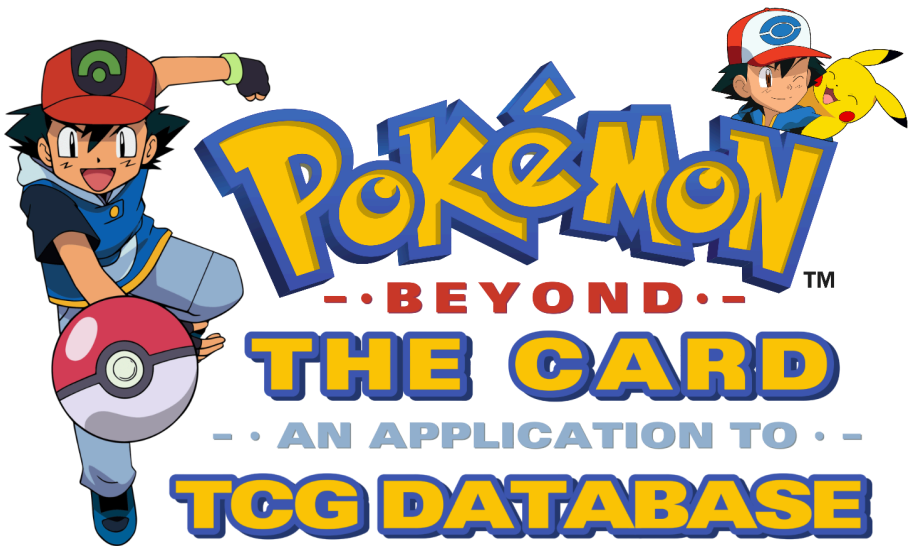


Beyond the Cards: An Application to Pokemon TCG Database

Final Report

Chengyi Li, Lewei Lin, Zhanyang Sun

Viterbi School of Engineering, University of Southern California

DSCI 551: Foundations of Data Management

Wensheng Wu

May 3rd, 2024

Google Drive Link: https://drive.google.com/drive/u/0/folders/16JtygDCJrsP1ZK89Lu88IB0jRn9szPUb

YouTube: https://youtu.be/97F56pkNTC8

Introduction

This project was created when noticing a gap in resources for those interested in Pokémon TCG-related usage. There are a few Pokémon card search websites out there, but none of them are comprehensive enough or make their data available so that others can use it as well. Therefore, we aim to develop a system for managing and querying Pokémon data using a web-based interface, providing an easy way to access Pokémon information, including images. The system is designed to cater to various user roles, including administrators, database managers, and general users, each with specific capabilities and permissions. The project utilizes Streamlit for the front end, enabling a dynamic and interactive user experience. It also uses MongoDB as the backend database to store a wide variety of Pokémon-related information, such as details about Pokémon and their associated cards.

Planned Implementation

Since our database will not have a fixed schema and will contain mixed data types for certain fields, we chose MongoDB to build our distributed database. We will have 5 tables: *Pokemon*, *Pokemon_Card*, *Type*, *Trainer*, and *Attack*. We plan to scrape the relevant information from different websites and use Python to create a JSON file storing the data for each table. The relationships among tables will be formed when generating the JSON files and linking the data. For instance, *Type_ID*, the primary key for the *Type* table, will also be the foreign key in the *Pokemon* table. We will write some codes to ensure such corresponding field matches.

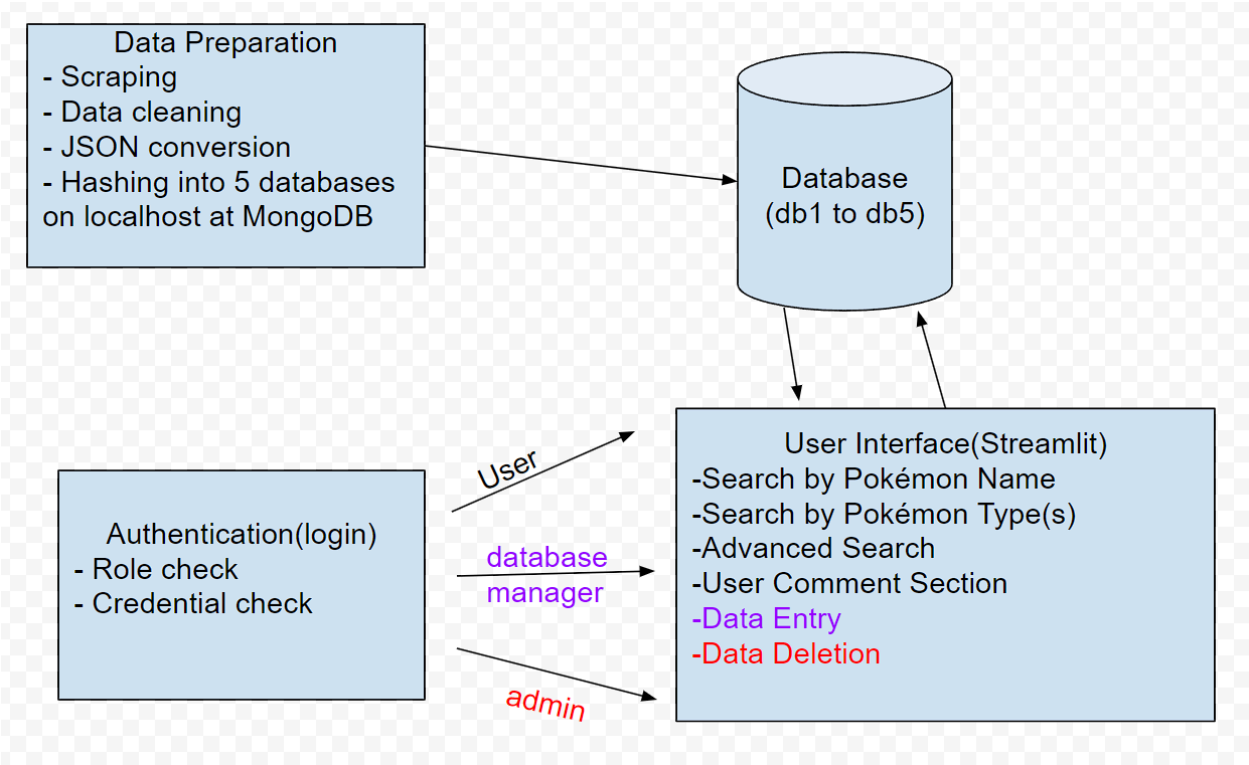
We will hash our data into 5 smaller databases on localhost at MongoDB based on the primary key of each collection. To be specific, we will convert the primary key into its string value and calculate the sum of the ASCII code for each of the characters. Then, we will take the modulus of the number of databases we have, which is five in this case. This hashing logic will generate the ID of the database that the data should be sent into, which will evenly distribute all the data across our local databases.

There will be 3 different roles of access to our database and website: “user”, “admin”, and “database_manager”. When populating our distributed database with our data, we will create these roles and grant their corresponding privileges as well. For the backend, we will use

Pymongo to develop some functions to retrieve data, including search by Pokémon names, filter by types, and advanced, multi-field search. Moreover, we will design some functions allowing data managers and administrators for additional access such as updating or inserting data into the database by tables. Furthermore, we will develop a function granting database administrator access to delete some data entries. We also plan to add field constraints and value constraints to ensure the integrity and efficiency of our database and enhance the search experience.

For the front end, we plan to use Streamlit to develop the web application and UI design, deploying it via an API. This will include, but not be limited to, providing instructions on the usage of search boxes, notifying users if the input is invalid, displaying search results, etc.

Architecture Design – Flow Diagram



Data Preparation: We scraped relevant information from several websites and saved them as either CSV or XLSX files. Then, we wrote some script for data cleaning and manipulation, converting those raw data files into five JSON data files we needed for the project. We used the JSON library to load the data and populate them into the MongoDB local databases using Pymongo. Next, we used a customized hashing function to distribute the data into five databases.

Three roles with their corresponding privileges were also created with assigned usernames and passwords.

Authentication: Users need to log into the system with different preassigned roles ("user", "database_manager", and "admin"). The system checks credentials before granting access to certain functionalities.

User Interface: Users interact with the system through a web interface built with Streamlit. Depending on their role, users can search, insert, update, or delete data. Functions like *display_pokemon_and_card_info*, *display_filtered_pokemon_card*s, *display_advance_search, db_entry*, and *delete_entry* let users interact with the database based on their needs.

Database Operations: The backend operations are handled using MongoDB with Pymongo, where data regarding Pokémon, their cards, attacks, types, and trainers is stored and manipulated. These functionalities are also realized by Pymongo and all data is stored in their corresponding collections in MongoDB. Finally, users can send different requests from the Streamlit app to manipulate or see different views of the data. Streamlit will provide good visualizations of these selected data while ensuring smooth and sound user interactions.

Implementation – Functionalities

User Login and Role Management: Different roles have different permissions, controlling what each user can do within the system.
Functions used in ***project.py***:

- handle_login()

Helper functions:

- **Streamlit Checkbox**: At the beginning of the UI, users can select which info of Pokemon and Pokemon cards they want to check.
- **Paginator**: Divide search results into manageable pages, improving the application's scalability when dealing with large data sets.
  Functions used in *main.py*:

- initialize_paginator_state(key)
- paginator(label, items, items_per_page=10, unique_identifier=None)

Search by Pokémon Name: Users can search Pokémon and their associated cards by entering Pokémon names. We have designed this function to achieve fuzzy search, allowing the input to be case-insensitive, partial match, and will automatically strip out white spaces. A message such as “No Pokémon found with the name '8'.” will display if no results are returned.

Functions used in *main.py*:

- display_pokemon_and_card_info(pokemon_name)
- display_pokemon_cards(pokemon_name, cards)

Functions used from *project.py*:

- search_pokemon_info(pokemon_name)
- search_by_name(pokemon_name)

Search by Pokémon Type(s): Users can select up to two Pokémon types, and an error message will return if users choose more than two types. The result will return all Pokémons match the type(s) and Pokémon’s corresponding cards. One thing to notice is that the cards may not match the type(s) we chose, since the types here are only based on Pokémon’s information.

Functions used in *main.py*:

- display_filtered_pokemon_cards(selected_types)
- display_pokemon_info(pokemon_name)
- display_pokemon_cards(pokemon_name, cards)

Functions used from *project.py*:

- filter_by_type(selected_types)

Advanced Search: This function combines multiple filters to refine search results, such as Pokémon Card’s Type, Illustrator Name, HP Range, and Retreat Cost Range. It will return all the Pokémon Cards that match the input search criteria, grouping by Pokémon. One thing we need to notice is that Illustrator Name needs exact matching. For your convenience, I will provide some names here for you if you want to test this function: Ken Sugimori, Ayaka Yoshida.

Functions used in *main.py*:

- display_advance_search()

Functions used from *project.py*:

- advanced_search(selected_types, selected_illustrators, hp_range, retreat_cost_range)

User Comment Section: This section allows users to leave recommendations or feedback when interacting with our application, enhancing user engagement and learning improvement opportunities.

Functions used in *main.py*:

- comment_section()
- load_comments()
- save_comment(comment)

Data Insertion and Update (database manager & admin): The DB manager and Admin can insert or update data in the database, ensuring that the information remains current and accurate. They can specifically choose which collection to modify. There are instruction prompts and error messages to guide them to correctly insert or update a data entry. The "*Create JSON*" button will display its input in JSON format for our database, and they can choose to "*Download JSON*" if necessary.

Insert & Update Constraints: We have built several functions and constraints for each field's data type and value ranges. For instance, *Type_ID* has to be a single integer of a comma-separated list of integers, ranging from 1 to 19 since there are currently 19 types in total for Pokémons, but only 11 types for Pokémon Cards specifically. If the input is not valid, an error message will display, acknowledging users what are incorrect. For example, for HP, users are not allowed to type in letters/words but only digits/numbers. If the value exceeds the set limit, it will display: "HP must be between 0 and 350."

Functions used in *main.py*:

- db_entry()
- validate_data(inputs, collection_name) → **for setting constraints*
- get_input(field_type)
- input_complex_list(label)

Functions used from *project.py*:

- upsert_data(collection_name, document, databases)

Data Deletion (only admin): Admins can specific entries to be removed from the database using the unique identifier of the selected collection.

Functions used in *main.py*:

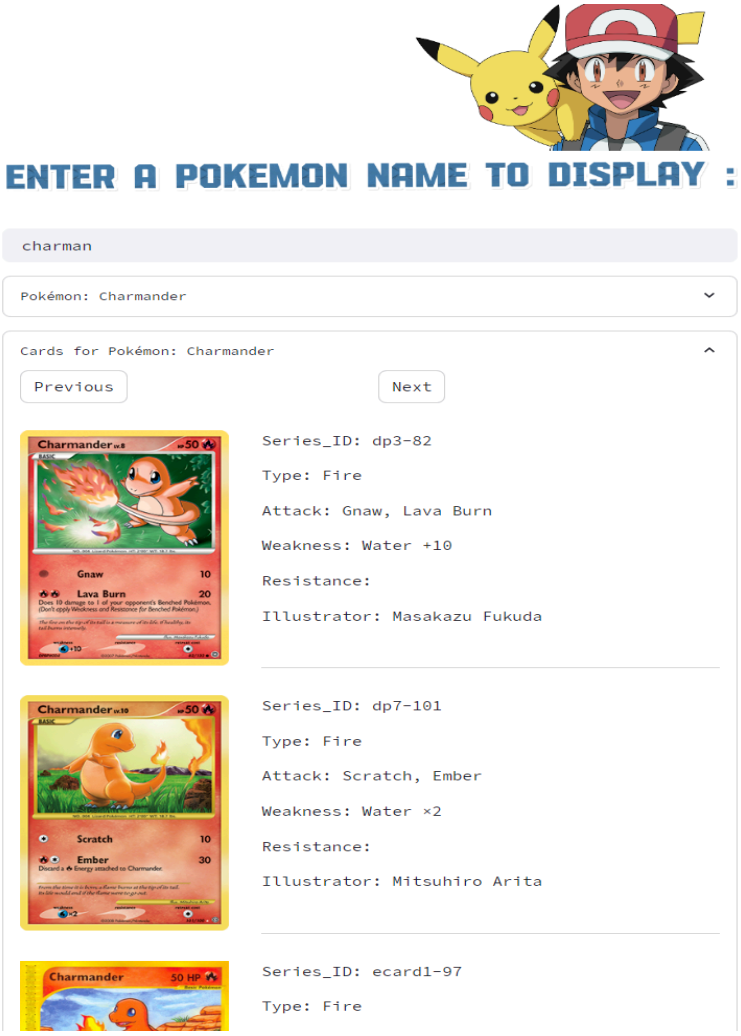
- delete_entry()

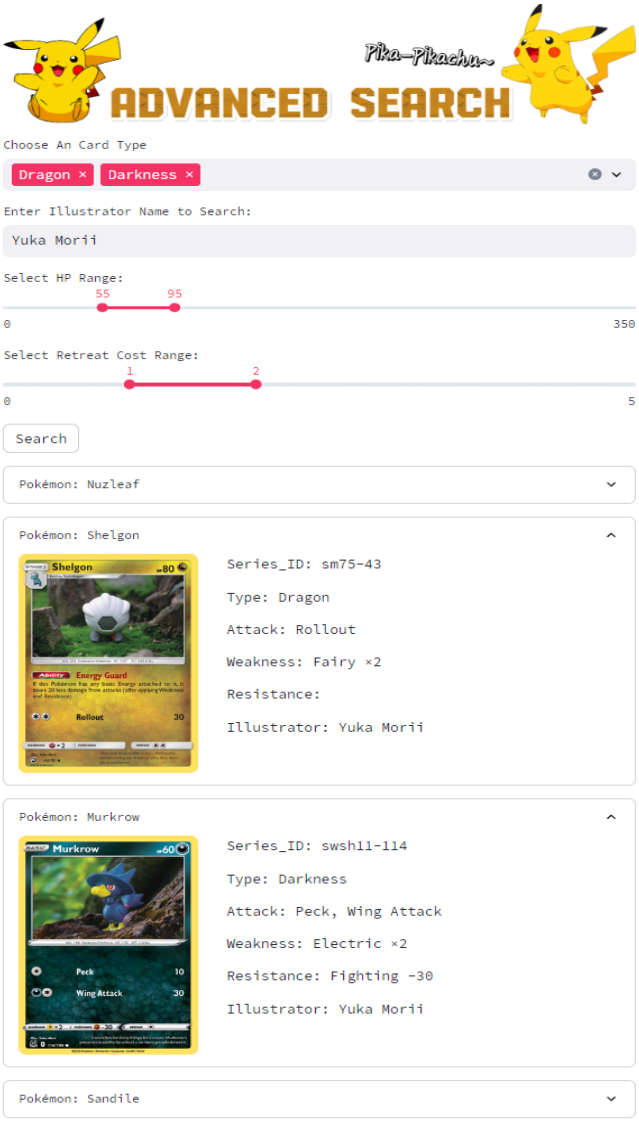
Implementation – Tech Stack

- **Front-end**: Streamlit in Python, used for creating the user interface.
- **Back-end**: Python scripts to define functions for data manipulation.
- **Database**: MongoDB, utilized for storing and querying data.
- **Libraries and APIs**: PyMongo for MongoDB integration, JSON for data manipulation.

Please continue to the next page for Implementation Screenshots…

Implementation Screenshots





Picture on the left shows the ‘search by Pokemon name’ function can have fuzzy matching, also we have a paginator to display cards on several pages.

Picture on the right shows the ‘advanced search’ function. It has four filters.

MongoDB Data Entry

Choose the collection to modify:

Pokemon

Enter Pokemon_ID:

2222

Enter Pokemon_Name:

USC

Enter Height:

15.50

Enter Weight:

66.60

Enter BMI:

10.00

Enter Type_ID:

6

Enter Trainer_ID:

10,20,30

Enter Region:

Kalos, Hoenn

Enter img_link:

https://images.pokemontcg.io/


```
{
  "Pokemon_ID" : 2222
  "Pokemon_Name" : "USC"
  "Height" : 15.5
  "Weight" : 66.6
  "BMI" : 10
  "Type_ID" : [
    0 : 6
  ]
  "Trainer_ID" : [
    0 : 10
    1 : 20
    2 : 30
  ]
  "Region" : [
    0 : "Kalos"
    1 : "Hoenn"
  ]
  "img_link" : "https://images.pokemontcg.io/"
}
```


Download JSON

ENTER A POKEMON NAME TO DISPLAY :

USC

Pokémon: USC

Height: 15.5m

Weight: 66.6kg

BMI: 10.0

Type: Ice

Trainer: Amarys, Avery, Blue

Region: Kalos, Hoenn

Super Effective Against: Grass, Ground, Flying, Dragon

Not Effective Against: Water, Ice, Metal

Weakness: Fire, Fighting, Rock, Metal

For a sample implementation of ‘data entry’ function, I inserted an entry to the *Pokemon* table.

To prove the card has already been inserted into our database, We can use the ‘search by Pokemon name’ function again. We can notice that the Pokemon named ‘USC’ is already in our database.

Learning Outcomes

Full-stack Development: Gained practical experience in developing database systems with both front-end and back-end components. Learned how to effectively use Streamlit to build user interfaces and how to manipulate MongoDB databases through Python.

Role-Based Access Control: Implemented a system with different access levels based on user roles, enhancing understanding of user management in web applications.

Database Design and Management: Enhanced knowledge of database schemas and MongoDB operations, including creating, querying, updating, and deleting data (CRUD). Learned how to structure data effectively to support complex queries and reports.

Real-Time Data Interaction: Developed skills in handling data manipulation and visualization, providing users with channels to give feedback and other interactive capabilities.

Problem Solving and Debugging: Improved problem-solving skills through analyzing and debugging user interface problems and complex issues involving database management.

Challenges Faced

User Authentication and Session Management

We had the problem of defining privileges and ensuring that users who log in with these roles can only have designated privileges. Since we are using MongoDB localhost, users need to log in with the given username and password so that they can have access to the database with proper roles. It became a problem since users who log in through localhost will automatically have all privileges and not be restricted to the privileges we intend them to have. We eventually figured out a way to create these roles and privileges in the local admin database. Then, we used the username and password pair when initiating the connection to the local database, which could ensure that the users had the designated roles and privileges.

Switching between Name and ID for Display

In the initial creation of our JSON files for various collections, we assigned unique IDs to each record in each collection, such as *Attack_ID*, *Trainer_ID*, and *Pokemon_ID*. These IDs are

primarily used in the database as references; for example, *Pokemon_ID*, *Type_ID*, and *Attack_ID* serve as foreign keys in the *Pokemon_Card* table.

However, to enhance user experience and ensure that users can easily understand what each field represents, we prefer displaying the names corresponding to these IDs in the search results. Conversely, when users select names from our web interface—such as choosing two *Type_name*—we need to map these names back to their respective *Type_ID*s. These IDs are then used to query and return relevant results from Pokémon and Pokémon Cards where *Type_ID* serves as a foreign key.

Therefore, to facilitate such conversion between Names and IDs, we have developed several helper functions:

- **get_names_from_ids:** Converts a list of IDs to their corresponding Names.
- **get_type_details:** Retrieves detailed information about a type based on its ID.
- **get_type_ids:** Converts user-selected type names into their respective IDs.
- **get_weakness_or_resistance:** Retrieves and replaces *Type_IDs* with their corresponding *Type_name*s for either weakness or resistance data.

These functions are designed to reference IDs back to their primary tables to find the corresponding names for display, or vice versa, ensuring seamless user interaction and bridging the gap between database logic and the user interface.

Defining Constraints

To ensure the integrity of our database, we added constraints to various data fields, focusing on type specifications and value ranges. This task required a thorough understanding of our data. For numerical fields such as *Damage* (from Attack), *Height*, *Weight*, *BMI* (from Pokémon), *HP*, *Retreat_Cost*, and values for *Weakness* and *Resistance* (from Pokémon Card), we set minimum and maximum values based on the value range of data within existing databases. Particularly, we will convert *Damage*, *HP*, and values for *Weakness* and *Resistance* into strings after user input, although these fields will initially accept numbers only. Additionally, we have limited the input values for *Type_ID* to a list of numbers ranging from 1 to 19. This restriction also applies to fields such as *Super_effective_against*, *Not_effective_against,* and *No_effect*, which reference *Type_ID*. While it is possible that future developments in Pokémon and Pokémon Cards could

introduce things such as new types or exceptionally high HP, our current constraints are designed to be as reasonable as possible based on existing database inputs.

Debugging

We encountered errors that weren't syntax-related but rather stemmed from flawed logic. For instance, we observed incomplete query results initially. After checking related functions, we discovered the root cause: we need to iterate through the database twice. Initially, one loop identifies the relevant database for a Pokémon, followed by another loop to retrieve and consolidate results from each database.

Simplifying Database Structure

Initially, we created 10 tables, but in the end, we streamlined our database to include only five essential ones. We eliminated redundant tables such as Pokemon_has_ability, Trainers_has_Pokemon, Region, and a few others from our initial MySQL ER design, aiming for a more concise and focused database structure and schema.

Individual Contribution

- **Chengyi Li** (Front-end Developer):

  Building the user interface for the web application using Streamlit, understanding the data processing logic, and ensuring the front-end accurately reflects the back-end's capabilities. Also ensuring that search results and data entries were displayed in a clear and organized manner. Using Streamlit’s expander and container layouts for better data visualization.

- **Zhanyang Sun** (Back-end Developer):

  Handling user authentication and authorization. Developing the server-side logic of the web applications with PyMongo for interactions with the MongoDB database. Managing the configuration and optimization of the distributed database system. Collaborating with the front-end developer to integrate the back-end with the user interface.

- **Lewei Lin** (UI/UX Designer and Database Administrator)

Cleaning and creating JSON files for our database. Defining functions to accomplish different types of search and filter functions, writing MongoDB queries using Pymongo in Python to retrieve relevant data from the databases. Creating illustrations for our web applications and ensuring a visually appealing user interface.

Conclusion

In this project, we created a web application for a Pokemon database with different built-in CRUD operations. We managed to develop this full-stack project with Pymongo and Streamlit from Python. We used MongoDB to store our data as the data is not very structured. We also developed our custom hashing function to ensure the data is distributed evenly across different local databases. We successfully created a visually appealing web application that demonstrates smooth data visualization and manipulation.

Future Scope

Web Application Deployment

- **Domain Purchase:** Consider purchasing a domain name for the web application to improve branding and accessibility.
- **Deployment Strategy:** Develop a deployment strategy using platforms like AWS, or Azure to make the application accessible to users.
- **Continuous Maintenance:** Plan for regular maintenance and updates to ensure the application remains secure and functional.

Data Entry Functionality Enhancement

- **Attribute Selection:** Allow users to choose whether to use ID or name for attributes like type, trainer, etc., providing more flexibility in data entry.
- **Selective Data Updating:** Allows users to update entries by providing only the attributes they want to change, rather than the whole info of this entry.

Image Handling Improvement

- **Local Image Storage:** In the current version of our project, images are provided by image links, requiring online retrieval when images need to be displayed. We can

improve image handling by downloading images into the local database, reducing dependency on online sources, and improving application performance, especially in areas with limited connectivity.

Scalability

- **Database Sharding:** Plan for database sharding to ensure scalability as the application grows and handles larger datasets. For now, we have db1 to db5, we can scale up the database by adding db6, db7, etc if needed.