MAD-II Project Report

Project Title

Abode Mantra: Your A-Z Cleaning Experts (Household Services Application) is a multi-user app (Admin, Customers and Professionals) which acts as platform for providing comprehensive home servicing and solutions.

Author

- Kavish Pal Singh
- 23f2005144
- 23f2005144@ds.study.iitm.ac.in
- About Me:
  Greetings,
  I am a Diploma-Level Student at IIT Madras pursuing a Bachelor of Science degree in Data Science and Applications. I have a strong interest in Data Science, Machine Learning, and their real-life applications. This project provided me with practical experience in full-stack development using Flask, SQL-Alchemy, JavaScript, Vue, Celery, and Redis to build a scalable data-driven application.

Description

This project extends the MAD-1 project by implementing a proper Frontend and Backend. The goal was to develop both components independently and then integrate them seamlessly using APIs. It applies the theoretical concepts I learned in the MAD-2 course to create a fully functional, end-to-end web application.

Frameworks and Libraries used

- Python==3.12.3
- Flask==3.1.0
- Flask-Security-Too==5.5.2
- Flask-RESTful==0.3.10
- Flask-Caching==2.3.0
- Flask-Excel==0.0.7
- Flask-SQLAlchemy==3.1.1
- SQLite==3.49.1
- SQLAlchemy==2.0.36
- Celery==5.4.0
- Redis==5.2.1
- pyexcel==0.7.1
- WeasyPrint==64.0
- datetime and smtplib ==Python's Inbuilt Library
- JavaScript (ES6+)
- Vue==2.7.16
- Vue-Router==3.0.0
- Vuex==3.0.0
- Flatpickr.js==4.0.0
- Chart.js==4.4.8
- Bootstrap==5.3.3

For the backend, I used **Python**, **Flask**, and its extensions, with **SQLite3** for data storage. On the front end, I used **JavaScript**, **Vue** with **Vue-Router** and **Vuex**, **Flatpickr** for date-time booking, **Chart.js** for data visualization, and **Bootstrap** for styling. For backend jobs and caching I used **Celery** and **Redis**.

DB Schema Design

ER Diagram Link

I designed the database schema to manage users and their roles, services and service requests, customers, and professionals, incorporating all essential columns specified in the project document. To enhance data access and formatting, I implemented **@property** methods in **models.py** to dynamically calculate the average ratings of professionals and services and format service request timestamps to increase their readability.

API Design

For this project, I implemented **11 API endpoints** using Flask-RESTful to handle various operations related to users, services, and service requests. These endpoints are designed for customer and professional registration, service management, service request handling, and user administration.
The API design is as follows:

1. RegisterAPI **(/api/register)**
   HTTP METHODS:
   i. **POST:** Registers new Customers or Professionals.
   Access: Auth-Token not required.; Requires to first get registered at **/register** to add user to database using datastore.

2. ServiceAPI **(/api/service/<int:service_id>)**
   HTTP METHODS:
   i. **GET:** Retrieves details of a specific service. Requires only the service_id.
   ii. **PUT:** Updates an existing service. Requires the service_id along with all service details such as name, type etc.
   iii. **DELETE:** Removes a specific service. Requires the service_id. This operation also triggers **ON DELETE CASCADE**, automatically deleting all associated service requests for the specified service.
   Access: Auth-Token required for all 3 methods. PUT and DELETE are restricted to Admin only.

3. ServiceListAPI **(/api/service)**
   HTTP METHODS:
   i. **GET:** Retrieves details of all services and supports searching by specifying the query parameter q=" ". **[Cached]**
   ii. **POST:** Creates a new service. Requires all the necessary details like name, type, price etc.
   Access: Auth-Token required for POST only.

4. ServiceRequestAPI **(/api/service_request/<int:serv_req_id>)**
   HTTP METHODS:
   i. **GET:** Retrieves details of a specific service request.
   ii. **PATCH:** Updates the specific service request. Customers can close or cancel the service request and Professionals can accept a new service request by sending appropriate data in payload.
   Access: Auth-Token required for both GET and PATCH.

5. ServiceRequestListAPI **(/api/service_request)**
   HTTP METHODS:
   i. **GET:** Retrieves details of all service requests and supports searching by specifying the query parameter q=" ". **[Cached]**
   ii. **POST:** Creates a new service request. Requires all the necessary details like service_id customer_id, etc.
   Access: Auth-Token required for both GET and POST.

6. CustomerAPI **(/api/customer/<int:c_id>)**
   HTTP METHODS:
   i. **GET:** Retrieves details of a specific customer.
   ii. **PUT:** Updates the specific customer's profile. Requires all the necessary details like name, contact, address, pin code etc.

   Access: Auth-Token required for both GET and PUT.

7. CustomerListAPI **(/api/customer)**
   HTTP METHODS:
   i. **GET:** Retrieves details of all customers and supports searching by specifying the query parameter q=" ". **[Cached]**

   Access: Auth-Token required for GET.

8. ProfesssionalAPI **(/api/professional/<int:p_id>)**
   HTTP METHODS:
   i. **GET:** Retrieves details of a specific professional.
   ii. **PUT:** Updates the specific professional's profile. Requires all the necessary details like name, contact, pin code etc.

   Access: Auth-Token required for both GET and PUT.

9. ProfessionalListAPI **(/api/professional)**
   HTTP METHODS:
   i. **GET:** Retrieves details of all customers and supports searching by specifying the query parameter q=" ". **[Cached]**

   Access: Auth-Token required for both GET.

10. UserAPI **(/api/user/<int:user_id>)**
    HTTP METHODS:
    i. **GET:** Retrieves details of a specific user.
    ii. **PATCH:** Updates the specific user's status. Admin can Block or Unblock users.
    iii. **DELETE:** Removes a specific user. When Admin rejects a Professional.

    Access: Auth-Token required for all GET, PATCH and DELETE.

11. UserListAPI **(/api/user)**
    HTTP METHODS:
    i. **GET:** Retrieves details of all users and supports searching by specifying the query parameter q=" ". **[Cached]**

    Access: Auth-Token required for GET.

<u>Architecture and Features</u>

I have structured my project into two major components: **Backend** and **Frontend**.
In the **Backend** directory, I have organized the **Flask configuration, database models, API endpoints, and Celery tasks** to efficiently handle business logic, data processing, and asynchronous operations. In the **Frontend** directory, I have placed all **JavaScript files containing Vue components, Vue Router, and Vuex for state management**. Additionally, I have included a standard index.html file that contains all necessary **CDN links** with initialization for **Vue application**.

The backend handles user authentication, service requests, and background tasks. At the same time, the frontend is responsible for rendering the user interface, handling navigation through routing, managing application state, and fetching data from APIs to update the user interface dynamically.

The **Core Functionalities** are:

1. **Authentication and Role-Based Access Control (RBAC)**:
   i. **Login** for Admin, Professionals, and Customers and **registration** for new Customers and Professionals.
   ii. Implemented **Flask-Security Token-based authentication** for Role-Based Access Control.
2. **Admin Dashboard:**
   i. **CRUD** operations for Services.
   ii. Ability to **Approve/Reject new Service Professionals** after verifying their profile.
   iii. Ability to **view all the Service Requests** made by customers.
   iv. Ability to **search the database** for Services, Service Requests, Customers and Professionals.
   v. Ability to **Block/Unblock Customers and Professionals**.
   vi. Ability to **view summarised overall service-ratings** and **service-statuses**.
3. **Customer Dashboard:**
   i. Ability to **book Services** and choose a **preferred date and time** using **Flatpickr.**
   ii. Ability to **view all the past and present Service Requests**.
   iii. Ability to **Cancel/Close Service Requests** and **rate the Professional and Service**.
   iv. Ability to **view and edit Customer profile**.
   v. Ability to **search the database** for Services to book and view past and present Service Requests.
   vi. Ability to **view summarised overall service & professional ratings** given and **service-statuses**.
4. **Professional Dashboard:**
   i. Ability to **accept Service Requests** that match the **Professional's pin code and service expertise**.
   ii. Ability to **view past and present Service Requests**.
   iii. Ability to **view and edit Professional profile**.
   iv. Ability to **search the database** for past and present Service Requests.
   v. Ability to **view summarised overall ratings** given by Customer and **service request statuses**.
5. **Backend Jobs:**
   - The backend jobs are managed by **Celery-Workers**, **Celery-Beat** and **Redis**.
     i. Admin has the ability to export a **CSV** file containing all the Service Requests closed by Professionals. The CSV file is exported using **Flask-Excel**.
     ii. The backend system sends **hourly reminders** to all available professionals to accept any service request which matches the professional's expertise and serviceable pin code. The reminders are sent in the form of emails using **smtplib** and **MailHog**.
     iii. The backend system sends **monthly reports** to all customers containing a summary of their service requests. The report is sent as a pdf using **WeasyPrint, smtplib**, and **MailHog.**
6. **Performance and Caching:**
   i. API responses are cached using **Flask-Caching** and stored in **Redis** to improve performance.

The **Additional Functionalities** are:

i. Integration of **Flatpickr** to enable Customers to select a preferred date and time when booking services.
ii. Utilization of **Chart.js** to display summarized data on the respective dashboards of **Admins, Professionals, and Customers** for better insights.
iii. Implementation of **Celery and Redis** to check every **2 hours** for unaccepted service requests that have passed their scheduled time, then the backend system **reschedules it for the next day** and sends an email notification to the respective Customer, informing them that no Professional was available to accept the request and hence it was rescheduled to next day.

Project Demo Video Link