Software Design Document
for
Eureka Eats

Version 1.2.2 approved

Prepared by Devin Chang, Derek Tan, Dana Mendoza, Iqra Irfan, David Tabor

11/23/2023

Revision History

Name	Date	Reason For Changes	Version
Derek Tan	11/10/23	Moved JSON message notes from SRD, fixed outdated information for code style.	1.1.0
Devin Chang	11/15/23	Updated restaurant Database components.	1.1.1
Derek Tan	11/17/23	Updated obsolete database notes, references, and application API codes.	1.1.2
Derek Tan	11/23/23	Updated JSON API notes, Section 10 module table, and reference list.	1.1.3
Derek Tan	11/28/23	Updated JSON API notes for restaurants. Fixed some typos.	1.1.4
Derek Tan	11/29/23	Updated Sections 6 and 7 for backend module(s). **The PSPECs are TBD.**	1.2.0
Devin Chang	11/30/23	Added 2.1 Assumption and Dependencies Added 7.6, 7.7	1.2.1
Iqra Irfan	12/1/23	Added frontend documentation for Sections 6, 7.	1.2.2
David Tabor	12/2/23	Changed DFD 0 to Context diagram. Fixed grammatical errors throughout.	1.2.3

1. Introduction

.

1.1 Purpose

Our website EurekaEats, release number 1.1.1, shows our website users the best-fit restaurants based on the user's preferences. Website users can review, rate, and find restaurants. Each search will connect with our restaurant database and display its content in a visually pleasing way. The SDD will give a clear understanding of the relations between the different software used for the project and the overall functionalities of our website.

1.2 Document Conventions

See the *Glossary(Module 11)* for conventional terms used by the team.

1.3 Intended Audience and Reading Suggestions

Developers assigned to this project will need to read every part of the SDD to better understand creating the web application.
Stakeholders and project managers must review the introduction, flow diagram, and user interface to understand what the product will do.

1.4 System Overview

The website EurekaEats will be hosted on a developer's computer(for now). The website will collect inputs, including mouse clicks and keyboard entries(through key input fields). These inputs will be matched with our database to output specific events and fields. The database will be hosted on a developer's computer till we upload it into the cloud.

2. Design Considerations

2.1 Assumptions and Dependencies

Our system is assumed to run on a local host computer for the current production and deployment stage. Our system will depend on that computer to be running continuously so that our website will be available to the public. Our system depends on the data pulled from Yelp Fusion API for displaying restaurants. For our map-based navigation, the system depends on the user accepting the terms to allow our system to pull their location. A map is generated from the Mapbox API using this location.

Required Hardware

Based on the use case of *EurekaEats*, we assume that the required hardware must be able to host website software. Specifically, it must run a web client, web server, and database system on top of the underlying operating system. This is true for both development and deployment releases.

Required Software

The required software for the project covers multiple areas: programming utilities, libraries, frameworks, and network tools. Programming utilities include languages or programs necessary for the development and operation of the website. Referenced from the requirements document, Python 3.12, NodeJS, Yelp Fusion API, Mapbox API, and MongoDB Community Edition 7.0 for our runtime languages, external APIs, and database. Libraries and frameworks include third-party code that our project requires to operate. The specific choices of third-party code include Python Flask for backend programming, PyMongo database driver for MongoDB, PyTest for unit testing code, and ReactJS for the frontend framework. Finally, network tools are other applications used to view the results of the *EurekaEats* website's functioning. Specifically, a modern browser such as Google Chrome, Microsoft Edge, or Firefox is required to make web requests to the server. Yelp Fusion API is where we will pull most of the restaurant information.

Operating System

Websites rely on web technologies, which usually do not rely on underlying OS-specific or hardware-specific features. Following this reasoning, the choice of operating system does not matter except for the requirement that it must be able to show GUIs and graphics to render webpages.

End-User Characteristics

From the purpose of the project given by requirements, EurekaEats is a useful tool for people to find choices that best fit their dining preferences. This implies the core assumption that end-users of the website have previously struggled with other similar services, such as Yelp or Google Places, to find their best-fit restaurants. However, end-users of the website will have various roles and skill levels organized in four major categories below:

- Casual Eater:
    - New User: Does not fully understand the UI and should only use some basic functions of the website: searching restaurants, customizing preferences, and leaving simple reviews.
    - Familiar User: Understands the UI and uses all basic functions of the website on top of what new users utilize. They should also leave restaurant ratings and occasionally feedback about the website.
- Food Critic:
    - Power User: Fully understands the UI and should use more advanced features of the service. They may not only leave reviews but should also enter discussions with other fellow users over opinions of various dining locations.
- Restaurant Owner:
    - Power User: Unlike the Food Critic, an owner should usually not leave restaurant reviews, although the review functionality will allow this. Neither should owners enter discussions with their patrons about their business or services. However, they should be able to report misleading or inappropriate content to admins.
- Administrator:

    - Uber User: This user will be a developer or owner of the website. They can moderate content and manipulate any data stored and managed by the website. However, they are not required to interact with less technical users or do casual site usage, such as reviewing, searching, or other features, unless during end testing.

Anticipated Functional Changes

None

2.2 General Constraints

The constraint that might affect our future release of the project is the amount of traffic for our website. We do not know how much traffic the website can handle before overloading. We will need to stress test our product to see how much traffic the website can handle and, if needed, move the host to a different server to meet these requirements.

We have to design our website to comply with region-specific laws. The main types of laws depend on the laws of a nation, state, or province, and they can affect how data is stored and processed.

The Website is designed to be viewed from a desktop/laptop. Looking through a mobile device might have unintended visuals and clunky user interfaces.

2.3 Goals and Guidelines

- Software Goals:
    - The website must have the correct functionality: no crashes or errors, and outputs should be correct.
    - Finish a working version by the last week of the semester. This is because the demonstration of the software is mandatory.
    - The website must keep sensitive data secure. This is because data breaches of user data can harm them or add risk for more harmful actions.
    - Finally, the website must be easy to understand when being used. This is because people prefer easier solutions to their problems.

- Programming Principles:
    - DRY (“Don’t repeat yourself”): This should reduce the codebase size to reduce mental overhead when making any changes. Also, this implies that code should use modular and reusable parts to not be repetitive.
    - CAT (“Code always tested”): This is essential in the Agile part of our development process. Tests (manual or automatic) must be made to ensure parts of the software function correctly. This also saves struggles in debugging when bugs are caught early.

2.4 Development Methods

We use a version of Agile development to develop our project. We have an idea of how to accomplish our project, so we separate these ideas into planning increments and then place them into our sprint backlog. Programming, testing, and reviewing are done concurrently during our sprint.

3. Architectural Strategies

Architectural Patterns:

- Client-Server model: the clients relay user actions to an application server, which processes their requests to provide necessary services.
    - Reason 1: All web applications follow this architectural pattern by default. Also, separating the user interfaces of the client from business logic improves code modularity and maintenance.
    - Reason 2: This model is highly accessible to remote users accessing the World Wide Web.
    - Reason 3: Popular languages familiar to the development team have professional web client-server libraries and frameworks. They also have libraries for interacting with a database such as MongoDB.

    - Reason 4: The protocols used by clients and the server to communicate are mature and well-tested: HTTP, etc.
    - Tradeoff: At the cost of introducing another risk factor (network condition), this architectural pattern was used for convention of web development and better organization of code. Also, reinventing a new pattern would be an excessive effort.
- Future Plans:
    - Research and implement more security.
        - Reason 1: This will be pushed to the future as it is costly to our limited development team.
        - Tradeoff: Not securing the site means we must be very careful about who can access it, or we could leak sensitive data.

4. System Architecture

4.1 Logical View

Functionalities pertaining to end-users.

- Select Eating Preferences
- Log In
- Sign Up
- Search for restaurant
- Click on Feature Restaurant
- Display Restaurant information
- Review Restaurant
- Favorite Restaurant

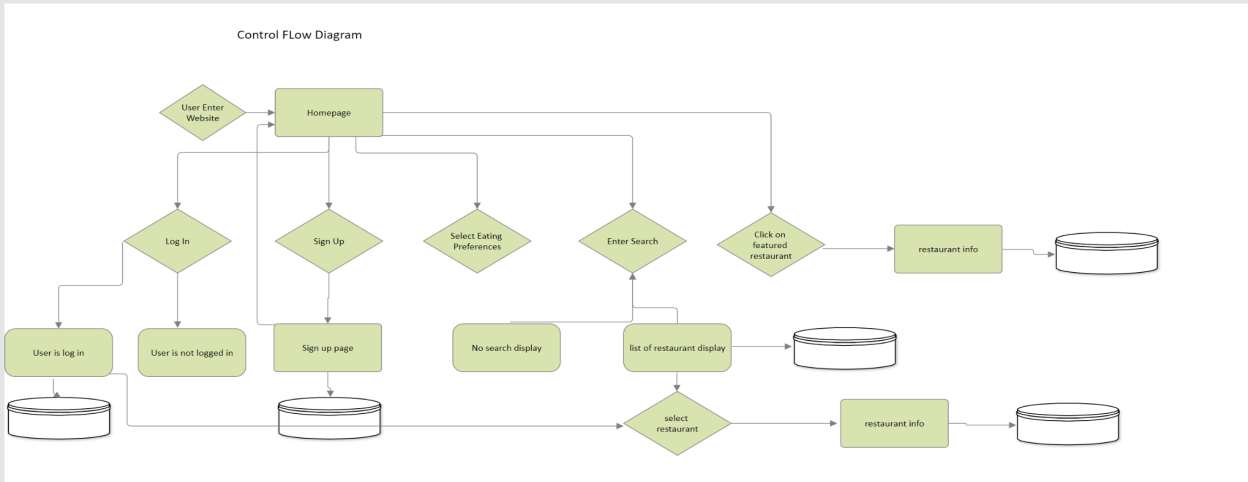


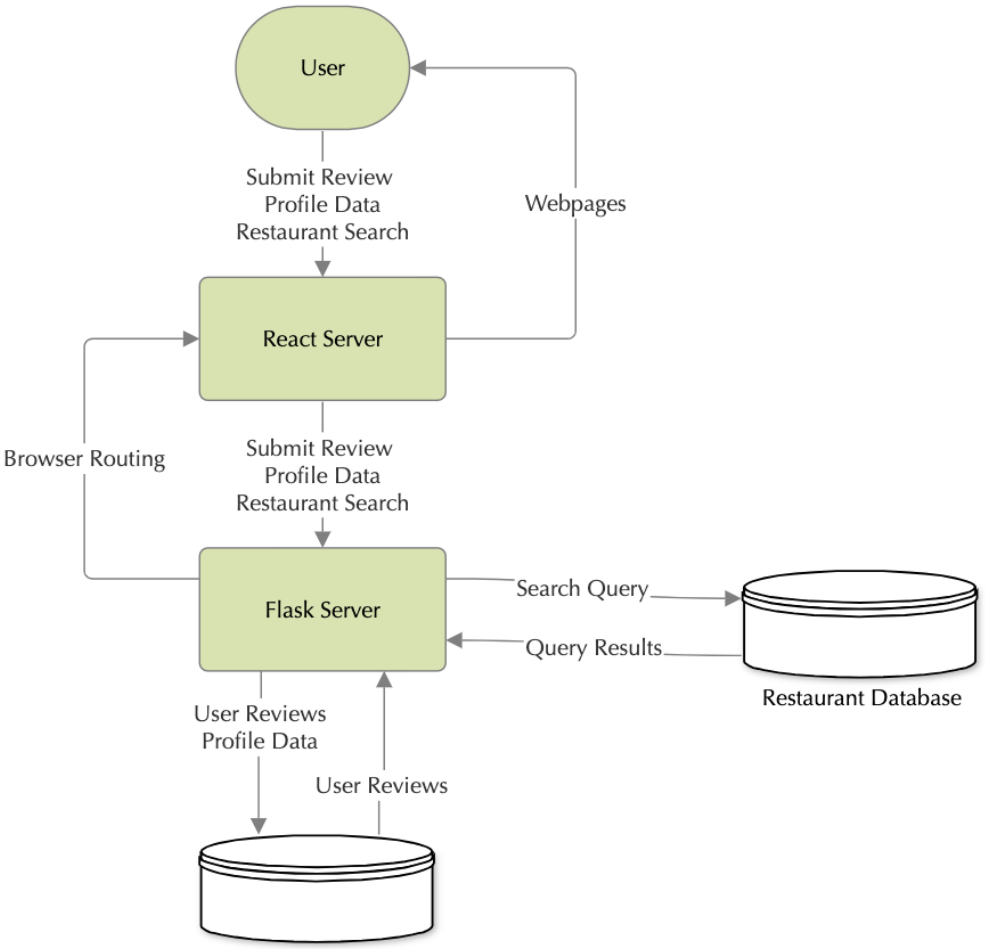
Figure: Control Flow Diagram

4.2 Development View

Figure: Context Diagram

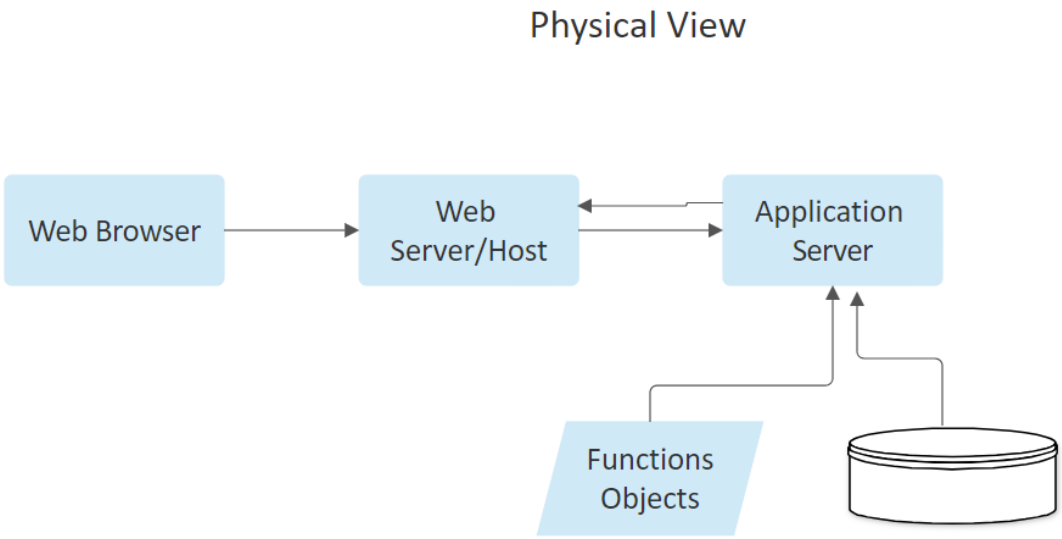
The figure above shows how the Eureka Eats system will interact with external APIs. Restaurant data, such as the restaurant name, location, and food type, will be taken from the Yelp API.

4.3 Process View



4.4 Physical view

Interaction between applications



5. Policies and Tactics

5.1 Choice of which specific products used

- Interpreters: Python (3.12), NodeJS (latest version)
  - Why: Easy to create, build, and test software because of easy syntax, easy start of execution, and an abundant ecosystem of libraries or frameworks.
- Database: MongoDB (Community Version 7.0)
  - Why: It is easy to change the structure of database entries versus relational databases such as SQL.
- Test Browsers: Google Chrome, Microsoft Edge, Firefox (all latest versions)
  - Why: The vast majority of online users for the website should use at least one of these browsers.

5.2 Plans for ensuring requirements traceability

- General Routine:
  - 1: A team member completes a work item under a feature where that story is assigned.
  - 2: The work item is moved to *In Review*.
  - 3: Present development team members will meet and discuss the changes with the author to verify if they completed the required feature/functionality.

    - 4: The work item remains in the In Progress section of the backlog if the required work is incomplete or invalid. The author must make additional edits to their incoming changes. If valid, the work item is moved to Completed.

5.3 Plans for testing the software

- See STP for EurekaEats

5.4 Engineering Guidelines

- TBD

5.5 Coding guidelines and conventions

- Commenting on important code is required.
- Naming all Python variables, functions, and classes or objects according to PEP 8 guidelines is required.
    - Indentation and naming are consistently used.
- JavaScript code style:
    - Use semicolons.
    - Indent consistently per file.
    - Use *Egyptian Brackets* for statement blocks.
    - No global variables. Use *let* or *const* keywords.
    - Use the === operator over the regular == operator.
- Each function or class must only fulfill one role.
- Handle exceptions.

5.6 Protocols

- JSON App Protocol over HTTP
    - **NOTE:** The *application* API Codes are organized by groups of up to 16 possible actions per part: restaurants, reviews, and users. This is to anticipate future changes.
    - -1: Unknown or invalid action code.
    - 0: Search restaurant(s).
        - Arguments: restaurant cuisine or price level. A token argument may become a later requirement for client verification.
    - 1: Retrieve data for a single restaurant.
        - Arguments: exact restaurant ID. A token argument may become a later requirement for client verification.

    - Note: This usually happens when a special restaurant page link is opened: `/page?id=<string ID>`
- 2: Suggest cuisine names.
    - Arguments: first N letters of a name.
    - Note: cuisine suggestions must appear below or near the search bar as the user types!
- 3 to 15: **Unspecified for future use.**
- 16: Search for reviews.
- 17: Create a review for a restaurant by a valid user.
- 18: Update an existing review for only the author.
- 19: Delete an existing review for only the author.
- 20 to 31: Unspecified for future use.
- 32: Get user information.
    - Private Info Arguments: username and session cookie for authorizing a complete fetch.
    - Public Info Arguments: the session cookie is null. Only gets the username and user summary ().
- 33: Update user profile.
    - Arguments: A JSON object with keys of profile setting names to the new setting values. Values include username, email, password, and summary.
- 34: Create a user account.
    - Arguments: The username, email, password, and repeated password for confirmation. All values here are required.
- 35: Delete a user account.
    - Arguments: The username string and password for confirmation are both necessary.
- 36: Log in to a user account.
    - Arguments: The username and password are both necessary.
- 37: Log out of a user account.
    - Arguments: The username and password for confirmation.
- 38 - 46: Unspecified for future use. (Intended to become admin actions.)

- 47: Dummy API call for testing. Does nothing but say “Hello World!” and needs no arguments.

Message Payload Type Codes:

- 0: number
- 1: string
- 2: object
- 3: boolean
- 4: null (no defined value)

Application Message Data:

- Format: JSON text
- App Operations: (usually for API requests to the EurekaEats backend)
  - `{ “action”: number, “args”: <object | array>}`
- App Data: (usually a response to an EurekaEats API call)
  - `{ “payload”: number, “data”: <object | array>}`

5.7: Maintaining the Software

- The EurekaEats website software will continuously integrate changes during development and beyond if possible. This is primarily achieved with Pull Requests to frame proposed changes for the main release branch. The author of a Pull Request must receive approval from most fellow team members:
  - Static analysis of code: code review by formatting, style, and overall logic.

5.10: System Building and Running Instructions (Development)

1. Install Python 3.12, Git, and VSCode if not found on the machine.
   a. VSCode requires Python Extensions for a friendlier development experience.
   b. Clone the GitHub repository locally using Git: `git clone <repo url>` where the Code Menu gives the repository URL on the GitHub page.
2. Install all Python dependencies for the backend with the command `python3 -m pip install -r requirements.txt`
3. Run the command `flask --app eureka run` to launch the backend development server.

    a. Use the `--debug` option after the "run" verb within the command above to restart the development server automatically on code changes.
4. Run the command `cd ./eurekaeats` in another terminal to enter the frontend folder.
5. Install all JavaScript dependencies for the frontend with the command `npm install`
6. Run the frontend portion of the project with the command `npm run start`
    a. The website should appear on the default browser installed.

5.11: Maintenance Plans

- Each development team member should follow these basic actions during maintenance:
    - First, make code changes and remember to test its operation.
    - Second, push the changes to a *development* repository branch.
    - Third, create a pull request for mandatory peer review based on code style and logic.
    - Finally, accept or reject the changes after discussion.

5.12: Source Code File Structure

- **Folder** `eureka`**:**
    - Folder "api": Contains separate Python Flask modules for handling frontend requests by URLs, JSON content, etc.
    - Folder "mockdata": Contains hard-coded objects to mock external data from other software components. This can change over time.
    - Folder "utils": Contains loaded environment variables by *python-dotenv* and utility wrapper class for the PyMongo *MongoClient*.
    - *__init__.py* file: contains backend server startup code. Sets route handlers and exit handlers after setting up the Flask application object. The exit handler using the *"atexit"* Python module gracefully closes the MongoDB connection for the Flask server.
    - *config.py* file: contains global constant variables read by Flask to configure the application. The only setting possibly needed is the *DEBUG* flag set to *True* for easing development.
- **Folder** `eurekaeats` **:**

    - Contains a default ReactJS project file structure based on this reference: [Getting Started | Create React App (create-react-app.dev)]
    - Folder *"src"*:
        - Contains page assets (images, icons, CSS) in *assets/ and public/*, React components each in named folders, and the main files: *App.jsx*, *index.js*.
        - Other JavaScript test files such as *App.test.js*, *reportWebVitals.js, and setupTests.js* were meant for JavaScript unit testing using the Jest testing Framework, but they are still currently unused.
- **Folder `tests` :**
    - Contains unit tests for the Flask application. Currently, the unit tests are outdated and need to be rewritten to test application action and database transaction functions.

6. Detailed System Design

6.1 Eureka (Backend Module)

6.1.1 Responsibilities

Primarily, the Eureka website component has the responsibilities of processing HTTP requests from the frontend according to the semantics of *Section 5.6*. Upon processing any request valid by the app protocol, this component will run an important application action to update the database and any other application state. The specific types of action categories include all crucial areas:

- Restaurant operations: e.g., fetching restaurant data, searching restaurants by criteria, etc.
- Reviewing operations: e.g., creating, fetching, updating, or deleting a review by any valid user.
- User actions: e.g., registering, signing in, or signing out with a user account… They also include general querying of user data and profile management.
- Admin actions to manage user content: e.g., deleting problematic user content such as spam or hostile messages. However, this area of functionality may not be implemented because of time constraints.

6.1.2 Constraints

Functional Assumptions:

- This component must be error-safe. No crashes or exceptions disrupting the processing of user actions should occur. See **Control Flow** within **Functional Constraints** for more details.
- This component must use a shared database client with a connection pool to the MongoDB server. Specifically, a *PyMongo* model object for the relevant collection is used by its methods to do important transactions to satisfy an application action.

Functional Constraints

- **Control Flow:** The component cannot throw an exception that would disrupt overall operation. This ensures that the website performs reliably and safely.
- **Role:** The component cannot render UI, as that functionality is exclusive to the front end. This follows the ideal of a single responsibility per module or submodule for keeping code maintainable and easy to develop.

Interfacing Rules:

- **Input:** The messages received by this component must be HTTP messages containing JSON content formed according to *Section 5.6*.
- **Output:** HTTP messages containing a JSON payload according to *Section 5.6*.

6.1.3 Composition

1. **"api" subcomponent:** contains submodules within the *eureka* folder: *restaurants, users,* and *reviews* are the primary submodules. Additionally, the *appcodes* submodule contains useful constants for the application API, specifically the action codes specified in *Section 5.6*. However, the *admin* submodule *may* be dropped from the codebase due to time constraints.
2. **"mockdata":** contains hardcoded data such as fake user JSON for older manual testing of the user login and logout functionalities. This submodule may remain obsolete since the database supersedes the hardcoded user data "entries".
3. **"utils":** contains two submodules. The *constants* submodule has special global constants of values loaded from a *.env dotfile* containing environment secrets such as API keys and MongoDB database credentials. The *service* submodule contains a wrapper class to abstract away details of retrieving a PyMongo model object for e.g the *users* collection in the MongoDB instance.

6.1.4 Uses/Interactions

- The frontend uses this module to handle its requests. Each request requires processing of the JSON encoded application action for a user, so this module serves as the backend for this purpose.

- The MongoDB instance of this website uses this module as the recipient of its data. The database instance alone cannot process specific business logic, so the backend Flask server formed within *eureka* is required for all business logic.

6.1.5 Resources

- **Data Resource:** The *MongoDB* instance with the database *EurekaEats*. This is a dedicated website database to manage all dynamic data such as that of users, reviews, and restaurants.
- **Computer Resources:** Enough RAM is required to run the database and Flask backend server software. The hardware must also be able to run at least one browser window in parallel with the database and application server.
- **Software Resources:**
    - **Key JavaScript Frameworks or Libraries:** React (frontend UI library), Webpack (bundler and build tool for web projects), Jest (testing).
    - **Key Python Frameworks or Libraries:** Flask (backend application framework), PyMongo (database client), PyTest (unit testing framework), *python-dotenv* (environment secret loader).

6.1.6 Interface/Exports

TBD

6.2 EurekaEats (Frontend Module)

6.2.1 Responsibilities

Responsibilities of the frontend module for the web application EurekaEats must allow users to seamlessly use the app without needing external instruction. Other responsibilities of the front end for this application include a search bar in the main header that connects to the MongoDB database to pull restaurants that match the user's search criteria (such as the restaurant's name or cuisine). The home page pulls featured restaurants from the database as well. Users can also interact with restaurants to “heart” them, add them to their favorites, and leave reviews, which are stored in the backend. The front end also provides screens for users to create an account, log into their account, and manage their account.

6.2.2 Constraints

Constraints of the front end for EurekaEats include device responsiveness so that various screen sizes can handle the website without formatting incorrectly and following general UI/UX and accessibility guidelines, such as using bright colors and contrasting text. Other constraints include following a consistent structure throughout the web application to avoid confusing the user.

6.2.3 Composition

The front-end files are contained within the folder “eurekaeats” and are built using ReactJS. Within the “eurekaeats” folder, there is a submodule, the “src” folder, that holds all the essential files that are inserted into the App.jsx file and rendered by the index.js file. The structure of the files within the “src” folder that handles various pages is given its folder with names that easily identify what those files code. For example, the “Login” folder contains the “Login.js” file and the “Login.css” file for styling. Additionally, the “src” folder contains “assets” that holds the images used throughout the web application, such as the logo.

6.2.4 Uses/Interactions

- The frontend interacts with the user to allow them to navigate through the app, utilize the various functions the application provides, and take in user inputs to store.
- The front end interacts with the backend to send over user inputs to be stored and retrieved later, such as when users want to log back into their account and view their favorites or access search results via the search bar.

6.2.5 Resources

- ReactJS Official Documentation is a primary resource to access different libraries that can be utilized to stylize the code efficiently.
- The computer used to program the frontend must be able to handle at least one browser window to view changes in the front-end code.
- Figma (website) is used to design the front end before hardcoding it into the web application.
- Key JavaScript Frameworks or Libraries: React (frontend UI library)

6.2.6 Interface/Exports

TBD

7. Detailed Lower level Component Design

7.1 Eureka API: appcodes.py

7.1.1 Classification
This is a code file containing constants for application action codes.

7.1.2 Processing Narrative (PSPEC)
TBD

7.1.3 Interface Description
No interfacing is required except for using the constants. Imports from `appcodes.py` typically include all or some of the constants an API module requires.

7.1.4 Processing Detail
N/A

7.1.4.1 Design Class Hierarchy
N/A

7.1.4.2 Restrictions/Limitations
The application action codes must follow Section 5.6: No additional codes should be defined beyond the range of 0 - 48. No constants should have a mismatched or undescriptive name relative to the action, e.g., *EE_ACTION_CODE_47* for the *dummy* action.

7.1.4.3 Performance Issues
N/A

7.1.4.4 Design Constraints
The constants must be declared in order based on the action code list within Section 5.6. This is to ease the implementation of the application action API for *EurekaEats*.

7.1.4.5 Processing Detail For Each Operation
N/A

7.2 Eureka API: restaurants.py

7.2.1 Classification
This file contains procedural logic to handle restaurant actions for *EurekaEats*.

7.2.2 Processing Narrative (PSPEC)

TBD

7.2.3 Interface Description

The main interface of this file is based on the fact that it contains a Flask blueprint instance. As a blueprint object, the restaurant action API router must fulfill communication interfacing and code usage interfacing. First, the router object attaches HTTP request handlers that receive HTTP JSON requests as their input and return HTTP JSON responses based on *Section 5.6*. Second, the router object is encapsulated in the file as a Python submodule under *eureka*, which is imported into the *main.py* file for usage in the *app* object. The *app* object will attach this router object to itself to handle all restaurant actions.

7.2.4 Processing Detail

Within this file are various stages of action processing: message validation, message unpacking, call processing, and then response forming. First, message validation happens in the `handle_restaurant_action` function. The function checks specifically for whether the request method is permitted and the content type is JSON. Second, message unpacking happens in the `restaurant_api_do` function. This function extracts the *action* and *args* fields from the received JSON for the application to determine what action API call will be done on what data. Third, call processing occurs: the given actions with arguments are dispatched to various helper functions for the required transactions. Finally, the response forming occurs when a *Python tuple* containing payload code and data is returned from a helper to the `restaurant_api_do` function. The returned data is then packed into a Python dictionary which is automatically framed as a JSON response by Flask.

7.2.4.1 Design Class Hierarchy

N/A: Code structure is procedural.

7.2.4.2 Restrictions/Limitations

By the single responsibility principle, this file cannot contain helper functions for a purpose other than processing restaurant actions.

7.2.4.3 Performance Issues

The key performance issues depend on external factors. These factors include web connection latency, database connection latency, and the efficiency of MongoDB commands. Latency in any of these three areas may degrade service. It is important to ensure that MongoDB commands are minimal: no unnecessary steps in the aggregation pipeline or read operations are done. Also, it is important to ensure that the web and database connections are stable to ensure no downtime: stalling the website's operation is detrimental to user experience.

7.2.4.4 Design Constraints

The functions must be organized in this hierarchy: HTTP receiver, action dispatcher, and action helper. Each level of the hierarchy from left to right narrows its focus from a general operation to a more specific operation, e.g the `restaurant_api_do` function

handles general call data (action code and arguments) while the `restaurant_api_search` function must only give a search result list.

7.2.4.5 Processing Detail for Each Operation
See 7.2.4 for the operation details.

7.3 Eureka API: reviews.py

7.3.1 Classification
This file contains procedural logic to handle user-reviewing actions for *EurekaEats*.

7.3.2 Processing Narrative (PSPEC)
TBD

7.3.3 Interface Description
The main interface of this file is identical to how *restaurants.py* does so. However, the actions only help users manage reviews.

7.3.4 Processing Detail
The sequence of action message processing stages is identical to how *restaurants.py* does so. However, the actions only help users manage their reviews.

7.3.4.1 Design Class Hierarchy
N/A: Code structure is procedural.

7.3.4.2 Restrictions/Limitations
By the single responsibility principle, this file cannot contain helper functions for a purpose other than processing user reviewing actions.

7.3.4.3 Performance Issues
The key performance issues depend on the same external factors as for *restaurant.py*.

7.3.4.4 Design Constraints
The design constraints are identical to those for *restaurant.py*.

7.3.4.5 Processing Detail for Each Operation
See 7.3.4 for the operation details.

7.4 Eureka API: users.py

7.4.1 Classification
This file contains procedural logic to handle user account actions for *EurekaEats*.

7.4.2 Processing Narrative (PSPEC)
TBD

7.4.3 Interface Description
The main interface of this file is identical to how *restaurants.py* does so. However, the actions only help users manage their accounts.

7.4.4 Processing Detail
The sequence of action message processing stages is identical to how *restaurants.py* does so. However, the actions only help users manage their reviews.

7.4.4.1 Design Class Hierarchy
N/A: Code structure is procedural.

7.4.4.2 Restrictions/Limitations
By the single responsibility principle, this file cannot contain helper functions for a purpose other than processing user actions.

7.4.4.3 Performance Issues
The key performance issues depend on the same external factors as for *restaurant.py*.

7.4.4.4 Design Constraints
The design constraints are identical to those for *restaurant.py*.

7.4.4.5 Processing Detail for Each Operation
See 7.4.4 for the operation details.

7.5 Eureka API: __init__.py

7.5.1 Classification
This file contains the main backend server code where execution begins.

7.5.2 Processing Narrative (PSPEC)
TBD

7.5.3 Interface Description
The interface is trivial: only a configuration Python file named as `config.py` is targeted by path as the inputted configuration. This configuration is respected by the Flask command when starting and running the application server.

7.5.4 Processing Detail
The factory function does the overall steps to start the application server. First, it consumes configuration constants from the `config.py` file. Second, it imports the pre-initialized PyMongo wrapper from the `utils/service.py` file. Finally, the blueprints imported from `eureka/api` are attached to the application instance object named *app*. This app object is then returned to the *Flask* running utility.

7.5.4.1 Design Class Hierarchy
N/A: Code structure is procedural.

7.5.4.2 Restrictions/Limitations
By the single responsibility principle, this file cannot contain helper functions for a purpose other than initializing the *Flask* application object with blueprints and configuration settings such as "debug" running mode.

7.5.4.3 Performance Issues
N/A: The file is much more trivial versus the *Eureka* API module code, so execution or memory overhead is not a concern.

7.5.4.4 Design Constraints
The setup of the *Python Flask* application sets an exit handler to close the database connection and do other cleanup before the program terminates. This is good practice to allow the OS to reclaim resources and do a graceful shutdown to minimize risk of data corruption. Also, the factory function `create_app()` does all the setup for the *Flask* app object internally to avoid having a global variable which could have increased debugging difficulty. Instead, the *flask* command implicitly uses the returned application object to run.

7.5.4.5 Processing Detail for Each Operation
See 7.4.4 for the operation details.

7.6 Eureka: mongoDB_restaurant.py

7.6.1 Classification
This code contains the methods to delete and store data in the MongoDB restaurant collection.

7.6.2 Processing Narrative (PSPEC)
Requires a MongoDB connection string. Require parameters stated in the function to add to the database.

7.6.3 Interface Description
Requires a MongoDB connection string to store the data.
Require parameters stated in the function to add to the database.

7.6.4 Processing Detail
Drop existing collection named 'restaurant.'
Store information parsed from Yelp Fusion API JSON file.

7.6.4.1 Design Class Hierarchy
N/A: File is used only to create and store the restaurant database in your local MongoDB database.

7.6.4.2 Restrictions/Limitations
Limitation depends on the amount of local host disk space.

7.6.4.3 Performance Issues
Depends on how much data is stored in the database, but current capacity is substable.

7.6.4.4 Design Constraints
Requires .env file that stores MongoDB connection string to start.

7.6.4.5 Processing Detail for Each Operation
See 7.6.4 for the operation details.

7.7 Eureka: yelp_fusion_api.py

7.7.1 Classification
This code contains the methods to create the database using data from Yelp Fusion API

7.7.2 Processing Narrative (PSPEC)
Requires a MongoDB connection string from .env file.
Requires a Yelp Fusion API key from .env file

7.7.3 Interface Description
Requires a MongoDb connection string to store the data.
Require Yelp Fusion API key send request to pull json files of restaurants for the Los Angeles area.

7.7.4 Processing Detail
Create a restaurant database with Yelp Fusion API. Restaurant information is pulled once a request with specification is sent to the Yelp server. Restaurant information is pulled as a json file and is then parsed to meet the specification of the restaurant database reference from module 8.2.

7.7.4.1 Design Class Hierarchy
N/A. The class in this file uses the classes from mongoDB.restaurant.py.

7.7.4.2 Restrictions/Limitations
Limitation depends on the Yelp server side to not display an error from a request. 500 requests can be sent a day but only one is needed to be sent upon a successful request. Restriction of request depends on the documentation of Yelp Fusion API in module 12 Reference.

7.7.4.3 Performance Issues
Performance depends on the amount of requests sent to the Yelp server. A batch of 50 restaurant information is pulled and parsed to the layout specified in the restaurant database design from module 8.2.

7.7.4.4 Design Constraints
Requires .env file that stores MongoDb connection string to start.

Requires .env file that stores Yelp Fusion API key to start.

7.7.4.5 Processing Detail for Each Operation

See 7.7.4 for the operation details.

8. Database Design

8.1 users database:

MongoDB items in **USER** database:
Field username - Username of the user. Type String
Field email - Email of the user. Type string
Field password - Password of the user. Type String
Field first_name - First name of the user. Type String
Field last_name - Last name of the user. Type String
Field location - General coordinate of the user. Type [double,double]
Field cuisine - what type of cuisine they prefer. Type String[]
Field price - the price point based on criteria. Type String
    Values: $, $$, $$$, $$$$ (based on Yelp price scale)
Field reviews - Reviews for the restaurant. Type Object
    Field restaurant_id - restaurant id for access to the restaurant database. Type int
    Field star_rating - Star rating review of the restaurant. Type int
    Field comment - User comments for the restaurant. Type String

Sample User Document:

```
{
    user_id: 1,
    username: "fakeperson123",
    password: “fakefakepass1”,
    first_name: "Fake",
    last_name : "Person",
    location: [122.222, 133.333],
    eating_preferences: [
        vegan: FALSE,
        keto: FALSE,
        gluten_free: FALSE,
    ],
    cuisine: [“Italian”, “Anglo”, “Taiwanese”],
    price:[
        low: FALSE,
        medium: FALSE,
        high: TRUE
    ],
    reviews: []
}
```

82. Restaurants Database:

MongoDB Entry Structure **Restaurant** Database**:**

- Field **_id**: MongoDb default unique id for each documents
- Field **id**: Yelp Fusion API id for the restaurant
- Field **name**: This must contain a title special to a dining location. Its type is also a non-empty string.

- Field **image_url** : a url which contain a image about the restaurant
- Field **is_closed** : is the restaurant closed down or not
- Field **review_count** : The review count of EurekaEats users for the restaurant
- Field **rating :** The rating pulled from Yelp API. Type double from 0-5.0
- Field **price** : The estimated price labeled using amount of $
- Field **location**: The location of the restaurant
    - **longitude** : The value of the longitude of the restaurant
    - **latitude** : The value of the latitude of the restaurant
- Field **address1**: The street number and name
- Field **address2**: Building number or apt number
- Field **city** : City location of the restaurant
- Field **zip_code** : zip code location of the restaurant
- Field c**ountry** : country location of the restaurant
- Field **state** : state location of the restaurant
- Field **phone** : phone number of the restaurant
- Field **type** : what the restaurant is labeled as and their cuisine
- Field **reviewers**: This must contain a list of reference IDs to any reviewers who are registered users for *EurekaEats*. This is an array type listing ObjectIDs for each critical user.
    - Changed from Project Proposal 2: We only need to list user IDs to reference reviews from any user's data. This is to keep the raw JSON as simple as needed so the database is easier to understand and maintain.


```
{
  "_id": {
    "$oid": "6552eb1e41bcd35a60d467a9"
  },
  "id": "cal0Wpupxj9c_AV7WzDXsw",
  "name": "GRANVILLE",
  "image_url":
"https://s3-media2.fl.yelpcdn.com/bphoto/znY6Wq1711cPvOtfKxWZZg/o.jpg",
  "is_closed": false,
  "review_count": 0,
  "rating": 4.5,
  "price": "$$",
  "location": {
    "longitude": -118.38068,
    "latitude": 34.0771299
  },
  "address1": "8701 Beverly Blvd",
  "address2": null,
```

```
  "city": "West Hollywood",
  "zip_code": "90048",
  "country": "US",
  "state": "CA",
  "phone": "(424) 522-5161",
  "type": [
    "New American",
    "Cocktail Bars"
  ],
  "reviewers": [
    "DoeEats"
  ],
}
```


Refer to SRD 4.3 Logical Database Requirements for more reference

9. User Interface

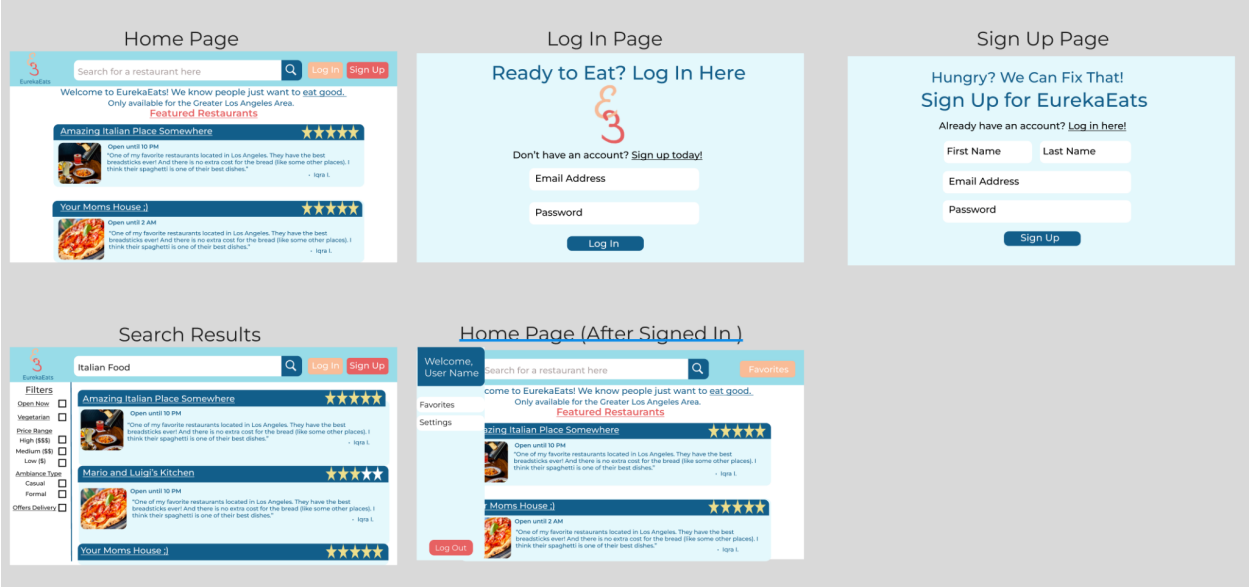
9.1 Overview of User Interface

- From a user's perspective, EurekaEats should offer a straightforward and intuitive user interface. Utilizing bright colors and large buttons will enhance user attention and reduce confusion. The inclusion of a wide range of filter options will empower users to tailor their search results. A simplified navigation menu will further reduce user confusion. The 'favorites' feature should save users time by allowing them to easily access previously preferred restaurants, streamlining their experience

9.2 UX Standards

9.2 Screen Frameworks or Images

These can be mockups or actual screenshots of the various UI screens and popups.



9.3 User Interface Flow Model

A discussion of screen objects and actions associated with those objects. This should include a flow diagram of the navigation between different pages.

Landing Page:

- search bar: connects to the backend to pull database restaurant data as specified by keywords imputed by the user
- login button: redirects to log in button
- sign in button: redirects to sign in button
- array of featured restaurants: clickable title to see more details (mockup for individual restaurants in progress)

Login Page:

- email-address input bar: takes user input, backend validates email from database
- password input bar: takes user input, backend validates from database
- login button: redirects to home page (not landing page)

Sign Up Page:

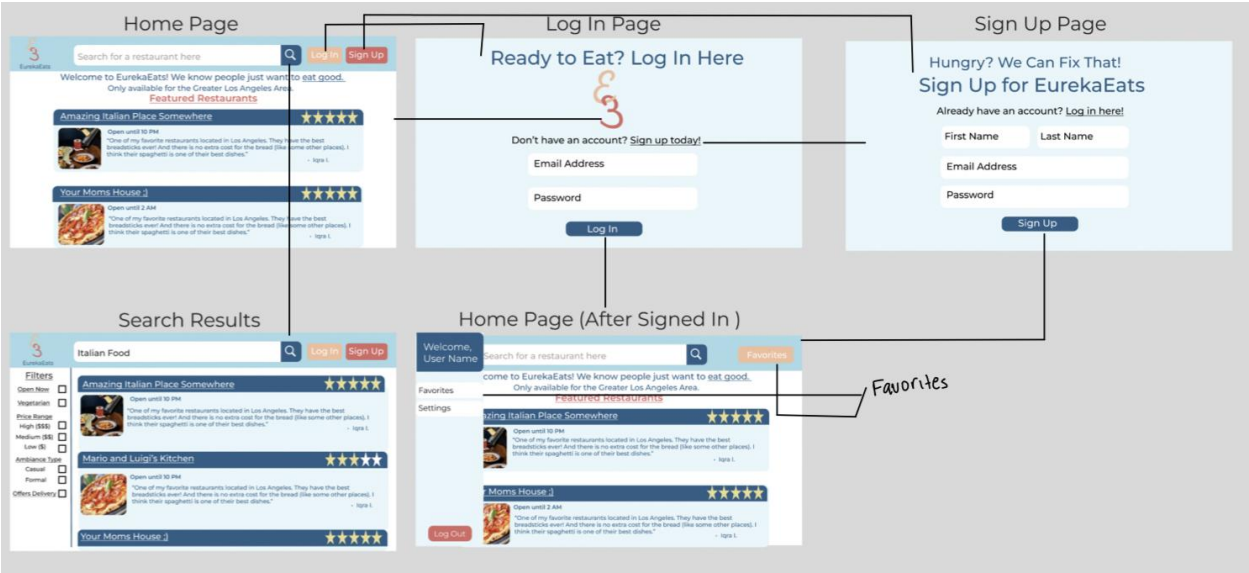
- Redirection to Login page if user already has an account
- first name/last name input bar: takes user input, backend sends to database to proceed with registration
- email address/password bar: takes user input, backend sends to database to proceed with registration (check if email has signed up before and validate password)
- sign up button: redirects to sign up page

Home Page (After Sign Up):

- favorites button: redirects to favorites page (mock up in progress)

- side menu button
    - favorites button: redirects to favorites (mock up in progress)
    - settings button: redirects to settings (mock up in progress)
    - logout button: redirects to home page, backend registers user logout event

Flow Diagram:



10. Requirements Validation and Verification

Requirements:	Module:	Testing:
login page	*eurekaeats/LogIn*	manual
sign up page	*eurekaeats/SignIn*	manual
list of restaurant	*eurekaeats/RestaurantPage*	manual
profile page	*eurekaeats/ProfilePage*	manual
home page	*eurekaeats/HomePage*	manual
landing page	*eurekaeats/LandingPage*	manual
user accounts functionality	*eureka/api/users.py*	manual / automatic
restaurant functionality	*eureka/api/restaurants.py*	manual / automatic
review functionality	*eureka/api/reviews.py*	manual / automatic

Create a table that lists each of the requirements that were specified in the SRS document for this software.
For each entry in the table list which of the Component Modules and if appropriate which UI elements and/or low level components satisfies that requirement.
For each entry describe the method for testing that the requirement has been met.

11. Glossary

- TBD: To be determined. This acronym means that the design point is not yet specified.
- N/A: Not applicable. This acronym means that the design point cannot be specified for the component or topic.
- *Possible*: Not guaranteed to be used or implemented during development, but is in consideration by the developers.

12. References

- EurekaEats GitHub Repository (by the SDD authors): csula-cs3337swe/202308Group6-repo (github.com)
- STP (Software Testing Plan): https://docs.google.com/document/d/17kt9ItAldqNdjvLxyN98njDJmlmT92YYg6CiU27Wg6g
- Planning Increment Guidelines by URL: https://scaledagileframework.com/pi-objectives/
- PyMongo MongoDB Library Documentation: https://pymongo.readthedocs.io/en/stable/tutorial.html
- Python Flask Documentation & Tutorial: https://flask.palletsprojects.com/en/3.0.x/
- Official React Dev Website by URL (by search query “create react app”): https://create-react-app.dev/docs/getting-started
- React Navigation Tutorial by URL: ://blog.logrocket.com/complete-guide-authentication-with-react-router-v6/#basic-routing-with-route
- React User Authentication Tutorial by URL: https://www.digitalocean.com/community/tutorials/how-to-add-login-authentication-to-react-applications
- Yelp Fusion API documentation https://docs.developer.yelp.com/docs/fusion-intro