Assignment 5

Final Assessment

Submit a single zip file called **assignment5.zip**. **You should not submit your node_modules folder or database folder. Instead, you should submit the resources required to install your dependencies using NPM.** The TA will run the provided database-initializer.js file before grading your assignment. **If your assignment requires additional database setup, you should modify the database-initializer.js file and provide a copy in your submission**. This assignment has 100 marks. You should read the marking scheme posted on cuLearn for details.

Assignment Background

In this assignment, you will develop a web application that will allow players to receive cards from a collectible card game and trade those cards amongst their friends. The card data used for this assignment is taken from the game Hearthstone. This is the same data used in tutorial #8 – a description of the structure of the data is provided at the end of this document. A `database-initializer.js` file has been provided that will create an empty '`a5`' database in MongoDB and add each card to a '`cards`' collection. **Note that the initializer will also delete any information in the 'a5' database**. Each user that registers for your web application will maintain their own set of cards and their own set of friends. Friends within the application will be able to view each other's cards and propose/accept trades with each other. All data your server uses for this assignment (cards, user profiles, session data, etc.) must be persistent. Your server should be able to be restarted at any point and all the data should still be present.

The specification of this assignment has intentionally been left more vague than previous assignments. The details of the overall design of the application (routes, pages, etc.) are left up to you. You are expected to make decisions that will result in a usable and reactive application that is scalable and robust. You will be required to include a short document with your submission explaining how to use your application and arguing in support of the design decisions that you made.

The list of requirements your application must meet are described in the sections below. You are encouraged to read the entire document before starting your design. Some more details of specific requirements may be available in the marking scheme. Additionally, some tips for success have been included at the end of this document.

Registration, Logging In/Out

Your application must provide a way for new users to register by providing a username and password. Duplicate usernames should not be allowed. New users should be initialized with no friends and a set of 10 randomly selected cards. A user must be provided with a way to view their own set of cards.

Existing users must be able to log in to the application using their username and password. You must also provide a way to log out of the application.

Friends

Users should be able to propose friend requests to other users of the application. A user should be able to search for other users by performing a username search. From the results of this search, the user should be able to propose a friend request to any other user that is not already their friend. Users should not officially become friends until the other user has accepted the friend request. If a user is logged in to the system when somebody tries to add them as a friend, they should receive some sort of notification that a friend request is available in 'real-time' (i.e., without refreshing their page). Note that this does not have to be immediate but should show up in a relatively short time span (e.g., 5 seconds).

A user who receives a friend request must be able to either approve or reject the friend request. If the user rejects a friend request, no changes to user profiles should be made (i.e., they should not become friends). If the user accepts a friend request, both users involved should become friends. That is, if X proposes a friend request to Y, and Y accepts, then Y is friends with X and X is friends with Y. After a request has been rejected or approved, it should be deleted from the server.

Users should be able to view a list of their friends and view their friends' list of cards. A user who is not your friend should be not able to view your cards.

Trades

Users should be able to propose trades to their friends. To propose a trade, a user needs a way to select a subset of their own cards, one of their friends, and a subset of their friend's cards. This should be done in a reactive manner, without requiring multiple page views. The card subsets could be empty (i.e., I give you nothing and you give me something), the entirety of their set of cards (i.e., I give you all my cards), or anything in between. Once the user has selected a trading partner and cards, they should be able to propose the trade. When showing a card during a trade, or elsewhere in the application, you can use the card's name attribute. You can choose to provide a method for viewing additional card information (class, rarity, artist, etc.) or use the additional card information for extra features, if you want.

As with friend requests, a user should receive some form of notification that they have received a trade request without having to refresh whatever page they are currently on. The receiving user should be able to view the trade details and choose whether to accept or reject the trade. If the trade is rejected, no changes should be made. If the trade is accepted, one of two possible cases may occur:

1. Both players still have the cards required to make the trade, in which case the selected cards should be switched between the two users. The user who has just accepted the trade should receive a notification that the trade has been completed. The proposing user does not need to receive a notification, but you can add this as an additional feature if you want.
2. One of the players can no longer fulfill the requirements of the trade, in which case no changes should occur. This case may occur if the user X proposes a trade to user Y, then also proposes a trade to user Z involving some of the same cards. If user Z accepts the trade first, then user X will not have the cards to fulfill the trade with user Y if Y accepts the trade. The user who accepted the trade should be notified that the trade is no longer valid. The proposing user does not need to receive a notification, but you can add this as an additional feature if you want. Alternatively, you can write the trade proposal code in a way that does not allow invalid trades to be proposed (e.g., by not allowing the same card to be used in more than one proposed trade simultaneously).

Once the trade has been accepted or rejected, the trade should be removed from the server.

Design Document

You must submit a PDF document that explains how to setup your system and how to test each piece of the required functionality. Ideally, it should be clear to the TA how to fully test each piece of functionality mentioned in the marking scheme, as well as any additional functionality you have added. This should be significantly more detailed than the usual README file that has been submitted with previous assignments. This document must also describe the design decisions you made, including a description of how your data is stored, what routes your server supports, and justification for all your design decisions.

Recap

Your zip file should contain all the resources required for your assignment to be installed and run by the TA, as well as your design document. **You should not submit your node_modules folder or your MongoDB database folder.** Submit your **assignment5.zip** file to cuLearn. Make sure you download the zip after submitting, verify the file contents, and ensure that your installation instructions are sufficient.

Summary of Fields in Card Documents:
Each card in the dataset has, at minimum, the following field names:

1. _id: the unique ID created by MongoDB when the card was inserted
2. artist: a string representing the name of the artist(s) that created the card art
3. name: a string representing the name of the card
4. cardClass: a string representing the class of player that can use the card
5. rarity: a string representing the rarity of the card
6. attack: an integer representing the attack value of the card
7. health: an integer representing the health value of the card

Tips for Success:

1. Identify patterns. Previous tutorials have covered similar concepts that will be useful here. As an obvious example, tutorial #8 worked with the same card data used in this assignment. Earlier tutorials involving polling will also be useful in handling notifications of friend/trade requests and improving the overall reactivity of your application. Additionally, certain parts of the requirements share similarities. Proposing friends and proposing trades involve much of the same logic, but with different data. The better you get at recognizing the patterns, the more efficient you will be in creating solutions.
2. Design first. Figure out what information you need to store in order to meet the application requirements (will adding collections help simplify your solution?). Consider the queries that you will need to execute and ensure that your design is sound before trying to code.
3. Discuss on Discord. Discussion is a severely underrated learning tool. We should be taking advantage of our large student body and sharing ideas. You are fully encouraged to discuss overall design and general approaches to the assignment. Exchanging ideas with others will lead many of you to have better overall designs in the end, which will ultimately save you time in implementing and debugging your solution.

4. Use Mongoose. Its features may reduce your complexity and save you a significant amount of effort.
5. Build incrementally. Always keep your overall design in mind but break the entire system down into smaller components and add features in a step-by-step fashion. For example, you could start by adding support for registration. Follow that with viewing cards, submitting friend requests, receiving pending friend requests, approving friend requests, and so on.
6. Confirm your database manipulations are correct BEFORE committing to a database. As an example, perform some 'dry runs' by logging out the documents you would be writing to the database in the console before you actually perform any write operations. This will drastically reduce the amount of time you spend manually manipulating the database to fix errors you have carelessly introduced.
7. Add authorization later. Some of the requirements state that only friends should be able to do X or view Y. Write the code without these rules to start – let everybody do everything at first. You can add the authorization in later.