Project Report
Modern Application Development – 2
Household Services Application Version 2

Author Details
Name – Shahbaaz Singh
LinkedIn – https://www.linkedin.com/in/shahbaazsingh/

I approached the development of *Household Services Application Version 2* systematically by breaking down the project into key milestones and tackling each functionality step by step. I started by understanding the requirements, designing the architecture, and setting up the backend and frontend structures. For each feature, I followed an iterative process—implementing, testing, and refining to ensure efficiency and reliability. Whenever challenges arose, I leveraged debugging techniques, online resources, and systematic troubleshooting to resolve them. This structured approach not only helped in building a well-functioning application but also deepened my understanding of modern application development.

Description –
In this project, we developed a multi-user *Household Services Application Version 2* using Flask for API, VueJS for User Interface, Bootstrap, SQLite for database, Redis for caching, and Redis and Celery for batch jobs. The app provides a platform for managing and delivering home services. It includes three roles:

- **Admin**: Handles service creation, user monitoring, service professional approval, and user blocking for fraudulent activities.
- **Service Professionals**: Accept/reject service requests based on their expertise and provide services.
- **Customers**: Book service requests, search for services, and leave reviews.

Core functionalities include user authentication, service creation and management, service request handling, search features, and the ability for service professionals to take actions on requests. The application also supports CRUD operations for services and service requests. The application also included Backend Jobs like Daily reminders for professionals, monthly report for customers and triggered exporting of CSV report for professionals.

Technologies Used:

1. **Flask**: Used for building the web application.
2. **Flask-SQLAlchemy**: An extension of Flask, used for database connections and ORM (Object Relational Mapping).
3. **Flask-Migrate**: Handles database migrations seamlessly.
4. **Flask-JWT-Extended:** P rovides JSON Web Token (JWT) authentication for securing Flask applications.
5. **Flask-Limiter**: Implements rate limiting in Flask applications to prevent abuse and excessive API requests.
6. **Flask-CORS**: Enables Cross-Origin Resource Sharing (CORS) to allow requests from different domains.
7. **Celery**: Used for executing background tasks and batch jobs asynchronously.
8. **Redis**: Used as a task queue backend for Celery and for caching frequently accessed data.
9. **Flask-Mail**: Enables sending emails from Flask applications using SMTP.
10. **Bootstrap**: A framework for front-end styling and creating responsive designs.

11. **datetime** (from Python’s standard library): Used for managing timestamps and date-related functionalities.
12. **os** (from Python’s standard library): Used to manage file paths, particularly for handling the documents directory.
13. **json**: A lightweight data format used for storing and exchanging data in a human-readable format.
14. **request** (from Flask): Extracts and processes data from incoming HTTP requests.
15. **wraps** (from functools): Used to create a *no-cache* decorator to prevent browser caching of specific pages.
16. **Werkzeug**: Handles routing, debugging, and application security.
17. **csv**: A simple file format used for storing tabular data, with values separated by commas.
18. **Axios**: A promise-based HTTP client for making API requests in both frontend and backend applications.
19. **APScheduler**: A Python library used for scheduling jobs and tasks, allowing you to run functions at specific intervals or times. It supports various types of job stores, schedulers, and executors, and can be integrated with Flask or other web frameworks for managing periodic or delayed tasks.

Architecture and Structure:

The *House-Services-Application-Version-2* project is structured into three main folders: **backend**, **frontend**, and **venv**, along with three files—**.gitignore**, **LICENSE (MIT)**, and **README.md**. The **venv** folder is hidden from GitHub due to its inclusion in the **.gitignore** file.
The backend folder contains:

- The app folder, which includes subdirectories such as **templates** (containing the monthly report HTML file) and other essential files: **__init__.py**, **models.py**, **routes.py**, **tasks.py**, **scheduler.py** and **utils.py**.
- The **exports** folder, designated for storing CSV exports.
- The **instance** folder for instance-specific configurations.
- The **migrations** folder for handling database migrations.
- The **uploads** folder for storing professional document uploads.
- Other key files like **celery_worker.py**, **config.py**, and **run.py**, which help in application execution and task management.

The frontend folder includes:

- **dist**, **node_modules**, **public**, and **src** directories, along with other necessary project files.
- The **src** folder contains:
    - The assets folder for storing static files like **images** and **styles**.
    - The **components** folder, which holds all Vue components required for frontend functionality.
    - **api.js** – Manages API requests and communication between the frontend and backend.
    - **App.vue** – The root component of the Vue application, defining the main layout and structure.
    - **main.js** – The entry point for initializing the Vue app, setting up plugins, and mounting the application.
    - **routes.js** – Defines the application's routing logic, mapping different paths to Vue components.

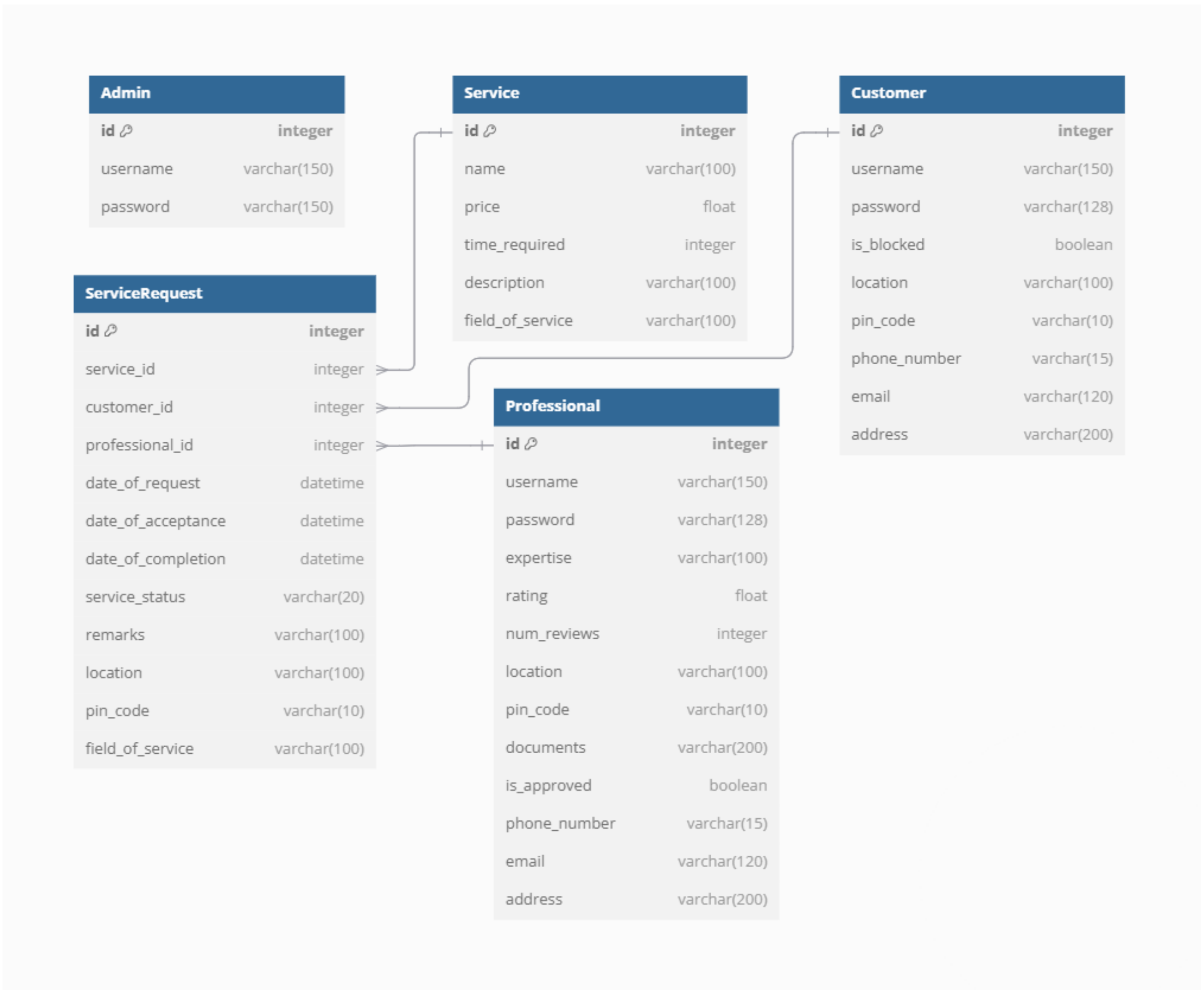
This structured organization ensures efficient development, scalability, and maintainability for the application.

DB Schema Design:

The database schema is designed with the following relationships and structures:

- Admin: The admin has root privileges to manage users and services. Admin has unique attributes like *username* and *password* with *id* as the primary key.
- Customer: A customer can create multiple service requests, establishing a one-to-many relationship between the Customer and ServiceRequest tables. The *id* is the primary key, and the table includes attributes like *username*, *email*, and *location*, etc.
- Professional: A professional can handle multiple service requests, creating a one-to-many relationship between the Professional and ServiceRequest tables. The table tracks details like *expertise*, *rating*, and *documents*, etc. Professional also have a function 'has_pending_requests()' to check whether the professional have pending requests if yes then how many pending requests, this function is needed for daily reminders.
- Service: Each service can have multiple service requests associated with it, forming a one-to-many relationship between the Service and ServiceRequest tables. It includes attributes like *name*, *price*, and *description*, etc.
- ServiceRequest: This table acts as a bridge linking services, customers, and professionals. It includes foreign keys referencing the service, customer, and professional tables, along with attributes like *date_of_request*, *status*, and *remarks*, etc

For more details on the database schema, relationships, and other components, please refer to the image below:



Admin	
id	integer
username	varchar(150)
password	varchar(150)

Service	
id	integer
name	varchar(100)
price	float
time_required	integer
description	varchar(100)
field_of_service	varchar(100)

Customer	
id	integer
username	varchar(150)
password	varchar(128)
is_blocked	boolean
location	varchar(100)
pin_code	varchar(10)
phone_number	varchar(15)
email	varchar(120)
address	varchar(200)

ServiceRequest	
id	integer
service_id	integer
customer_id	integer
professional_id	integer
date_of_request	datetime
date_of_acceptance	datetime
date_of_completion	datetime
service_status	varchar(20)
remarks	varchar(100)
location	varchar(100)
pin_code	varchar(10)
field_of_service	varchar(100)

Professional	
id	integer
username	varchar(150)
password	varchar(128)
expertise	varchar(100)
rating	float
num_reviews	integer
location	varchar(100)
pin_code	varchar(10)
documents	varchar(200)
is_approved	boolean
phone_number	varchar(15)
email	varchar(120)
address	varchar(200)

API Design

- **User_api (Registration)**: Implemented using GET and POST methods. The GET method retrieves user details, including their associated services and service statuses from the database. The POST method is used to create new users, storing their information such as name, email, and role (customer or professional) in the system.
- **home_api**: Implemented using GET method this endpoint allow users to get to the home page of the application.
- **Admin_api**: Implemented using GET, POST, PUT and DELETE methods. The GET method is used by admins to fetch customer and professional details. The POST method allows admins to add new services, by specifying price, description, time required, field of service, name. The PUT method is used to update professional and customer status of being approved, blocked, etc., updating services, while the DELETE method enables admins to delete services.
- **Customer_api**: Implemented using GET, POST, and PUT methods. The GET method retrieves a customer's profile, service requests, and any ongoing services they have requested. The POST method allows customers to submit new service requests, specifying details like service name, pincode, and location. The PUT method is used by customers to update the status of a service request or modify any details of the service, such as rescheduling or providing additional information.
- **Professional_api**: Implemented using GET and POST methods. The GET method retrieves a list of professionals, and their specific service expertise. The POST method allows professionals to accept and reject services. The PUT method is used by professionals to update the status of a service request (e.g., "Accepted," "Completed," etc.) or to mark a service requests as completed.
- **token_api**: Implemented using POST method this endpoint allow users to obtain a new access token using a valid refresh token. It verifies the refresh token, retrieves the user's identity, and generates a new access token if valid.
- **Document_api**: Implemented using GET method this endpoint allow admin to obtain the document submitted by a particular professional.

Wireframe:

The controllers that were used:

```
@main.route('/api/home', methods=['GET'])
@main.route('/api/register', methods=['POST'])
@main.route('/api/documents/<path:filename>', methods=['GET'])
@main.route('/api/refresh-token', methods=['POST'])
@main.route('/api/admin/login', methods=['GET', 'POST'])
@main.route('/api/admin/dashboard', methods=['GET'])
@main.route('/api/admin/export_closed_requests', methods=['POST'])
@main.route('/api/admin/customer_info', methods=['GET'])
@main.route('/api/admin/block_user/<int:id>', methods=['POST'])
@main.route('/api/admin/unblock_user/<int:id>', methods=['POST'])
@main.route('/api/admin/professional_info', methods=['GET'])
@main.route('/api/admin/approve/<int:id>', methods=['POST'])
@main.route('/api/admin/unapprove/<int:id>', methods=['POST'])
@main.route('/api/admin/services', methods=['GET'])
@main.route('/api/admin/create_service', methods=['POST'])
@main.route('/api/admin/update_service/<int:id>', methods=['POST'])
@main.route('/api/admin/delete_service/<int:id>', methods=['POST'])
@main.route('/api/admin/logout', methods=['POST'])
```

```
@main.route('/api/customer/login', methods=['POST'])
@main.route('/api/customer/dashboard', methods=['GET'])
@main.route('/api/customer/create_request', methods=['GET'])
@main.route('/api/customer/search_services', methods=['GET'])
@main.route('/api/customer/request_service', methods=['POST'])
@main.route('/api/customer/service_requests', methods=['GET'])
@main.route('/api/customer/update_request/<int:id>', methods=['PUT'])
@main.route('/api/customer/close_request/<int:id>', methods=['POST'])
@main.route('/api/customer/logout', methods=['POST'])
@main.route('/api/professional/login', methods=['POST'])
@main.route('/api/professional/dashboard', methods=['GET'])
@main.route('/api/professional/export_pending_requests', methods=['POST'])
@main.route('/api/professional/download_export', methods=['GET'])
@main.route('/api/professional/pending_requests', methods=['GET'])
@main.route('/api/professional/accept_service/<int:id>', methods=['POST'])
@main.route('/api/professional/reject_service/<int:id>', methods=['POST'])
@main.route('/api/professional/accepted_requests', methods=['GET'])
@main.route('/api/professional/update_service_status/<int:id>', methods=['POST'])
@main.route('/api/professional/logout', methods=['POST'])
```

Video:
Google Drive Links:

Demo Video – https://drive.google.com/file/d/1-OoRDdkN53LRvHTYiB17wukgy6RTc77t/view?usp=sharing