Name: Eoin Doherty

Title: Note Taking Website

Description: A website where users can create and upload notes. Users can also search for other users, and notes.

Features Implemented:

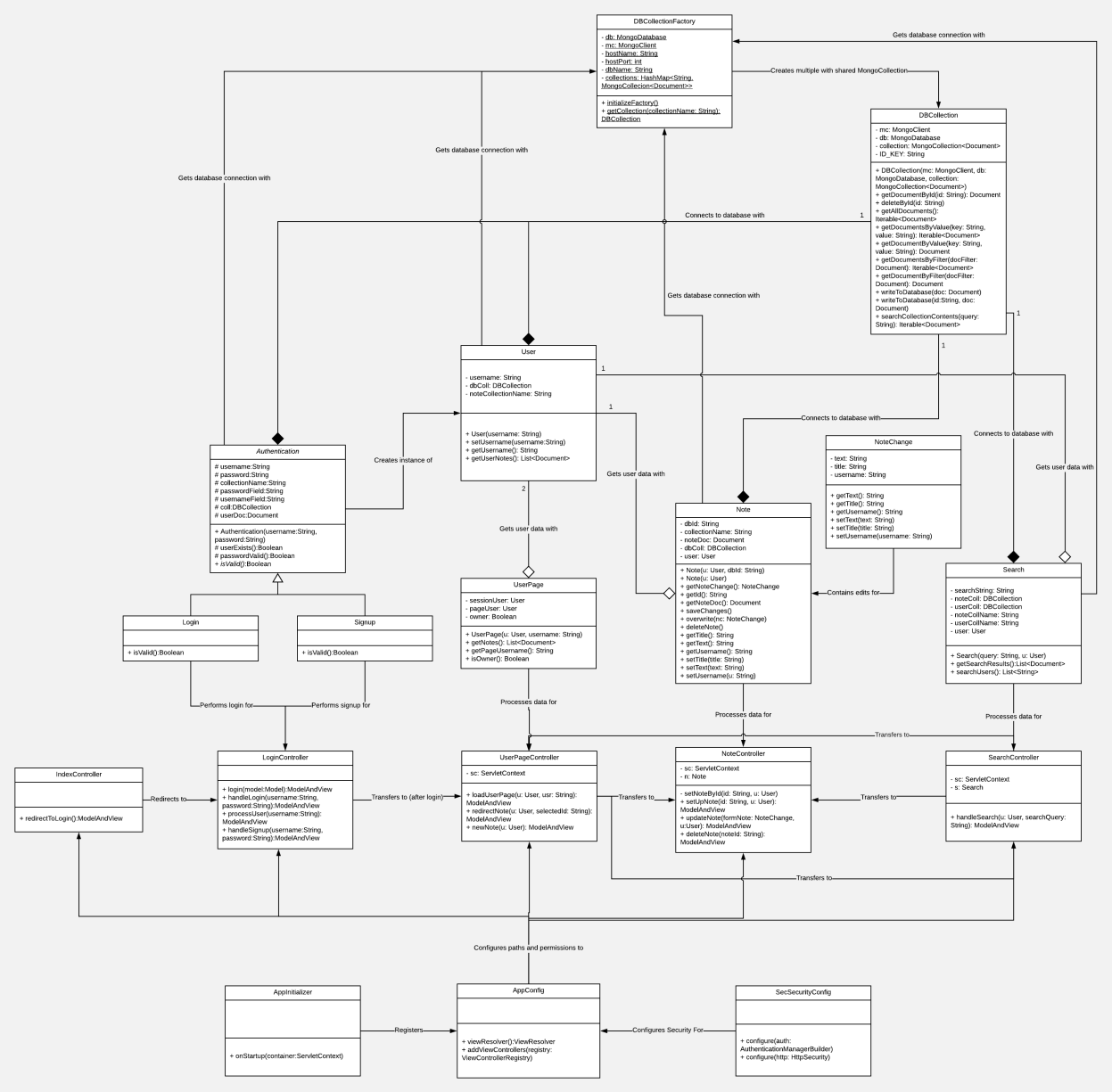
ID	Feature
UR-01	Users should be able to enter a valid username and password to log in. If they are not registered yet, they should be able to enter their credentials to sign up for an account.
UR-02	Once logged in, users should be able to create documents from their profile page.
UR-03.1	After entering new information for a note, users can click the save button to save the document. They must enter a title for the note.
UR-05	Users can edit or remove notes they have previously made.
UR-06.2	Users should be able to delete notes.
UR-08	Users can search for notes by content.
UR-09	Users can search for other users and view their publicly available notes.

Features Not Implemented:

ID	Feature
UR-03.2	Users have the option of adding tags to notes.
UR-04	When viewing a document, users should be able to share that document with other users by clicking a share button and entering a list of users who can view the document. They can also choose to share the document with all users. They should be able to specify whether or not the shared users can copy that document.
UR-07	While viewing a document that has been shared with them, users should be able to copy that document if the original note owner has made it sharable.

Final Class Diagram

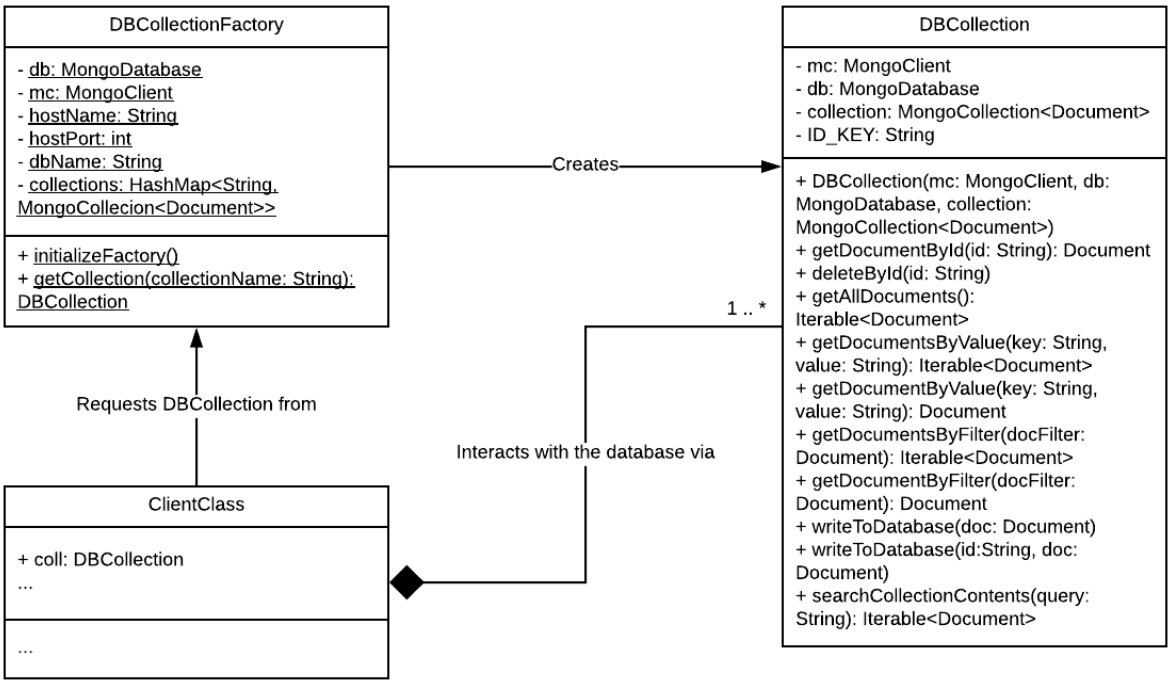
(Also available in the repo: **/Final_Class_Diagram.png**)



I had to change most of the class diagram once I started implementing things. The original class diagram put the controllers in charge of business logic while models just stored data for the views. This is not proper MVC; I refactored the system so the models would execute the business logic and the controllers would just handle redirects and routing. I also had to refactor the database connection to cut down on the number of open connections to the database. Before, each instance of a page would open a new connection to a database collection (MongoDB's equivalent to a table). I changed the database connection to use a flyweight factory to ensure that only one connection exists per collection at any time. The core

structure from before is still there, however. There is only one controller per view (.jsp files that are not in the class diagram) and only one model per controller.

Design Pattern: Flyweight Factory for Database Connection



UML Diagram of DBCollection and DBCollectionFactory implementation
*The same structure is at the top of the class diagram in the **Final Class Diagram** section*

For the database connection, I used flyweight objects and a flyweight factory. Connections to the database are handled by DBCollection, the flyweight object. Client classes, like the models, can obtain a DBCollection by calling DBCollectionFactory.getCollection, the flyweight factory, and passing it the name of the collection that is needed. The factory will check a map of collection names to MongoCollections to see if a MongoCollection (a class from the MongoDB API) has already been instantiated for that collection. If it has, the factory will pass that MongoCollection to the DBCollection constructor and return the new DBCollection to the client object. If a MongoCollection has not been created for that collection, it will create one, add it to the map, and pass the new MongoCollection to the DBCollection object's constructor.

The Intrinsic state of the DBCollection object is the MongoCollection, the MongoDatabase, and the MongoClient. This implementation does not make full use of the flyweight design pattern since each DBCollection does not have any extrinsic state. The factory does create a new DBCollection object with each call to getCollection, however, so adding extrinsic properties would work and function as expected. I thought about adding a User class

as an extrinsic property, but database access in this project did not depend on user permissions.

I chose this design pattern to cut down on the number of connections to the database. Every MongoCollection instance is a new connection to the database, and using separate connections for each model is not scalable. Using the DBCollection class and its factory, I could restrict the number of connections to one for every collection and wrap the MongoCollection object in another class to add more general functionality. Any model or session specific properties like which user created this class can be added to DBCollection later if needed.

What I Have Learned About Analysis and Design

I have learned much about software architecture and planning out a system's structure from this project and this class. I learned that planning using class diagrams can be a useful guide for how to structure code. Even though I changed my class layout after planning, I still had a general idea of what models, views, controllers, and other classes I would need to create and how they needed to tie together. This planning helped me organize my code and create a stable website. I also learned that user requirements can be a useful tool for measuring progress and deciding on what to implement next. If I did not know what I wanted or needed to add next, I looked at my requirements and worked on the feature I thought was most important. This project has taught me how to create class diagrams, how to write requirements, how to implement an MVC application, and how to implement design patterns, but more importantly, it has taught me how useful planning is in a software project. After I wrote requirements and laid out the structure of my website, writing the code for it was fairly easy. Creating this website would have been much harder if I had rushed into it without planning.