Project Report

Mad2 Household Project-Servigo Application

Author

Name: Vikas Rathore
Roll_no: 22f1001805
Email: 22f1001805@ds.study.iitm.ac.in

About Me

My name is Vikas, and I'm really interested in both design and machine learning. Whenever I see any design, I naturally start thinking about how it can be made better or easier to use. This habit helps me come up with creative ideas.

I'm also very curious about how fast AI is growing and how it's being used in real life. It keeps me excited to learn more and explore new things in this field.

Description

In this project, we need to build a platform where customers can book professionals like painters, cooks, or caretakers for household services. Service professionals get job opportunities, and customers can easily get their work done. The admin manages everything from verifying professionals to handling users and services.

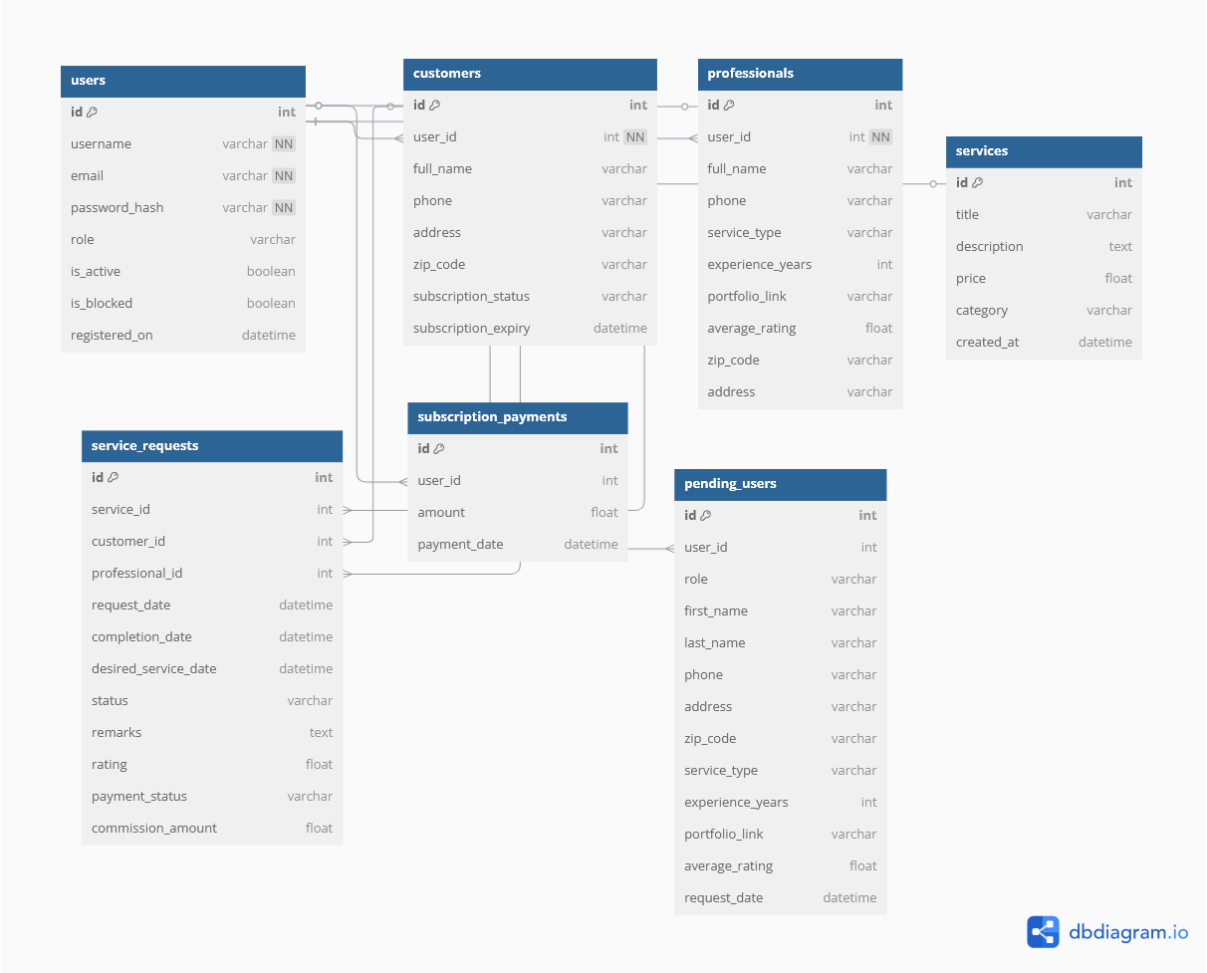
Technologies used

In this project, I've used a mix of backend, frontend, and utility tools to build a complete household services platform. Here's a quick breakdown of what I used and why:

- **Flask** – Used as the main backend framework to build APIs and connect everything smoothly.
- **Flask SQLAlchemy** – Helped me interact with the database using Python instead of writing raw SQL queries.
- **Flask-JWT-Extended** – Used for login and token-based authentication for admin, customer, and professional roles.
- **Flask-Mail** – To send emails like monthly reports to customers and reminders to professionals.
- **Flask-CORS** – Allowed the frontend (Vue.js) to communicate with the Flask backend without CORS issues.
- **Flask-Caching** – Added to improve performance by storing frequently accessed data temporarily.
- **Flask-Bcrypt** – Used to securely store and check hashed passwords during login.
- **Blueprints** – Helped in organizing my backend code cleanly by separating routes by modules.

- **Werkzeug** – Used internally by Flask, but helped with request parsing, security, and response handling.

DB Schema Design



I designed the database based on the three main roles: admin, customer, and professional. The users table holds common info like email, username, and role. Then, I created separate tables for customers and professionals since they have different details (like experience, subscription, etc.).

I used a pending_users table to keep signups separate until approved by the admin. services holds all the available services, and service_requests tracks bookings, status, payments, and reviews between customers and professionals.

Foreign keys link the tables properly, and I used enums to keep status and roles consistent. This setup keeps things clean and easy to manage.

API Design

I created APIs for all the core features like user signup/login, role-based dashboards, service creation and management, service requests (booking, accepting, rejecting, etc.), subscription handling, and admin controls.

All APIs follow REST standards using Flask with proper HTTP methods (GET, POST, PUT, DELETE). I've used JWT for authentication and role checking, and CORS is enabled for frontend access.

The APIs are modular and organized using Flask Blueprints. A separate YAML file lists the API structure.

Architecture and Features

I've structured the project into two clean sections: backend and frontend.

On the backend side, I wrote all the core API logic in `routes.py`, and set up the database models in `models.py`. The app kicks off from `main.py`, where I've configured things like JWT authentication, email setup, Redis caching, and registered all blueprints. I didn't use any templating engines like Jinja—this project is purely API-driven. For background jobs like sending monthly reports or professional reminders, I used Celery, and the job schedules are defined in a simple YAML file.

On the frontend, everything lives inside the `src` folder, built using Vue.js. That's where I created all the components, pages, and routes. Static files like `index.html` sit in the `public` folder.

I've implemented the main features such as login/signup for all three roles (admin, customer, and professional), dashboard views, and full service request flows. Admins can approve users and manage services.

To go a step further, I added a payment system so customers can pay once the job is marked complete. There's also a subscription option where customers can pay $20 to unlock premium features. Revenue is automatically tracked through both subscriptions and service commissions, and everything runs smoothly thanks to the APIs and background tasks handled by Celery and Redis.

Video

https://drive.google.com/drive/folders/1f6ExV8O8dDsAp0VAnn0F0oooWsNBPJad?usp=sharing