Apostle TCG Database Project

Collaborators:

Ethan Wen	015621918	ethan.wen@sjsu.edu
Kyaw Soe Han	015543671	kyawsoe.han@sjsu.edu
Kyle Chen	013983424	kyle.chen@sjsu.edu
Mathew Estrada	016927157	mathew.estrada@sjsu.edu
Nathan Pham	015903446	nghia.pham@sjsu.edu

Application Overview and Goals

Compared to other card based games, trading card games(TCG) often have a much larger overall card pool - there are roughly 13k unique Pokemon TCG cards, and over 22k unique Magic: The Gathering cards. This number makes it difficult for players to remember every card off the top of their head, and players with large collections may find it difficult to locate a card they need to serve a specific purpose within a deck.

Ethan is currently designing a digital TCG as a hobby, which as of right now has over 1000 unique cards, although a majority of them are currently unrendered. The digital format of the card game means all cards exist as data, either in the form of spreadsheet values or png images. This project aims to create a web based database application which will help players navigate through and organize this large quantity of data in manageable and useful ways.

The basic functionality of the application allows users to view card data in a digestible manner, with a means to navigate and narrow down to more specific cards using a set of search parameters and filters.

Functional Requirements

In order to meet the above user needs, this project will use a two level client-server architecture. We choose two-tier over three-tier because a website of this nature is unlikely to require user accounts, and thus there is no sensitive data which needs to be secured behind additional levels. Furthermore, the estimated number of users will be fairly low, so the project does not need to be highly scalable, and the simpler two-tier architecture is easier to manage.

The data server will contain tables that hold all card data. The display layer displays cards as an array of images on a series of scrolling pages, and the top of the page will have filter parameters so the set can be tuned. A basic home page with a search bar will allow users

to enter queries, redirect them to a more advanced search page, or redirect them directly to the set of all cards. Image data will be stored outside of the database, within the application files.

As new cards are created for the game, they can be added by game administrators as json files to the Github repository. There does not need to be any user infrastructure within the application to add card data, as that is outside the scope of this project.

ER Data Model

Database Functional Requirements:

Card:

Each card belongs to a set and each card can be uniquely identified by the ID of the set it is in and the Card's ID in that set. Cards also have a name, description text, and multiple colors assigned to it. It can also have a list of multiple subtypes. Every card has an image. There are keywords that are associated with each card.

Card Types:

Each card is categorized into 4 main categories: Character, Spell, Item and District. Card types cannot be overlapped. Character cards have attack, health, and cost values. Spell cards have cost values. Item cards also have cost values. District cards have tier and orb values. District cards can have multiple orb values.

Image:

Each image has a unique ImageID and an artist associated with it.

Keyword:

Every card has one or more keywords. Each keyword has a string for its word.

Set:

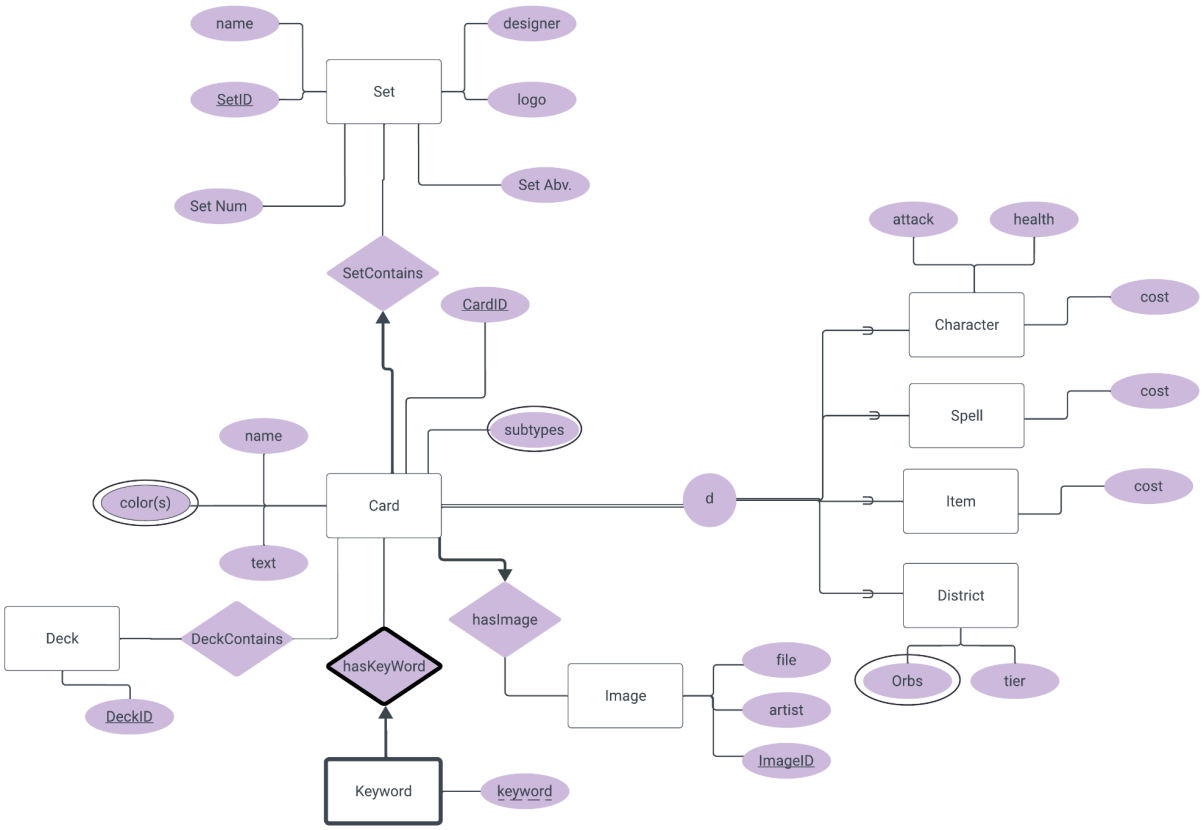
Each set has a unique SetID. Each set also has a name, set designer, and logo.

Deck:

Each deck has a unique DeckID. A deck has a list of cards that are part of the deck collection.

ER diagram :

https://lucid.app/lucidchart/e327e2fd-0bbd-4432-bc35-2f38c399691b/edit?viewport_loc=-822%2C-499%2C2796%2C1350%2C0_0&invitationId=inv_62c89ea6-2fa2-4654-8ca2-62a81e886ce3



Database Design and Relational Schema

Set: (set_ID: String, name: String, designer: String, logo: String)

PK	set_ID	VARCHAR/STRING
	name	VARCHAR/STRING
	designer	VARCHAR/STRING
	logo	VARCHAR/STRING

Functional Dependencies:

set_ID → name
set_ID → designer
set_ID → logo

All non-candidate keys (name, designer, logo) are functionally dependent on the primary key (set_ID). There are no other functional dependencies, so the relation is in BCNF.

Card: (card_ID: String, set_ID: String, name: String, text: Text, cardType: String, image_ID: String)

PK	card_ID	VARCHAR/STRING
PK, FK to Set.set_ID	set_ID	VARCHAR/STRING
	name	VARCHAR/STRING
	text	TEXT
	cardType	VARCHAR/STRING
FK to Image.image_ID	image_ID	VARCHAR/STRING

Functional Dependencies:

card_ID, set_ID → name
card_ID, set_ID → text
card_ID, set_ID → cardType
card_ID, set_ID → image_ID

All non-candidate keys (name, text, cardType, and image_ID) are functionally dependent on the primary key (card_ID, set_ID). There are no other functional dependencies, so the relation is in BCNF.

Subtypes: (card_ID: String, set_ID: String, subtype: String)

PK, FK to Card.card_ID	card_ID	VARCHAR/STRING
PK, FK to Card.set_ID	set_ID	VARCHAR/STRING
PK	subtype	VARCHAR/STRING

All attributes of this relation are part of the primary key so the relation is in BCNF.

Character: (card_ID: String, set_ID: String, attack: Integer, health: Integer, cost: Integer)

PK, FK to Card.card_ID	card_ID	VARCHAR/STRING
PK, FK to Card.set_ID	set_ID	VARCHAR/STRING
	attack	INTEGER
	health	INTEGER
	cost	INTEGER

Functional Dependencies:

card_ID, set_ID → attack
card_ID, set_ID → health
card_ID, set_ID → cost

All non-candidate keys (attack, health, cost) are functionally dependent on the primary key (card_ID, set_ID). There are no other functional dependencies so the relation is in BCNF.

Spell: (card_ID: String, set_ID: String, cost: Integer)

PK, FK to Card.card_ID	card_ID	VARCHAR/STRING
PK, FK to Card.set_ID	set_ID	VARCHAR/STRING
	cost	INTEGER

Functional Dependencies:

card_ID, set_ID → cost

The only non-candidate key (cost) is functionally dependent on the primary key (card_ID, set_ID). There are no other functional dependencies so the relation is in BCNF.

Item: (card_ID: String, set_ID: String, cost: Integer)

PK, FK to Card.card_ID	card_ID	VARCHAR/STRING
PK, FK to Card.set_ID	set_ID	VARCHAR/STRING
	cost	INTEGER

Functional Dependencies:

card_ID, set_ID → cost

The only non-candidate key (cost) is functionally dependent on the primary key (card_ID, set_ID). There are no other functional dependencies so the relation is in BCNF.

District: (card_ID: String, set_ID: String, tier: String)

PK, FK to Card.card_ID	card_ID	VARCHAR/STRING
PK, FK to Card.set_ID	set_ID	VARCHAR/STRING
	tier	VARCHAR/STRING

Functional Dependencies:

card_ID, set_ID → tier

The only non-candidate key (tier) is functionally dependent on the primary key (card_ID, set_ID). There are no other functional dependencies so the relation is in BCNF.

DistrictOrb: (card_ID: String, set_ID: String, orb: String)

PK, FK to District.card_ID	card_ID	VARCHAR/STRING
PK, FK to District.set_ID	set_ID	VARCHAR/STRING
PK	orb	VARCHAR/STRING

All attributes of this relation are part of the primary key so the relation is in BCNF.

Color: (card_ID: String, set_ID: String, color: String)

PK, FK to Card.card_ID	card_ID	VARCHAR/STRING
PK, FK to Card.set_ID	set_ID	VARCHAR/STRING
PK	color	VARCHAR/STRING

All attributes of this relation are part of the primary key so the relation is in BCNF.

Deck: (deck_ID: Integer)

PK	deck_ID	INTEGER

All attributes of this relation are part of the primary key so the relation is in BCNF.

DeckContains: (deck_ID: String, card_ID: String, set_ID: String)

PK, FK to Deck.deck_ID	deck_ID	INTEGER
PK, FK to Card.card_ID	card_ID	INTEGER
PK, FK to Card.set_ID	set_ID	VARCHAR/STRING

All attributes of this relation are part of the primary key so the relation is in BCNF.

Keyword: (keyword: String, card_ID: String, set_ID: String)

PK	keyword	VARCHAR/STRING
PK, FK to Card.card_ID	card_ID	VARCHAR/STRING
PK, FK to Card.set_ID	set_ID	VARCHAR/STRING

All attributes of this relation are part of the primary key so the relation is in BCNF.

Image: (image_ID: String, file(URL): String, artist: String)

PK	image_ID	VARCHAR
	file(URL)	VARCHAR
	artist	VARCHAR

Functional Dependencies:

image_ID → file(URL)
image_ID → artist

All non-candidate keys (file(URL), artist) are functionally dependent on the primary key (image_ID). There are no other functional dependencies so the relation is in BCNF.

Major Design Decisions

Card Primary Key

Within the database, the primary key of cards is a joint key of varchar and integer as opposed to a single index. This is primarily a game design consideration - cards are generally designed within sets, so their unique identifier is a combination of their set of origin and their number within that set. This also allows for easier implementation of “reprints” and “variant art” cards in the database by allowing two cards that would normally have the same numerical index but exist in different game contexts to be isolated by set.

Indexing

Since our application uses MySQL for the DBMS, the tables are automatically indexed by the primary keys. The primary key is the most efficient attribute to index, so no further indexing of other attributes is needed.

Deck Implementation

We want to limit the storage of user data, so rather than storing decks(built by users) in the database, we opt for an alternative to export it to a JSON file that they can then copy and import when they need to. The primary function of this web app is to search and filter through cards, so this holds priority over user data.

a. This feature was not included in the first release (see Future Expansions)

Implementation Details

Web Design - ReactJS, NodeJS, ExpressJS
CSS - Tailwind
Database - Oracle MySQL

The Frontend was designed in ReactJS using Tailwind CSS as a CSS framework. Additional dependencies were installed, including react-router-dom, which was primarily used to handle client-side routing between different pages and components. It was also used to generate and format URL search queries that could then be passed to the backend routes as query parameters.

The Backend used MySQL, which was a basic project requirement, for the database. It also used NodeJS as the Backend server, with ExpressJS being a Node framework for web applications that allowed us to handle the Backend routing middleware. This was used to connect the database server with our application so that we could pass requests from our frontend to the database and get responses containing the data we wanted to display. It also lets us parse and format our query parameters using routes and write SQL SELECT queries to be passed to the SQL connection.

Example: Select Query to Fetch Card Data

```
getItemCardById: function (setID, cardID, callback) {
    let q = "SELECT * FROM Card\n";
    q += "JOIN itemCard USING (cardID, setID)\n";
    q += "JOIN Color USING (cardID, setID)\n";
    q += "LEFT JOIN Keyword USING (cardID, setID)\n";
    q += "LEFT JOIN Subtype USING (cardID, setID)\n";
    q += "WHERE setID = ? AND cardID = ?";

    pool.query(q, [setID, cardID], function (error, results, fields) {
        if (error) throw error;
        callback(results[0]);
    });
},
```

Example: Select Query Using Search Parameters

```
getCardsByQuery: function (order = "id", query = undefined, callback) {
    let q = 'SELECT * FROM Card\n'
    let values = []

    if (query) {
        q += 'LEFT JOIN CardSet USING(setID)\n';
        q += 'LEFT JOIN Subtype USING(cardID, setID)\n';
        q += 'LEFT JOIN Keyword USING(cardID, setID)\n';
        q += "WHERE 1 = 1\n";
        q += 'AND cardName like ?\n';
        q += 'OR cardID like ?\n';
        q += 'OR setID like ?\n';
        q += 'OR setName like ?\n';
        q += 'OR cardText like ?\n';
        q += 'OR designer like ?\n';
        q += 'OR subtype like ?\n';
        q += 'OR keyword like ?\n';

        for (let i = 0; i < 8; i++) {
            values.push('%' + query + '%');
        }
    }
```

The basic data flow of the application is as follows:
The application fetches data from the sql server with a get request upon rendering a page or passing a search query. This get request is passed onto the backend node server using a route which specifies how we want the backend endpoint to respond. The route parses the request to get the query information and passes it onto handler functions, defined in our cardModel.js file, when we get a request matching one of our routes. The handler functions take the arguments and generate SELECT queries written in SQL format, which is then communicated to the SQL server connection. This generates a response that we can then retrieve and use in our frontend.

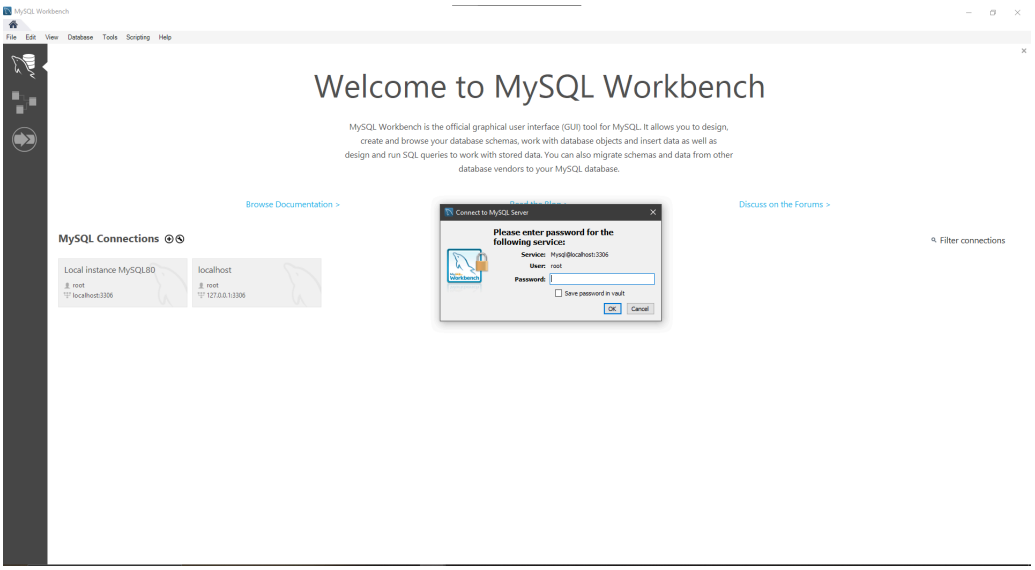
Application Demonstration

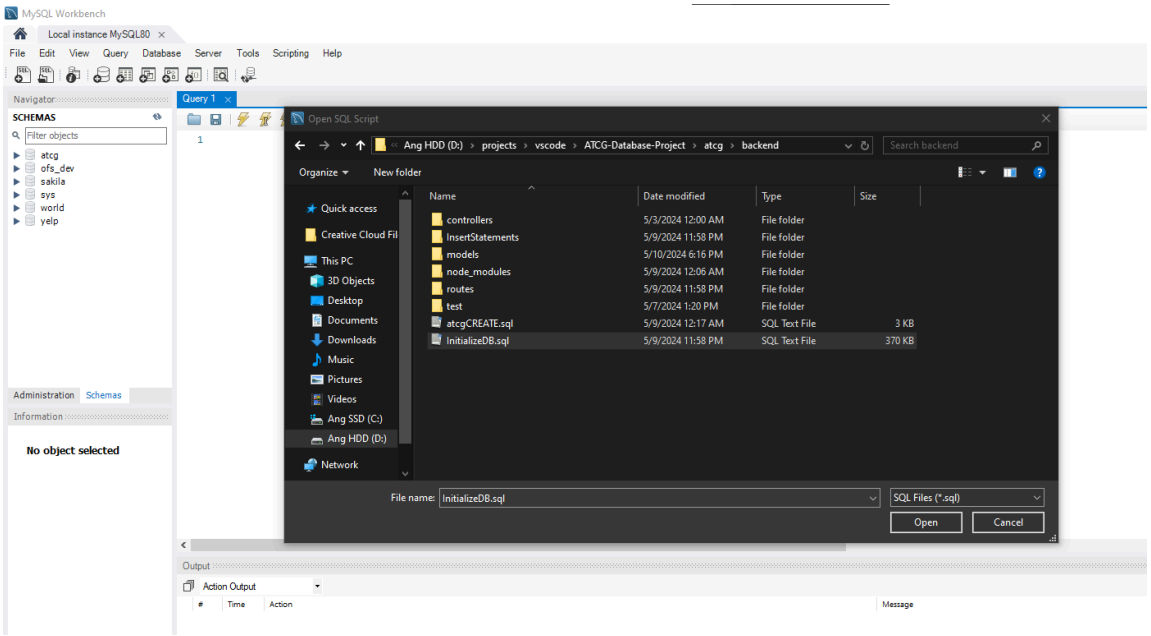
Presentation containing the application demonstration:
https://drive.google.com/file/d/1SpYrGHUoC71B-WZv927BM9EZcJtPnaFp/view?usp=sharing

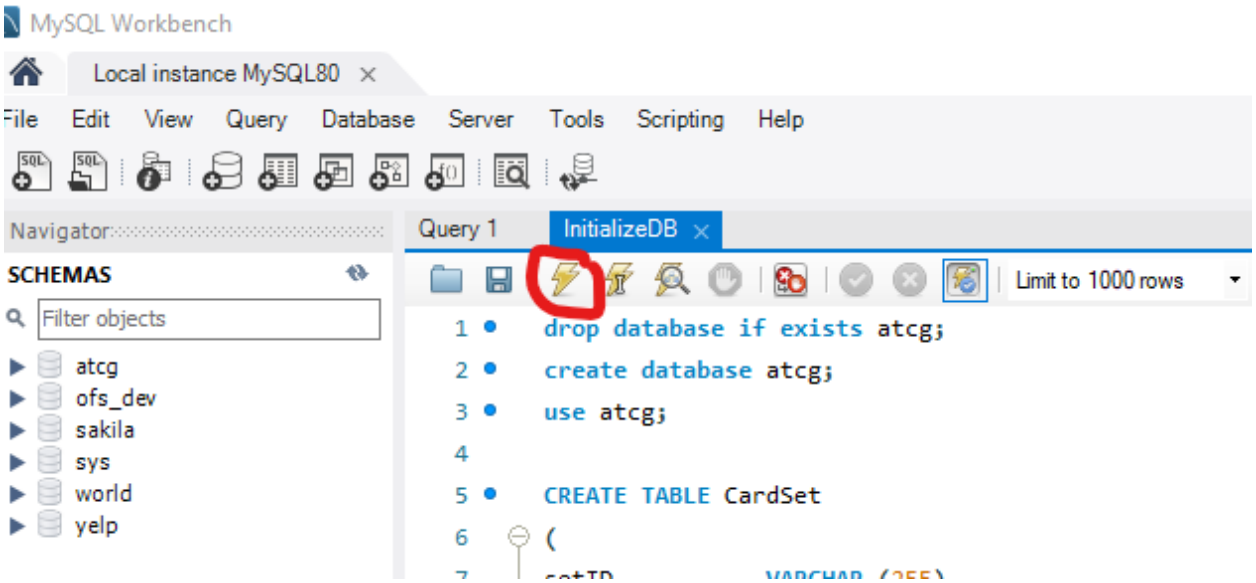
1. **Installation and Setup(For Windows)**
    a) Before starting the project, ensure that you have MySQL, NodeJS, and NPM installed. NodeJS should be at least v.20.8.0 and NPM should be at least 10.2.0
    b) Pull the project from the github repository

c) To install the dependencies: change directory into frontend and backend folders and execute *'npm i'* or *'npm install'* in each. This will install any missing dependencies using the package.json files inside of each folder.

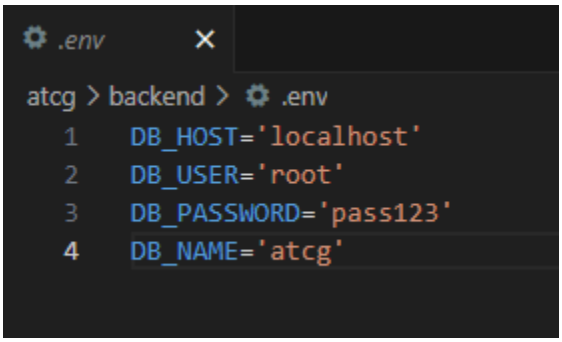
d) Load the database into your MySQL server connection using the initializeDB.sql file located in the /backend folder. The default connection that we will use is the root localhost
Example using MySQL Workbench:



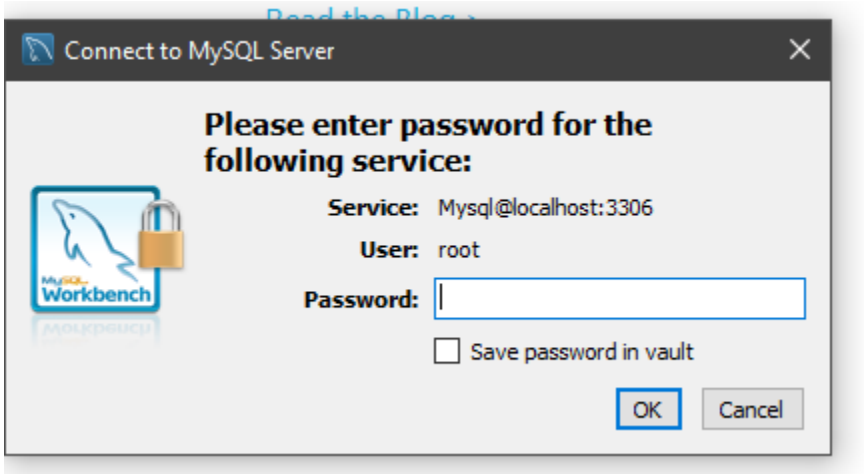




e) Once we see that the atcg database is generated with all the data loaded, we go back to the /backend folder: create a **.env** file, which will contain the environment variables Node uses to connect to this SQL instance



If using the localhost root instance, the only thing that needs to change is the DB_PASSWORD, which should be replaced with the password you used here



f) To check that the connection works: run *'**node --env-file=.env index.js**'* from the backend file, which should generate this response

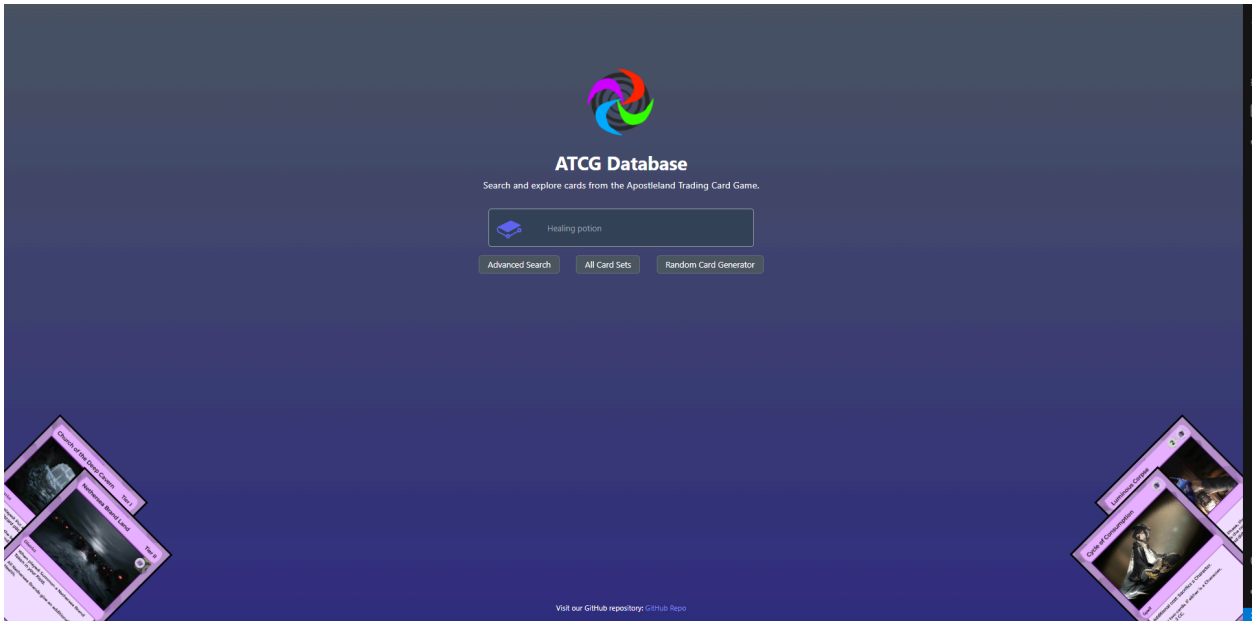
```
D:\projects\vscode\ATCG-Database-Project\atcg\backend>node --env-file=.env index.js
Server running on http://localhost:3001
```

2. Starting the Project

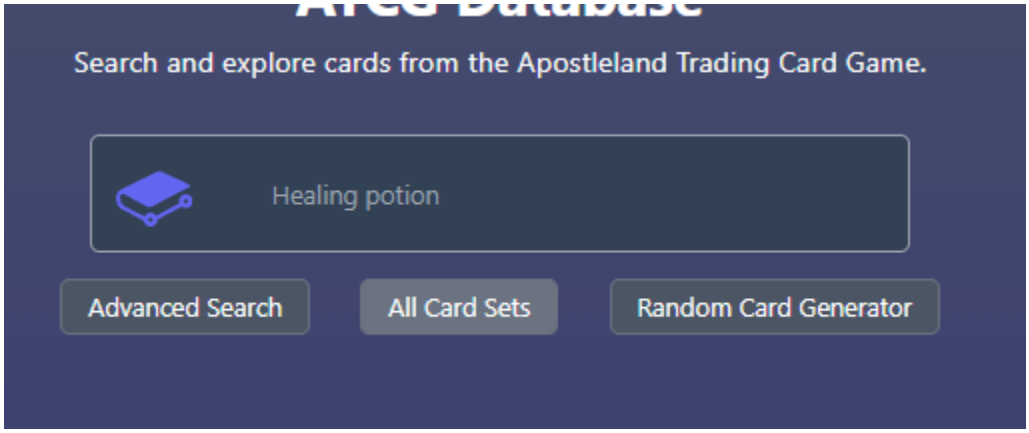
a) To run the application, both backend and frontend servers must be running
b) Navigate to /atcg/frontend and run ***npm start***
c) Navigate to /atcg/backend and run ***node --env-file=.env index.js***
d) After the front end server initializes, it should open up a browser window at http://localhost:3000/. If not you can navigate there yourself

Feature Demonstration

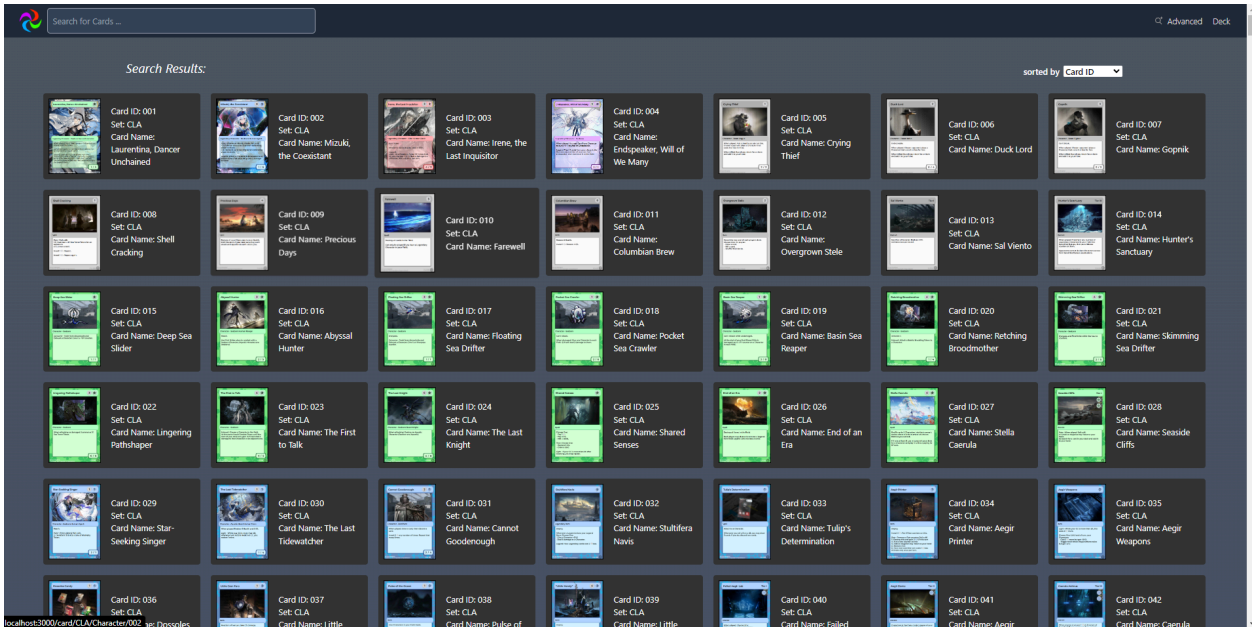
Home Page



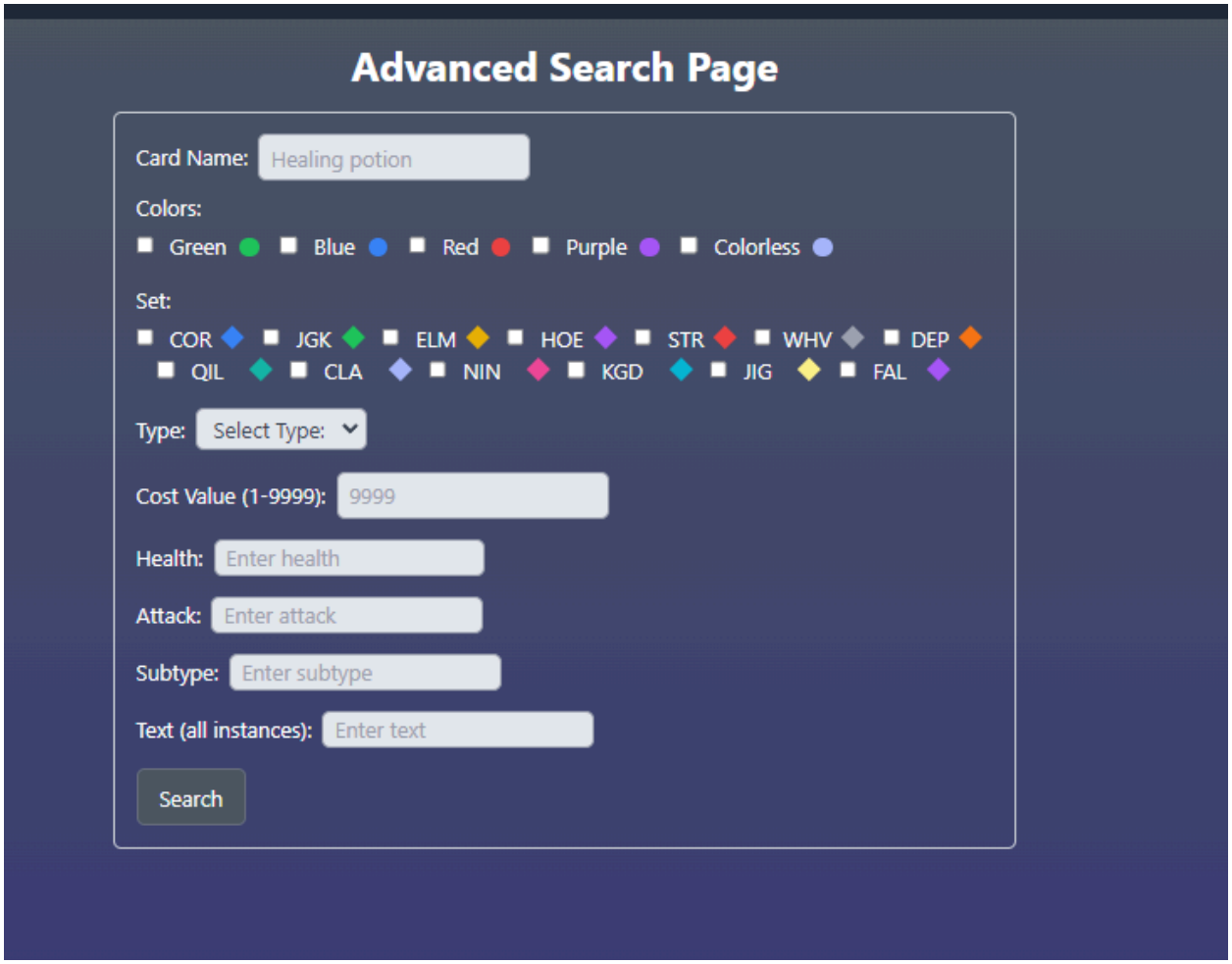
Search, Random Card Generator



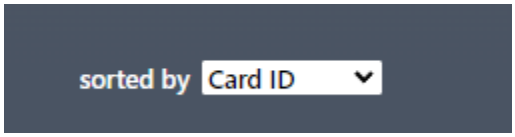
Full Card Display



Advanced Search



Ordering



Detailed Card Pages

Future Expansion

Due to time constraints, the lower priority system of a built-in deck assembling tool was cut from the current version. This would be one of the first expansions in a future update. A detailed explanation of how this feature was intended to work can be found under Major Design Decisions.

Additionally, all images are currently stored within the repository. This is a short term solution with poor scalability, and in a future update these images would be moved to an external image sharing site.

Further quality features can also be implemented, such as additional card information on individual card pages, displaying card type symbols next to cards in the search results, and having a more robust set of advanced search options.

Conclusions

Within the scope of the assignment and the allotted time constraints, we believe this project to have adequately met the basic needs of the user. As a relatively simple application that fits a very specific niche, it is unlikely to receive particularly dramatic updates or require constant upkeep. Still, it serves as a powerful proof of concept for other projects we may work on in the future.

One major takeaway from this project is the need to have a very specific understanding of the application requirements in order to plan a reasonable workload schedule. Our initial

understanding of the project scope was skewed by a lack of information about the necessary size of the data sample. While other groups most likely were able to randomly generate batch data to fill their databases with arbitrary code, all of the data used in this application is preexisting in an alternate format which needed to be converted manually. This resulted in a stall in our work pipeline as it was difficult to test the backend schemas without proper data, and difficult to test frontend visuals without images.