file server is being stressed with many image requests, it can be provisioned with more resources without affecting the web server.

Also, by abstracting database connection and access through a Database class rather than having each model directly use mongodb's functions, we allow for the future possibility of increasing the scale of the project through using a database system that is more optimized for the queries and use cases that we find are most frequently used.

Additionally, by using a generic filesystem as our image store, we are able to make that filesystem virtual - via docker volumes - in such a way that changing the storage medium

(for example, by moving to a larger or more performant system) would require very few, if any, changes to the server code.

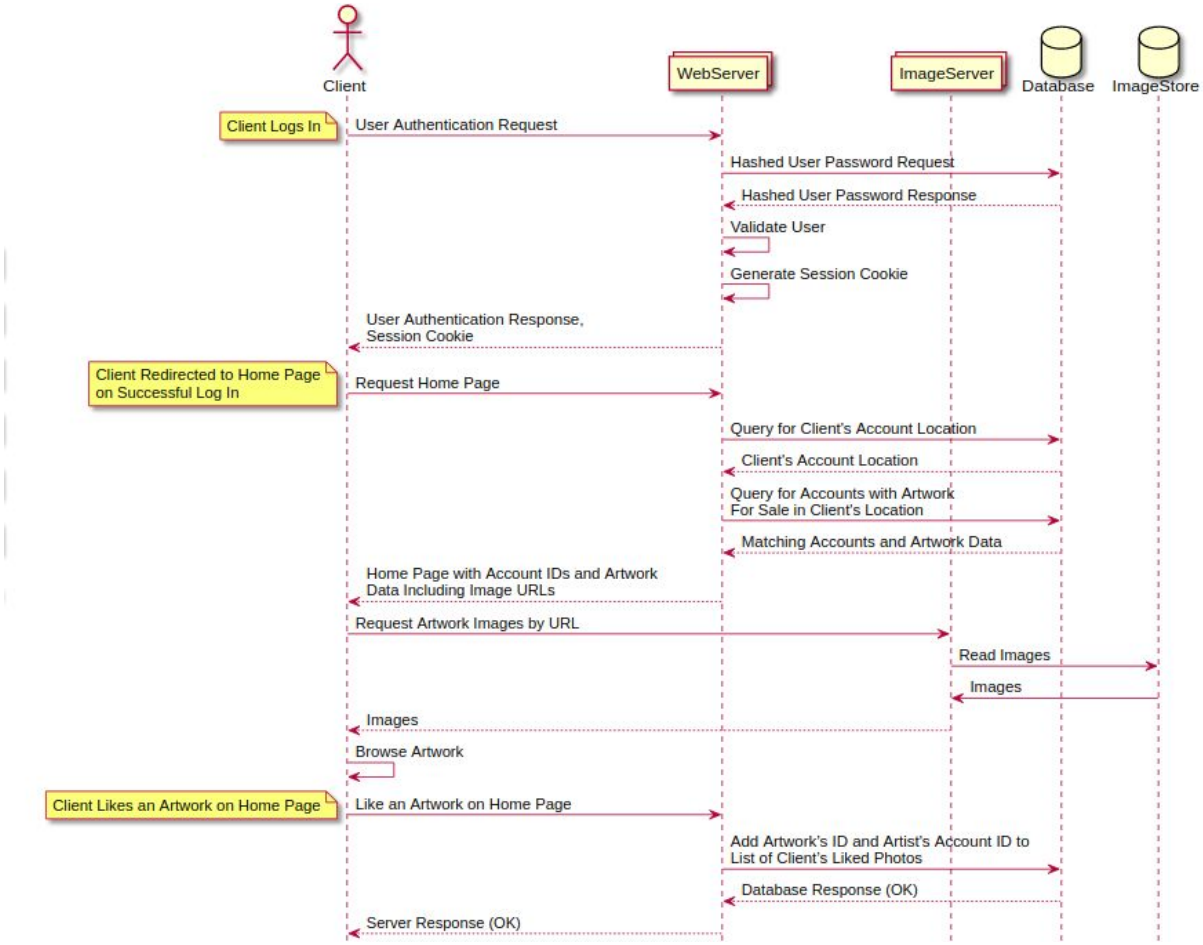
Front-End Portability

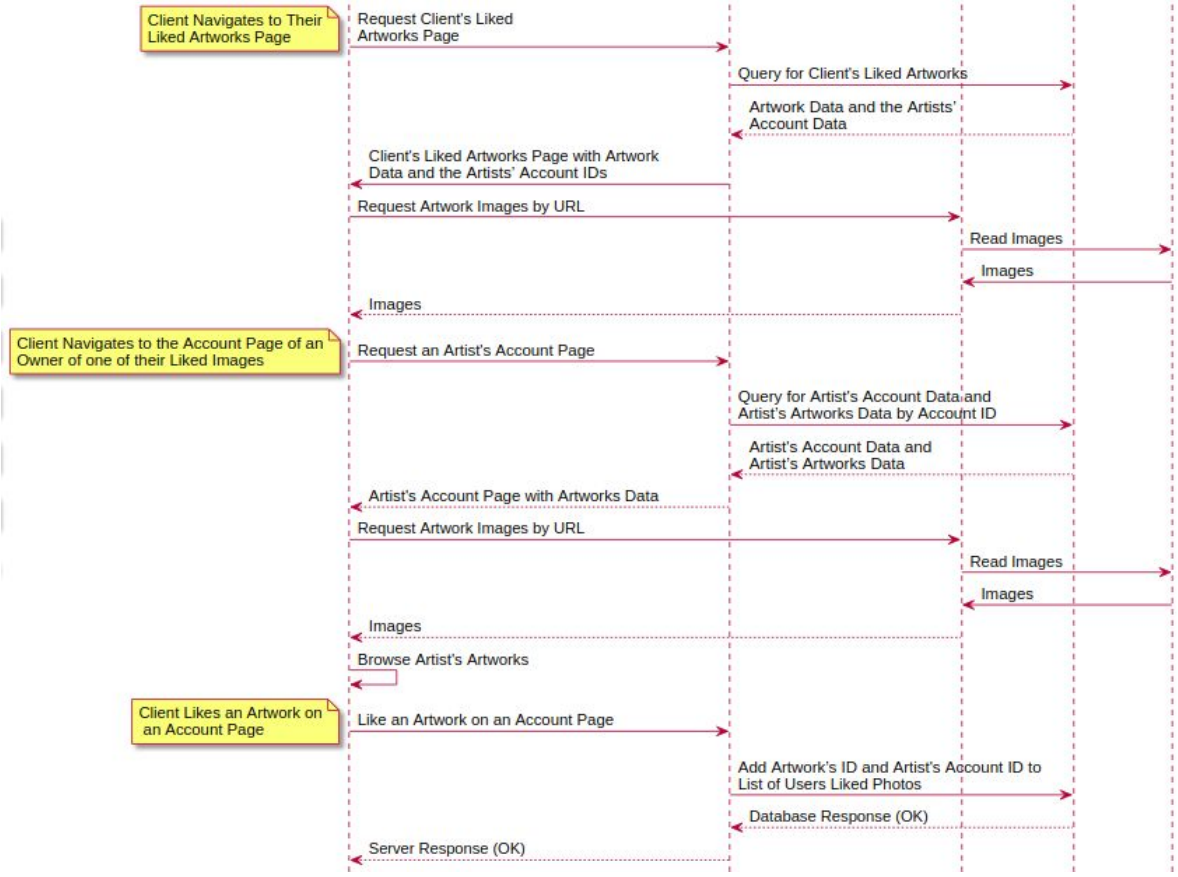
The application front-end will be generated and rendered on the server-side and sent to the client as HTML and JavaScript. This reduces reliance on the users' front-end software, and every modern web browser supports these, so TindArt will be available on any modern browser. Additionally, if we were to develop a different front-end interface in the future - for example, a mobile app - we would only require additional logic in the route classes to return a different front-end view when accessed from the new front-end interface. The database, models, and image store access would not require any changes, meaning that we could create such an interface without worrying about rewriting much of our application.

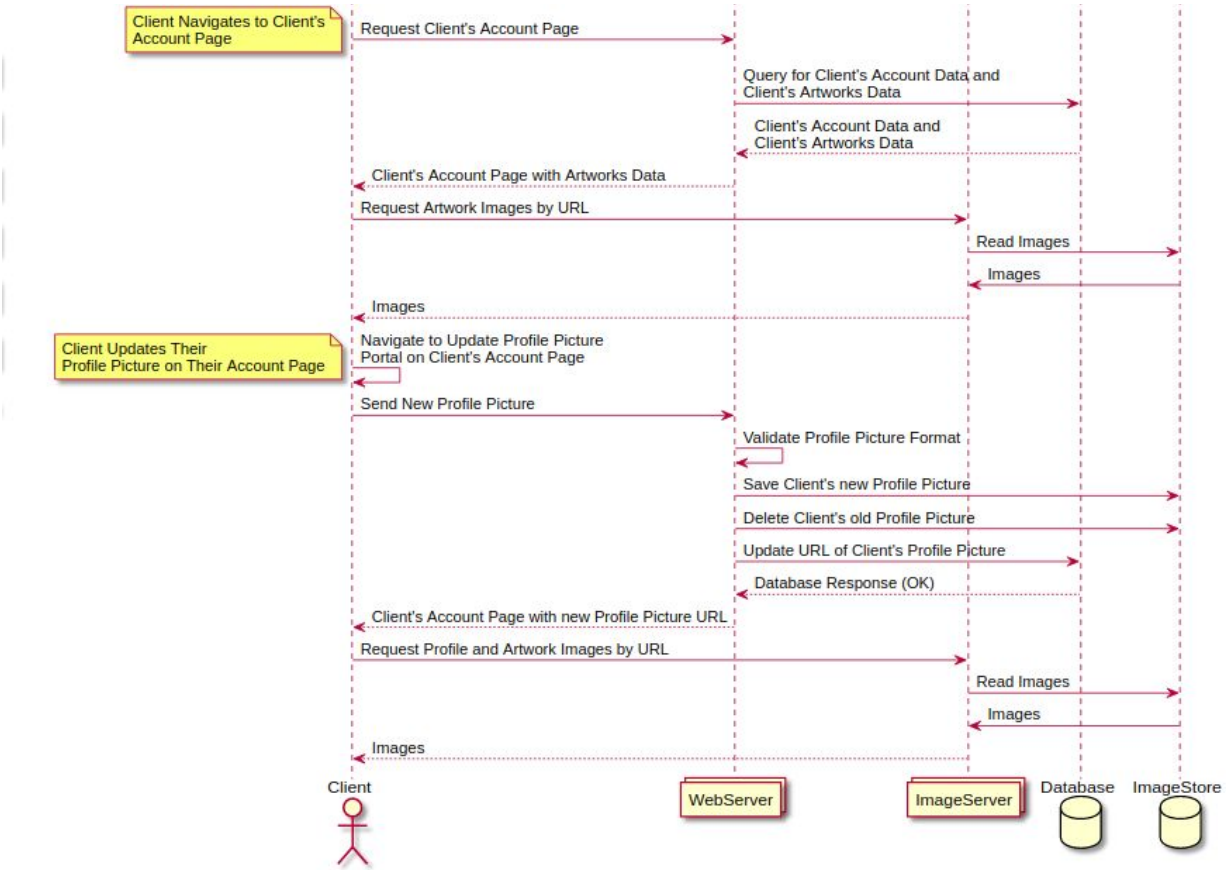
Back-End Portability

Our backend will be built using two docker images - one for the image server, one for the web server - and docker-compose, allowing our servers to run on any hardware, operating system, or cloud service that supports docker with no additional configuration. Additionally, our image storage system is tied only to a generic filesystem, which means we can be incredibly flexible with regards to the storage system that we choose - whether that be a simple networked harddrive, a cloud storage system, or any other kind of filesystem.

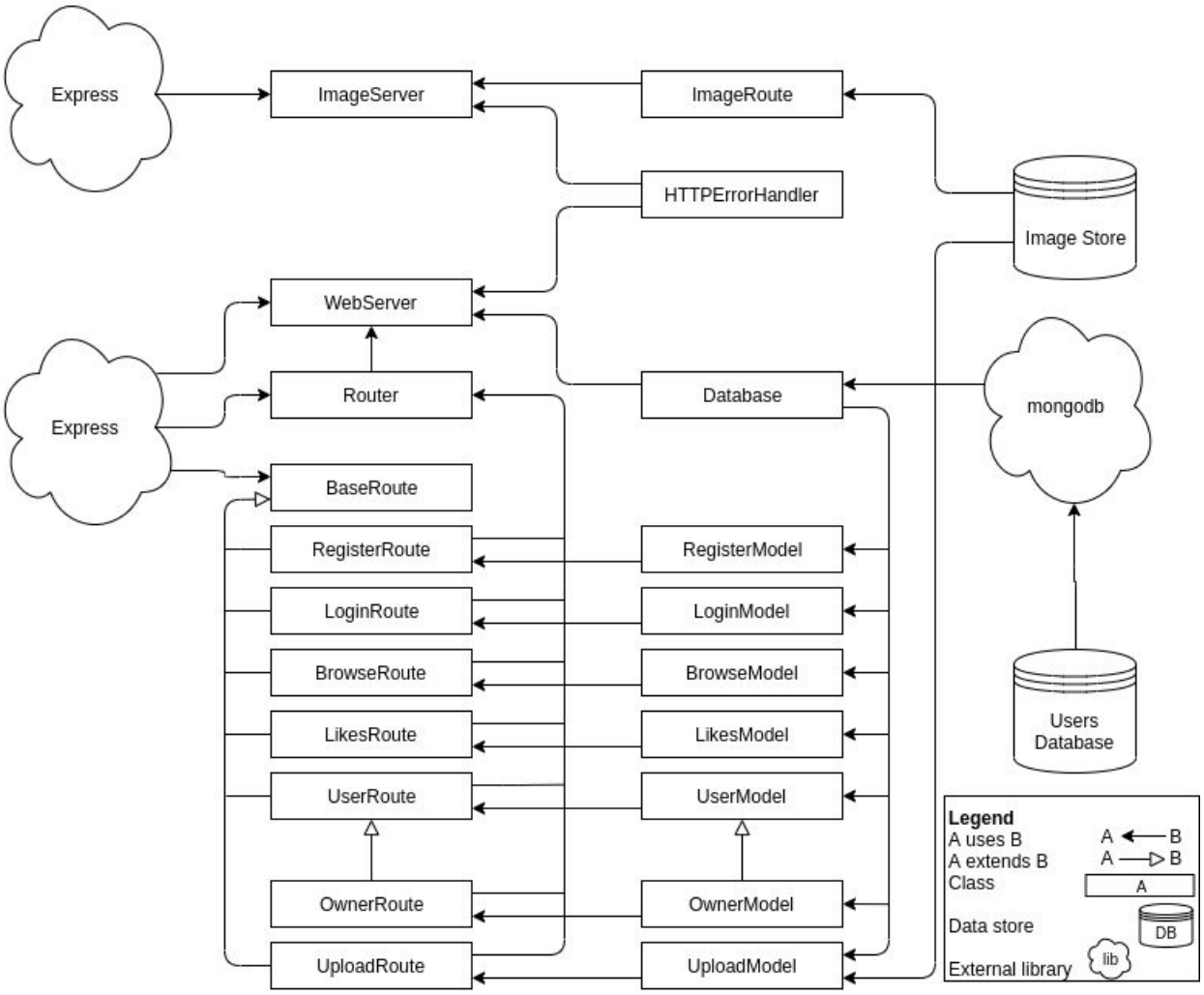
Appendix A: Dynamic View - Sequence Diagram




Client Navigates to Their Liked Artworks Page
Request Client's Liked Artworks Page
Query for Client's Liked Artworks
Artwork Data and the Artists' Account Data
Client's Liked Artworks Page with Artwork Data and the Artists' Account IDs
Request Artwork Images by URL
Read Images
Images
Images
Client Navigates to the Account Page of an Owner of one of their Liked Images
Request an Artist's Account Page
Query for Artist's Account Data and Artist's Artworks Data by Account ID
Artist's Account Data and Artist's Artworks Data
Artist's Account Page with Artworks Data
Request Artwork Images by URL
Read Images
Images
Images
Browse Artist's Artworks
Client Likes an Artwork on an Account Page
Like an Artwork on an Account Page
Add Artwork's ID and Artist's Account ID to List of Users Liked Photos
Database Response (OK)
Server Response (OK)


Client Navigates to Client's Account Page
Request Client's Account Page
Query for Client's Account Data and Client's Artworks Data
Client's Account Data and Client's Artworks Data
Client's Account Page with Artworks Data
Request Artwork Images by URL
Read Images
Images
Images
Client Updates Their Profile Picture on Their Account Page
Navigate to Update Profile Picture Portal on Client's Account Page
Send New Profile Picture
Validate Profile Picture Format
Save Client's new Profile Picture
Delete Client's old Profile Picture
Update URL of Client's Profile Picture
Database Response (OK)
Client's Account Page with new Profile Picture URL
Request Profile and Artwork Images by URL
Read Images
Images
Images
Client
WebServer
ImageServer
Database
ImageStore

Appendix B: Static View - Class Diagram



Appendix C: Architectural Decision Records

ADR #1: Photo storage and access

We considered two ways to store images that are uploaded and accessed by users:

- Store the images on the web server, or
- Store the images on a separate file server.

Negative Consequences

Web Server

- Increased web server latency
- Decreased web server performance
- Inability to scale individual services

Separate File Server

- Harder to implement
- Higher initial development time
- Need a connection point from the web server to the file server

Benefits

Web Server

- Single server
- Easier to manage
- Quicker to implement
- Conceptually simpler

Separate File Server

- Separation of services
- Removes stress from web server
- Allows for individual service scaling
- Easier to maintain each service

Decision

It was decided to create a separate file server to alleviate stress on the web server for better performance of the application, at the expense of increased architectural complexity.

Status

Accepted

ADR #2: Photo storage convention

We considered two ways to organize stored photos on the file server filesystem:

- Store each photo in a directory structure based on the uploading user, or
- Store each photo with a unique name in a single, global, unstructured directory.

Negative Consequences

User Directory

- Requires masking of user data in the photo url
- Duplication of user-to-photo relation, in both the database and directory structures
- Requires unique name for each folder

Single Directory

- Latency of photo downloads will likely increase over time, due to lookup cost, if not managed

Benefits

User Directory

- Partitioning of data can stabilize latency

Single Directory

- Conceptually simpler
- Easier to implement

Decision

It was decided that storing the photos in a single directory with the use of UUIDs for unique image filenames would be the better option, as the expected latency costs of doing so is low and therefore the increased simplicity of doing so is more beneficial.

Status

Accepted

ADR#3: Database access

We considered two ways with which the web server could access the database:

- Each route has a model which directly calls mongodb functions, or
- One database class abstracts mongodb functions to provide generic database functionality.

Negative Consequences

Direct Access

- Modifiability reduced, if the database or schema ever change, then a significant amount of work would be required for each route

Database Model

- Performance overhead
- Initial implementation takes longer

Benefits

Direct Access

- Quick implementation of each individual feature

Database Model

- Easier to modify later
- More modular architecture
- More localized effects of changing the database or the access to the database
- Likely better database security due to fewer database access points

Decision

It was decided that the use of a single database class to handle the connection to the database and expose required functionality to each route's database model was the better option, to allow for a more modular architecture at the cost of a higher initial build time.

Status

Accepted