USC-DSCI-551

Group: 80
Members: Chuanzhou(Austin) Zhang (USC-Id: 8165737934)
Kenneth Chan (USC-Id: 5772665598)
Ariel Martinez Birlanga (USC-Id: 5483611649)

Restaurant review application with distributed database - Final Project Report

Introduction

In Los Angeles, a city celebrated for its diverse culinary scene, navigating the plethora of dining options can be a daunting task. Our Restaurant Rating Application steps in as a simplified, accessible alternative to more complex platforms like Yelp, specifically designed to enhance the dining experience by focusing on straightforward community-based ratings and reviews.

This platform is tailored for ease of use, encouraging Los Angeles diners to actively participate by sharing their experiences without the clutter of traditional review websites. By emphasizing direct and simple feedback mechanisms, the website fosters a community-centric approach, making it an invaluable tool for anyone looking to explore dining options in the city while supporting local businesses with clear and concise feedback. Our app has three ways to represent the evaluation of a restaurant:

- **Starts**: Media score of the restaurant out of 5 (evaluates quality).
- **Number of Scores**: Number of people who voted (evaluates popularity).
- **Likes/Dislikes**: Secondary parameters to create Stars and Number of Scores.

Planned Implementation (From Project Proposal)

After reviewing the project proposal, we're confident that our completed website has successfully achieved most of its objectives. On the user front-end, our current website offers a user-friendly interface for rating restaurants in LA, closely resembling the simplified functionality of Yelp. Users can easily find essential information about restaurants and rate them with a single click, resulting in a combined rating for each establishment. However, during development, we made the decision to streamline the rating process by consolidating various aspects like ambiance, taste, service quality, and atmosphere into two main measures: overall quality and restaurant popularity.

As for administrators, our website fulfills all previously proposed functions, allowing them to add, edit, and remove restaurant listings. The clear separation between administrators and users enhances website security.

On the technical side, our user website and administrative app have completed the tasks outlined in the previous report. We've effectively utilized two Firebase instances to ensure efficiency in our operations. Despite deploying the website on Netlify instead of AWS as initially suggested, we've maintained flexibility in scalability. Similar to AWS, Netlify offers real-time web traffic monitoring and allows website owners to adjust resources based on traffic changes. This ensures our website remains adaptable to varying demands and maintains optimal performance.

Architecture Design

Our project is divided into three sections: a JavaScript based website (***App.jsx***), and 2 python scripts (***Foody.py*** & ***Manager.py***) used for the Login and admin interface (*Foody.exe* is also available for windows). Similarly, we have a total of 3 databases, one being exclusively used to store the Login data of users and admins, and two used for our real scalable data in which this project focuses, restaurant data.

Databases structure

Restaurant DBs

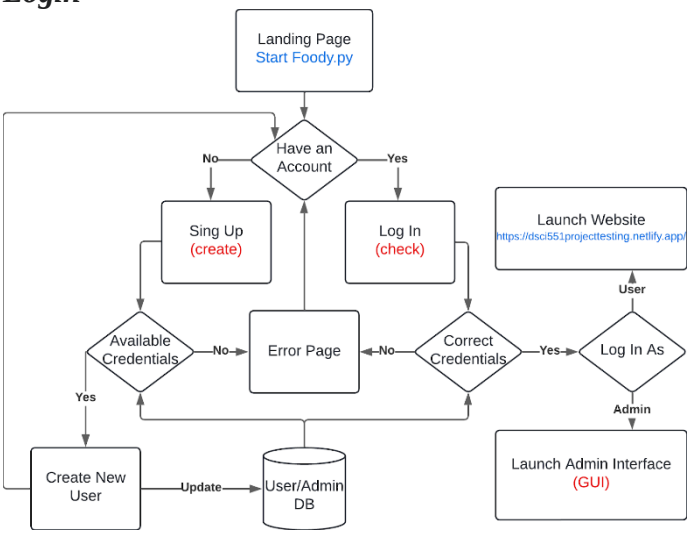
```
Retaurant{
ID:
        {
        "Name": str,
        "Score":
                {
                "Stars": num(float),
                "Likes": num(int),
                "Dislikes": num(int),
                "Num of score": num(int)
                },
        "Type of food": str,
        "Location": str)
        }
}
```

User Identification DB

```
    Login{
Admins:
        {
        Username:
                {
                Password: str
                }
        }
Users:
        {
        Username:
                {
                Password: str
                }
        }
}
```

Although both users and admins will be able to visualize the data in the Restaurant databases, users will only be able to modify/interact with ***score*** parameters. Contrarily and to make the database "consistent", administrators can not alter ***score*** parameters.

Flow Diagram - Login

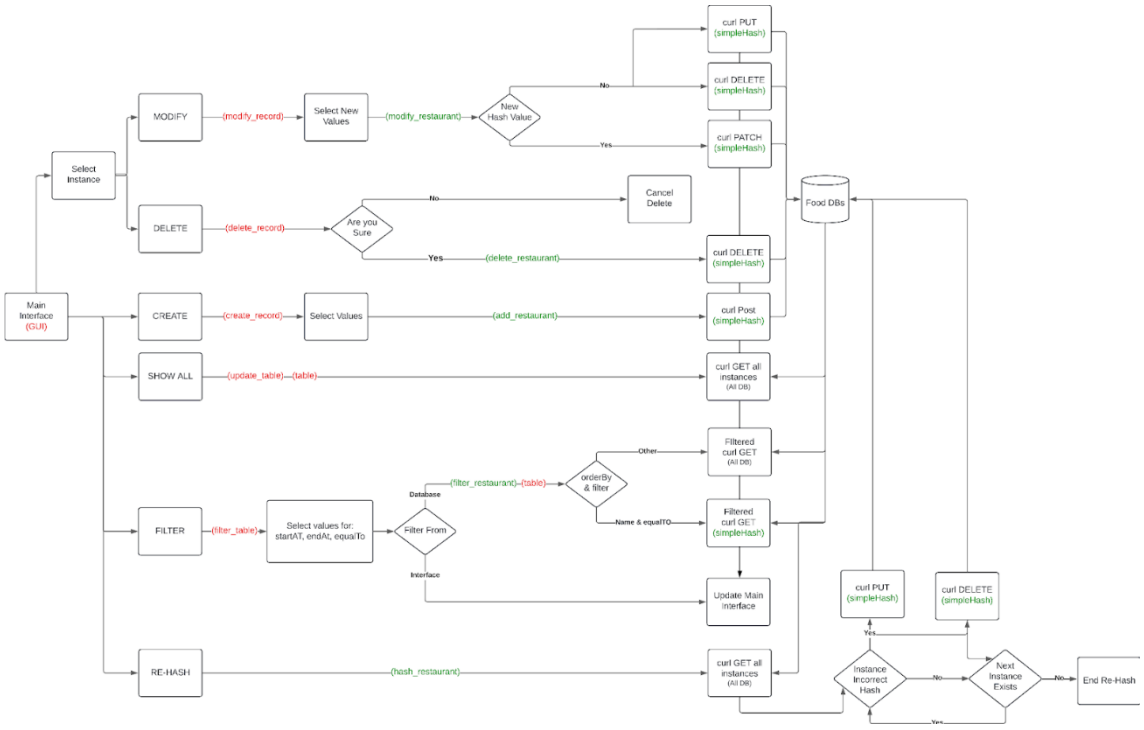


The first action anyone (user or admin) should do before interacting with the database, is to execute *Foody.exe* or the *Foody.py*. This will open a Login window in which you can write your username, password, and status (User or Admin), and select either "Sign Up" or "Log In".

The "Sign Up" (**create** submodule), will check with the dedicated User/Admin credentials database to check if the username exists. If available it will update the database with the new credentials, otherwise, an error window will pop up. Another error window will appear if you try to create an admin account.

For the "Log In" (**check** submodule), this will check that the username and password is correct (an error message will appear otherwise). Login as an user will open the web application, whereas login as admin will open a graphical interface (**GUI** module).

Flow Diagram - Admin Interface



This second Flow Diagram represents the structure of our Graphical User Interface for Admins. In red you can see the primary modules of the *Foody.py* script and in green the ones from *Manager.py*. The main interface is composed of a (filtered) view of the database (empty before Filter/Show All), and 6 buttons corresponding to the crud system and hashing: Modify, Delete, Create, Show All, Filter, and Re-Hash. To use Modify and Delete you will need to select (click) in one of the instances shown in the database view.

Create and **Modify** will open a window to insert the desired new values for the instance. These new values will be added to the corresponding database (only access 1 database) via POST and PATCH operations respectively. In case **Modify** changes the hash value, it will use a DELETE and PUT operation to change the location (database) of the instance.

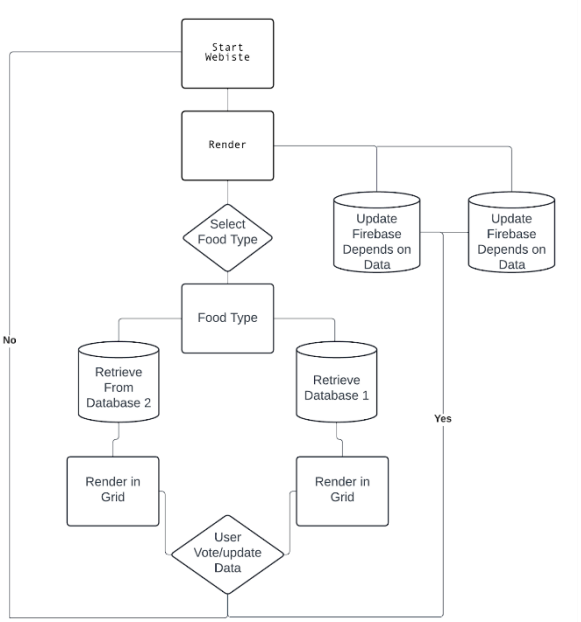
Delete is the simplest as it will only ask you for confirmation and proceed to use a DELETE operation in the corresponding database. Similarly, **Show All** will simply send a GET operation to all restaurant databases to receive all the instances.

Filter will open a window to ask about the filtering values, if you select to filter only on the values already present in the interface, the interface instances will be updated without interacting with the database. Otherwise, if you select to filter the whole database, it will access (GET) all the databases to search for the wanted instances (if "Name" and equalTo are selected it will only access one database as it will know its hash value).

Finally, **Re-Hash** will look if all the current instances in the interface are correctly hashed, searching in each database (GET) and in case of an incorrect hashed instance it will change its database host (Delete & PUT). This method is thought for when new

databases are added, since the hash method will be slightly altered, old instances might suddenly turn to be incorrectly hashed.

Flow Diagram - User Website



- Initial Rendering:
    - When a user accesses the "Foody Website" in their browser, the HTML file loads, initiating the rendering process. The App.jsx component is the main entry point for the website and is rendered within the element defined in the HTML file.

- Fetching Data from Firebase:
    - Upon rendering, the useEffect hook in App.jsx is triggered. The component checks if the typeOfFood state variable has been set (i.e., if the user has selected a type of food). If a type of food is selected, the component queries Firebase databases (db1 and db2) to fetch restaurant data corresponding to the selected type of food.

- Processing Data:
    - The fetched restaurant data undergoes processing within the useEffect hook. This processing includes sorting the data alphabetically by restaurant name and adding additional properties such as id, db, and userData to prepare it for display in the grid.

- Rendering UI:
    - The processed restaurant data is then rendered in the <AgGridReact> component, which displays the restaurant information in a grid format. Additionally, a <select> element is rendered above the grid, allowing users to choose a type of food to filter the displayed restaurants.

- User Interaction:
    - Users can interact with the grid by checking/unchecking checkboxes in the "like" and "dislike" columns to indicate their preferences for each restaurant. These interactions trigger the onEdit function, which updates the corresponding data in Firebase databases (db1 or db2). The onEdit function also updates the userData state variable, reflecting the user's ratings for each restaurant.
- Updating UI:
    - After user interactions, the UI is updated to reflect the changes in the grid. The <AgGridReact> component re-renders with the updated restaurant data, showing the latest ratings provided by users.
- End of Process:
    - The process continues as users interact with the website, providing ratings and exploring restaurant listings. The App.jsx component remains active throughout the user's session, facilitating data retrieval, processing, and UI updates as needed.

Implementation

In the following sections we are going to explain how the different parts of our code works, the process done by the implemented functionalities, and the tech stack used in the project.

Full code in: https://github.com/ambirlanga/DSCI551_Project.git

Functionalities - Login (Foody.py)

Firstly, when you execute the app (*Foody.py* or *Foody.exe*) a Tkinter will automatically pop up with several widgets: 2 labels (username and password), 1 checkbutton (check to log as admin) and 2 buttons (log in and sing up).

Log In

The Log In button will be bind to a submodule (check) that first will check if you want to log in as an admin (checkbutton). If selected it will send a GET operation to get the list of username:password tuples from the Admin section in the "User Identification DB". First it will look if an instance with the username written in the first label exists, and afterwards if the passwords correspond to what is written on the second one. If the data is correct, it **will open the Admin Interface**, otherwise an error message will pop up. Backtracking a bit, if we do not select the admin checkbox, it will follow the same process but with the User section of the "User Identification DB", **opening the User Website** if correct (will close *Foody.py*) and showing an error window otherwise.

Sign Up

Similarly, Sing Up will be bind to another submodule (create). This will check all the usernames present in the User section of the "User Identification DB" (We do not want people singing as an Admin, therefore an error window will appear if tried). If the username exists an error window will be displayed, contrarily, if available the new instance will be inserted, and a confirmation message will appear as confirmation.

Functionalities - Admin Interface (Foody.py + Manager.py)

The Admin Interface is composed of two scripts: *Foody.py* (shared with the Log In), used for visualization and user (admin) interaction, and *Manger.py*, used for elaborate interactions with the "Restaurants DBs". After login as an administrator, you will be presented with the primary interface window (*tkinter_root*) present in the GUI module, composed of 6 buttons (listed in the architecture) and a central screen (*tkinter_tree*) used to visualize the database (empty before filtering). Now let us talk about how each button is implemented.

Create

- create_record (*input: tkinter_root, tkinter_tree*). Launched from a button bind. Opens a secondary window with 1 confirmation button 3 writable labels for the user to introduce the desired "Name", "Location" and "Type of food" of the new restaurant instance (remember that modifying "Score" data is exclusive for End User). After confirmation it will launch *add_restaurant* to insert the instance and obtain the id. If successful it will also update the central screen with the new instance (without any additional requests).
- add_restaurant (*input: name, foodtype, location*)(*output: status_code, id*). External module that creates an instance with the input data ("Score" data are all set to 0). The instance will be hashed (*simpleHash*) and inserting it to the corresponding database via a POST request. Will return the status code and the id of the instance (0 if an error occurs).

Modify

- modify_record (*input: tkinter_root, tkinter_tree, [currentValues]*): Launched from a button bind in which you must previously select (click) an instance of the tree view in order to enable the button. Opens a secondary window with the same structure as in *create_record* (with the only addition of showing the current values of the instance besides the label name). When all the desired parameters have been modified (leaving a label empty will keep their current value) and the confirmation button has been pressed, *modify_restauran* will be launched to modify the database. If successful it will also update the central screen with the modified instance (without any additional requests).
- modify_restaurant (*input: id, NEWname, NEWfoodtype, NEWlocation. Name, stars, likes, dislikes, numberScores*)(*output: status_code, id*): External module that starts by comparing the hash value (*simplehash*) of the old and new names. If the hash value is the same, the function will simply send a PATCH request with all the modified values to the corresponding database. If the hash value differs. It will send a PUT request to the new database to insert the modified instance, and a DELETE request to the old database to remove the obsolete version. Will return the status code of the PUT/PATCH and the id of the instance.

Delete

- delete_record (*input: tkinter_root, tkinter_tree, [currentValues]*). Launched from a button bind in which you must previously select (click) an instance of the tree view in order to enable the button. Opens a secondary window with a readable label asking if you are sure of deleting the instance (shows its ID), and 2 buttons for either cancelation (will close the secondary window and nothing will happen) or confirmation. will open *delete_restaurant*). If confirmation is selected and

successful, it will also update the central screen by deleting instances (without any additional requests).

- delete_restaurant (*input: id, name*)(*output: status_code*). External module that will send a DELETE request to one database (*simpleHash*) for the instance with the selected "id".

Filter

- filter_table (*input: tkinter_root, tkinter_tree*): Launched from a button bind. Opens a secondary window used to extract filtered data from the database and make it visible in the *tkinter_tree*. First you can select from which column to filter (Name, Location, Stars…) from a multiple option box. Below it there are writable label to input the values for the "equalTo", "startAt", and "endAt" firebase parameters (writing in equalTo will disable the other two), where if one is left empty, it will not be used in the query. Finally, you can select (via buttons) to filter the data from the database (*full_click* submodule) or from the current tkinter_tree (*current_click* submodule).
    - Full_click: Creates the query (String) and sends it to the *filter_restaurant* module. If the column "Name" was used and the filter "equalTo" was used, a boolean set to "True" and the value of the equalTo will be sent alongside the query to *filter_restaurant*, otherwise "False" and "None" will be sent. The returned instances will be inserted by calling the *table* module.
        - hash_restaurant (*input: boolean, name, query*)(*output: {filteredInstances}, status_code*): External function used to extract filtered instances from the restaurant databases. If the input boolean is set to "True", it will make use of the hash value (*simpleHash*) of the input "Name" and send the filtered GET request to just one database. If set to "False", it will send the filtered GET request to all restaurant databases.
        - table (*input: {filteredInstances}, tkinter_tree*): Iterates through all the input instances and inserts them one by one to the *tkinter_tree* (previous values are removed from the view).
    - Current_click: Filters the current instances of the *tkinter_tree* (no call is made to firebase). For the selected column values, use '<' and '>' to filter if the "startAt" and "endAt" options are used, and '"==' and '!=' if "equalTo" is used. If an instance does not satisfy the filter, it will be deleted on the spot.

Show All

- Update_table (*input: tkinter_tree*): Launched from a button bind. Sends a general (unfiltered) GET request to all the restaurant databases and calls the table module to insert them in the *tree_view*. Button used in case you want a full view of the database.

Re-Hash

This module is thought to be used only when new databases are added and therefore old instances' hash values are redefined. The binding submodule for the Re-Hash button will extract the values of all the instances in *tkinter_tree* in a loop, and for each instance the *hash_restaurand* module will be executed to check its hash value reallocating the instance if necessary.

- hash_restaurant (*input: id, name, foodtype, location, stars, like, dislikes numberScores*): External module that searches for the instance with the input id. If the database in which is found corresponds to its current hash value (simpleHash) do nothing, otherwise, delete the instance from the database it was found and send a PUT request with that same instance data to the database that corresponds to the new hash.

simpleHash (*Manager.py*)

- simpleHash (*input: name*): Used in all the external (*Manager.py*) methods mentioned in this section. Returns the hash value of a restaurant based on the lowercase ascii value of the name's first letter ('109' being the current bar line). An 'elif' will be added for every extra database, and in case they grow to be at least one database per letter (lowercase 97-122), we will start analyzing also the ascii value of the second character.

Functionalities - User Website

Let's delve into each part of the **App.jsx** code, highlighting the implementation specifics, functionalities, and integrations.

React Component Structure and Hooks

The component utilizes **React Hooks**, specifically **useState** and **useEffect**, which are fundamental for state management and lifecycle operations in functional components. The **useState** hook initializes the **userData** state variable, an array that stores user data fetched from Firebase. This state is crucial for reactive UI updates, as changes in state trigger re-renders of the component.

The useEffect hook is employed to fetch data when the component mounts. This is performed by setting up a Firebase query that listens for real-time updates to user data, specifically targeting users flagged as 'active'. The onValue listener responds to any changes in the specified Firebase path, ensuring the UI reflects current data without manual refreshes.

Firebase Database Interactions

Firebase functions facilitate real-time data interactions. The **ref** function specifies the database path, **query** constructs queries based on specific conditions, and **onValue** sets up a persistent listener that invokes a callback with the current snapshot of the data:

- **ref(db1, 'users')**: Defines a reference to the 'users' node in the first database configuration (**db1**).
- **query(..., orderByChild('active'), equalTo(true))**: Orders data by the 'active' child and filters for entries where 'active' equals true.
- **onValue(userQuery, callback)**: Listens for real-time updates, ensuring the component always displays up-to-date data.

AG Grid Integration

AG Grid is integrated to render user data effectively. The **AgGridReact** component is used with several key properties:

- **rowData**: This property is bound to the **userData** state, ensuring the grid displays the latest data.
- **columnDefs**: Defines the structure of the grid columns, which would typically include configurations like header names, field bindings, and properties like sortable and filterable, tailored to the data's needs.
- **defaultColDef**: Sets default properties for all columns to maintain consistency and reduce redundancy in definitions.

Utility Functions for Local Storage

Utility functions saveToLocalStorage and getFromLocalStorage enhance the user experience by providing persistence of certain application states across sessions:

- **saveToLocalStorage(key, value)**: Saves data to the browser's local storage, useful for settings or data that should persist when navigating away from the page.
- **getFromLocalStorage(key)**: Retrieves data from local storage, allowing the application to restore user preferences or states after a page reload.

Dark Mode Detection

The application adapts to the user's system preferences for themes, particularly dark mode, enhancing accessibility and user comfort:

```
const isDarkMode = window.matchMedia("(prefers-color-scheme: dark)").matches;
```

This line checks if the user prefers dark mode and can be used to toggle themes dynamically in the application, ensuring it adheres to preferred visual settings.

Importing Firebase Configurations

The import of multiple Firebase configurations indicates the application's ability to switch between different data sources or environments seamlessly:

```
import { database as db1 } from "../firebase.db1.config";

import { database as db2 } from "../firebase.db2.config";
```

This capability is critical for applications that operate across different environments, such as development, testing, and production, enabling developers to manage various datasets or configurations without changing the codebase.

Each segment of the App.jsx code not only showcases how modern web applications handle state and real-time data efficiently but also demonstrates best practices in user interface rendering and preference management. This meticulous approach ensures a robust, user-friendly, and maintainable web application.

Key functionalities of the web application:

Select Filter Functionality

- **Purpose**: Allows users to filter restaurants based on specific criteria like type of food.
- **Implementation**: Utilizes React to capture the filter input from the user interface, then queries Firebase to retrieve restaurants that match the selected criteria.
- **Display**: The filtered results are dynamically displayed using AG Grid, updating the grid to show only the restaurants that meet the filtering conditions.

Add Like Functionality

- **Purpose**: Enables users to express positive feedback for a restaurant.
- **Interaction**: When a user clicks the "like" button, the system checks if the user has previously disliked the restaurant.
- **Database Update**: If previously disliked, it decrements the **dislikes** count and increments the **likes**. It updates the **numScore** and recalculates the **ratings** based on the new totals.
- **UI Feedback**: Displays the updated like count and recalculated ratings instantly, reflecting the user's input.

Add Dislike Functionality

- **Purpose**: Allows users to express negative feedback for a restaurant.
- **Interaction**: Similar to the "like" functionality, clicking "dislike" will decrement the **likes** count if the restaurant was previously liked.
- **Database Update**: Increments the **dislikes** count, updates the **numScore** by subtracting appropriately, and recalculates the **ratings**.
- **UI Feedback**: The UI updates to show the new dislike count and adjusted ratings, ensuring real-time feedback is visible to the user.

Exclusive Choice Rule

- **Purpose**: Prevents users from having both a "like" and a "dislike" active for the same restaurant simultaneously.
- **Implementation**: Before updating the user's new choice, the system checks for an existing opposite reaction and reverses it if present, ensuring users can only have one active feedback type per restaurant.
- **Effect**: This rule maintains the integrity of the feedback system, ensuring that data reflects clear user preferences.

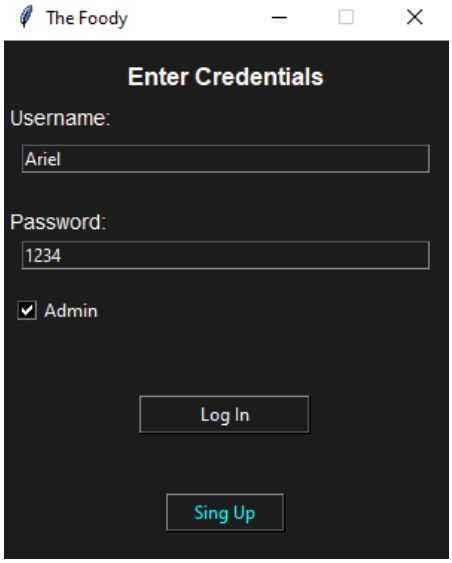
Updating numScore and Ratings/Stars

- **numScore Calculation**: Defined as the total feedback, calculated by adding the number of dislikes to the number of likes.
- **Ratings Calculation**: Uses a predefined formula to convert the net scores into a star rating or similar metric, which could consider both the number of likes and dislikes. This is done by dividing the net positive feedback (Likes - Dislikes) from the total feedback (numscore).

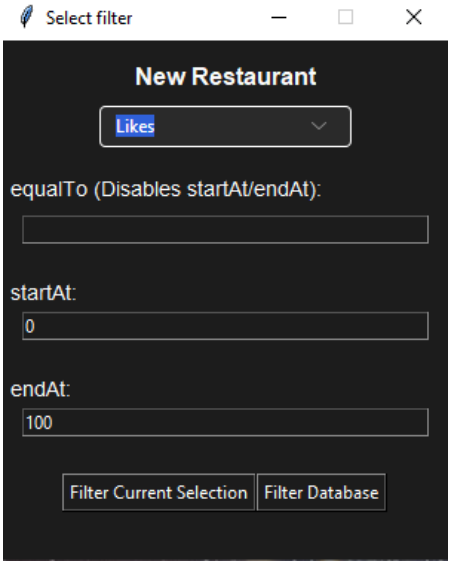
- **Dynamic Updates**: Both **numScore** and ratings are recalculated in real-time upon any change in likes or dislikes, using Firebase transactions to ensure data consistency.

Implementation Screenshots

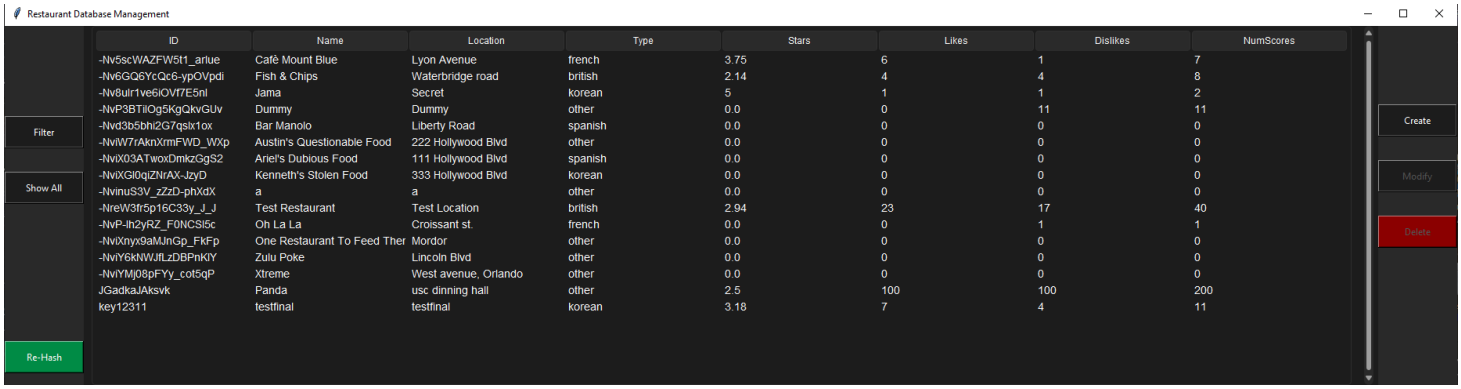
Log In



Admin Interface - Filter



Admin Interface – Primary Window



Foody Web Application

Tech Stack - General

Programming Languages

- **Python**: Language used for Login and Admin Interface (*Foody.py* & *Manager.py*).
- **JavaScript**: Language used for the End User Website.

Database

- **Firebase**: 2 firebase databases were used to work with our restaurant data (with the possibility of scaling the number), and a secondary third database was used to store the usernames and passwords of administrators and users.

Login and Admin Interface: Libraries

- **Request & json (Database Connectivity)**: Request gave us the possibility to send the different curl operations to the database, whereas the json library allowed us to correctly encode and decode the data from and to firebase.
- **Tkinter & sv_ttk (Interface)**: Tkinter was the framework we used to graphically visualize the Login and Admin Interface, which include a variety of widgets like buttons and trees (table) to provide interaction and visualization. Sv_ttk is a simple external Tkinter library used to create the 'dark theme' used in our interface (improve aesthetics).
- **Sys & webbrowser (Website Connectivity)**: Minor libraries used to close the Login window and open the User Website. Sys was also used to test manager.py methods.

Tech Stack - Website

Frontend Technologies

- **React**: Utilized to build dynamic user interfaces, offering efficient rendering and state management for responsive web applications.
- **AG Grid**: Provides a powerful grid system for displaying and manipulating large datasets, supporting functionalities like sorting and filtering.

Styling

- **Tailwind CSS**: Applied for rapid UI development, leveraging utility-first classes that simplify the application of styles and enhance design consistency.

Development and Build Tools

- **Vite**: A modern build tool that speeds up development with features like fast refresh and optimized builds.
- **ESLint**: Ensures code quality and consistency, helping to identify and correct issues in JavaScript code.
- **PostCSS**: Extends CSS capabilities, used here to integrate with Tailwind CSS for advanced styling options.

Package Management

- **npm**: Manages dependencies, streamlining the setup and update process for various libraries and frameworks used in the project.

Learning Outcomes

This project has served as a great learning experience in database management (particularly firebase). More specifically, we have learnt how to use different tools like 'Tkinter' or 'React' to integrate and visualize a database for different types of access levels (users and admins). This, however, did not come without any challenges.

Challenges faced

During the development of the Restaurant Rating Website, our team faced significant challenges, particularly in mastering new web development tools such as React, Firebase, and Tailwind CSS, which were not covered in our academic coursework. This required intensive self-study and practical application to integrate effectively. Additionally, combining these self-taught tools with those learned in class posed integration challenges, necessitating meticulous planning and problem-solving to ensure seamless functionality and user experience across the website. Similarly, as we detailed in the Mid Progress Report, 'Tkinter' also posed a challenge, requiring more self-study and practice to understand how this library works. These hurdles tested our adaptability and expanded our technical prowess beyond the classroom.

Individual Contribution

The following list shows in which segment of the project (*Foody.py*, *Manager.py*, and Website) and/or which module of the same each of use has worked on:

- Chuanzhou(Austin) Zhang:
  - *Manager.py*: modules(`delete_restaurant`)
  - Website(*App.jsx*): Complete implementation (collaboration with Kenneth)

- Kenneth Chan:
  - *Manager.py*: modules(`add_restaurant`)
  - Website(*App.jsx*): Complete implementation (in collaboration with Austin)

- Ariel Martinez Birlanga:
  - *Manager.py*: modules(`simpleHash, modify_restaurant, hash_Restaurant, filter_restaurant`)
  - *Foody.py*: Complete implementation*

*Mention to Keneth lines(330-370) (no actual implementation of the modules).

Conclusion

As with everything, there is very little you can do by learning only the theory about things. Therefore, a real case implementation for database management and integration as this project serves as an excellent experience for us to understand all of the concepts we learnt in class as well as to broaden our view about the possibilities and challenges present in this type of project.

Overall, we think that we manage to accomplish our initial idea ("a simplified, accessible alternative to more complex platforms like Yelp") and integrate all the desired functionalities. On a more critical side, we also feel that while the options available for an administrator are quite varied and complete, the user website options still have room to grow in variety and aesthetics.

Future Scope

To further enhance the Restaurant Rating Website and maximize its impact for users, the top three improvements that could be considered are:

1. **Advanced Filtering and Search Capabilities**: Refining the search functionality to include detailed filters such as price range, dietary preferences (e.g., vegan, gluten-free), and ambiance can help users find restaurants that meet their specific needs more precisely. This would improve user satisfaction by tailoring the search results to individual preferences.
2. **Geolocation Features**: Integrating geolocation technology would enable the website to offer restaurant suggestions based on the user's current location. This feature would be particularly useful for users exploring dining options on the go, providing them with immediate and relevant recommendations.
3. **Website Design Enhancement**: Improving the aesthetic and functional aspects of the website design to make it more visually appealing and user-friendly. This could include modernizing the layout, using more intuitive design elements, ensuring the user interface is engaging and easy to navigate, and also making more functionalities in the website. A well-designed website can significantly enhance the user experience, encouraging longer visits and more frequent interactions.