COMP2406 Assignment 5 Design Documentation

Hearthstone Trading Card Web Application

D. Zachary van Noppen
101086708

April 20, 2020

1.0 OVERVIEW

The design of the web application is focussed on creating a visually appealing, intuitive interface that will allow users to interact with other users and trade virtual cards. The design of the project will be outlined in three sections: Server Routes, Data Storage and Page Views. Information on the functions and routes pertaining to each section will be included in brackets after each heading for reference. The project was created with NodeJS, using a mongoDB database (managed with mongoose), running on an ExpressJS web server, with front end rendering made in the PUG template engine.

2.0 SERVER ROUTES

The application supports two routes: requests and users. The initial idea was to use the users route to be able to specify individual users by id, allowing them to be searched by URL in the server. However, after focussing more on a single page application, this approach required verification of every user attempting to request another via URL. The approach this project implements is loading all relevant friend data on page loads, reducing the amount of requests that go through the server, and therefore removing the possibility of requesting a user someone is not authorized to view.

2.1 Users Route

Since the users route is not used as a way to navigate between friends, it is instead used only to handle querying for users to add (through the **/add** route) and logging in and out (by posting to "**/**" and the /**logout** route).

2.1.1 Adding friends ("/add")

This route does not add friends, as that is handled by a request in the request router. The **/add** route returns all users requested by and AJAX request on the main page. Each time a user types in the search box, the database is searched and all users that contain the query string are returned. Users who are already friends are excluded.

2.1.2 Login handling (“/”, “/logout”)

When a user registers or logs in to the page from *http://localhost:3000/* the form data is sent to */users/* to be processed. It is verified for authentication, making sure that no users other than the current one are already logged in, and then passed into a handler function that proceeds depending on the type of login.

2.1.2a Registration (loginHandler(), createNewUser())

For registration, ten random cards are selected from the data and then a new user is created. If the username does not already exist, it is added to the new user object and saved in the database. The main page (**index.pug**) is then generated based on the user information created. More information on how the page is created will be found in section 4.0.

2.1.2b Login (loginHandler())

If the user is not registering their account, the handler will search for their information in the database. If found, a new session will be created and the main page (**index.pug**) will be rendered based on the user found in the database. An additional step is completed to query the database for the names of each friend on the user’s friend list. This is done since the friend array stores ObjectIDs instead of usernames.

2.1.2c Logout (loginHandler())

The logout route takes the session currently in use and destroys it, removing it from the database. The user is redirected back to the home page.

2.2 Requests Route

The **/requests** route has three sub routes: get(“/”), post(“/”) and post(“/:requestID”). These three routes handle the request verification of all trades and friend requests on the server.

2.2.1 Server Polling (get ”/”, getRequests())

This subroute is used when polling the server. It searches the database for any entries that correlate to the userID of the current session, and returns those elements as an array. That data is then checked on the client to see if it has already been seen, or if it should be rendered again. It is handled in this way instead of storing if it has been sent on the server) to allow for the page to be refreshed and those notifications sent again.

Another way of handling this would be to store what data has been sent in a session variable, however that would mean that if the user had two pages open, only one would receive the notifications.

2.2.2 Creating Requests (post ”/”, createRequest())

When sending a request, the server determines if the user is making a trade or a friend request based on a type parameter sent in the body. The server then builds new Request Schema objects based on the request parameters. The server saves that data and responds successfully upon completion.

2.2.3 Request Handling (post ”/:requestID”, verifyRequest())

This route takes a request id and a boolean value as parameters. The boolean determines if the user is accepting or declining the trade. Declining (false boolean value) deletes the request from the server. Otherwise, the request is found from the server based on the request id, and the type (friend or trade) is determined.

Friend requests find both users in the database and add each other’s names to the friends list. The request is then deleted.

Trade requests are more involved than friend requests. Since only card names are sent to the server, each card object must be found in the database and kept in an array, both for offered and sent cards. Then, the sending and receiving users are found and checked to see if the cards found previously are still available for trade. If both users are found, and the cards are valid for trading, the cards are swapped and a successful response is sent back to the server.

3.0 DATA STORAGE

The data used by the web app are stored in the “a5” mongoDB database. Four collections are used to store the data: users, requests, cards and the auto generated sessions database. The three collections designed for the application are detailed specifically to make sure data is validated based on mongoose schema.

3.1 UserSchema

The UserSchema defines the data that can be stored in the database. There are four properties: username, password, cards and friends. Username and password are strings that can only hold at maximum 20 alphanumeric characters/numbers. The cards and friends elements are arrays.

Cards hold entire card documents. This saves time querying for each card that a user holds when sending user information later in the program. While this causes some data sent with the user to be useless (not every request for a user will need the race information about a certain card), it reduces the amount of queries to the database for users that have large card collections. Additionally, card data is only ever sent with the user once, on page loads, which justifies the saved query time.

Friends however, store only an array of usernames. The user does not need to know all the information about their friends. A user should not be able to access the password of their friends, so just the username is used. ObjectIDs could have been used, however this causes unnecessary queries on the server and since usernames are unique and not changeable in the application, it works the same.

3.2 CardSchema

Each item in the card schema is either a string or number, depending on expected numeric or character data.

3.3 RequestSchema

Requests can have two types: either friend or trade. Because the types are used differently, the sender, recipient and type properties are required, but the offer and return arrays are not. These are used only in trades. All required properties are strings, while the offer and return arrays hold the names of cards that will be traded.

4.0 PAGE VIEWING

The page data is generated in PUG, though modified significantly by the **index.js** script throughout a user's session. The PUG page takes a username, array of card objects and array of friend names as parameters to render the page. The friends list, and list of cards is then generated by the page. When friends or cards are added the page is reloaded automatically, displaying the changes. However, this is not handled dynamically like requests are. These actions are redirects to decrease the amount of polling on the server.

The requests on the page are dynamically created as they are received from the server. The page polls the server looking for requests every three seconds, and if the request has not already been sent, the page will create a new request notification, and HTML element to allow the user to perform an action. The requests that have already arrived are stored on the page, not in the server so that when the page is reloaded the requests are sent again.