



# Quack

## Author

Dawar Farooq

21f2000557

21f2000557@ds.study.iitm.ac.in

## Description

Quack is a service marketplace platform that connects customers with professional service providers. The system manages the complete lifecycle of service requests with role-based access for customers, professionals, and administrators.

## Technologies Used

### Backend

- **Flask:** Core Python web framework providing routing, request handling, and application structure
- **Flask-SQLAlchemy:** ORM for database interactions, enabling Python object mapping to database tables
- **Flask-Migrate:** Database migration tool for schema version control and updates
- **Flask-RESTX:** Extension for building RESTful APIs with automatic Swagger documentation
- **Flask-JWT-Extended:** Authentication via JSON Web Tokens for stateless authorization
- **Flask-CORS:** Cross-Origin Resource Sharing support for frontend-backend communication
- **Flask-Mail:** Email functionality for notifications and verification

- **Celery:** Asynchronous task queue for background operations like email sending and document processing
- **Redis:** In-memory data store used as message broker for Celery tasks
- **PostgreSQL:** Robust relational database with advanced features for data integrity
- **Gunicorn:** WSGI HTTP Server for production deployment with worker management

## Frontend

- **Vue.js 3:** Progressive JavaScript framework using Composition API for building reactive user interfaces
- **Pinia:** State management for Vue with TypeScript support and devtools integration
- **Vue Router:** Client-side routing with navigation guards for authentication
- **Axios:** Promise-based HTTP client for API calls with request/response interception
- **Bootstrap 5:** CSS framework for responsive design with custom theming
- **Chart.js/Vue-Chartjs:** Data visualization components for analytics dashboards
- **Vite:** Modern frontend build tool with hot module replacement and optimized builds

## Deployment

- **Docker:** Application containerization for consistent environments
- **Docker Compose:** Multi-container orchestration for backend, frontend, database, and Redis
- **Nginx:** Web server for static file serving and API proxying

## DB Schema Design

## erDiagram

```
User {
    int id PK
    string username UK "Unique username for login"
    string password "Hashed password"
    datetime date_created "Auto timestamp"
    text description "Professional description"
    string experience "Years/type of experience"
    string service_type "Category of service offered"
    boolean profile_docs_verified "Document verification status"
    boolean blocked "Account access control"
    string status "pending/approved/disapproved"
    text rejection_reason "Admin feedback"
    string name "Full name"
    string email UK "Contact email"
    string phone_number "Contact number"
    string profile_image "Avatar path"
    string address "Physical location"
}
```

```
Role {
    int id PK
    string name UK "admin/customer/professional"
}
```

```
UserRoles {
    int user_id PK,FK "Composite primary key with role_id"
    int role_id PK,FK "Composite primary key with user_id"
}
```

```
Service {
    int id PK
    string name "Service title"
    float price "Cost in currency"
    string time_required "Estimated duration"
```

```
    text description "Detailed service information"
}
```

```
ServiceRequest {
    int id PK
    int service_id FK "Reference to service"
    int customer_id FK "User requesting service"
    int professional_id FK "User providing service"
    datetime date_of_request "Creation timestamp"
    datetime date_of_completion "Fulfillment timestamp"
    string service_status "pending/accepted/completed/cancelled"
    text remarks "Request details"
    string location_pin_code "Service location"
    date preferred_date "Scheduled date"
}
```

```
Document {
    int id PK
    int user_id FK "Owner reference"
    string document_type "id_proof/certification/etc"
    string file_name "Original filename"
    string file_path "Storage location"
    datetime upload_date "Submission timestamp"
    boolean verified "Approval status"
    boolean rejected "Rejection status"
    text rejection_reason "Admin feedback"
}
```

```
User ||--o{ UserRoles : has
Role ||--o{ UserRoles : belongs_to
User ||--o{ ServiceRequest : "requests as customer"
User ||--o{ ServiceRequest : "fulfills as professional"
Service ||--o{ ServiceRequest : requested
User ||--o{ Document : uploads
```

## Database Design Implementation Details

- **Many-to-Many Relationships:** UserRoles junction table implements many-to-many relationship between users and roles
- **Self-Referential Relations:** ServiceRequest references User twice (as customer and professional)
- **Constraints:**
  - Unique constraints on username and email in User table
  - Foreign key constraints enforce referential integrity
  - Non-nullable fields prevent data inconsistency
- **Status Tracking:** Enumerated status fields for workflows (service requests, document verification)
- **Security Considerations:** Password stored as hashed values, not plaintext
- **Timestamps:** Automatic creation timestamps for auditing

## API Design

The API implements a RESTful architecture with JWT-secured endpoints organized by resource and role permissions:

### Authentication Endpoints

- `POST /auth/register` : New user registration with role selection
- `POST /auth/login` : User authentication returning JWT tokens
- `POST /auth/refresh` : Token refresh for maintaining sessions
- `POST /auth/reset-password` : Password recovery workflow

### Customer Endpoints

- `GET /customer/profile` : Retrieve customer profile data
- `PUT /customer/profile` : Update customer information
- `GET /customer/services` : List available services with filtering
- `POST /customer/service-requests` : Create new service request

- `GET /customer/service-requests` : List customer's service history
- `PUT /customer/service-requests/<id>` : Update request details
- `POST /customer/reviews` : Submit professional reviews

## Professional Endpoints

- `GET /professional/profile` : Retrieve professional profile
- `PUT /professional/profile` : Update professional information
- `POST /professional/documents` : Upload verification documents
- `GET /professional/service-requests` : View assigned requests
- `PUT /professional/service-requests/<id>/status` : Update request status
- `GET /professional/earnings` : View earnings analytics

## Admin Endpoints

- `GET /admin/users` : List all users with filtering
- `PUT /admin/users/<id>/status` : Approve/block users
- `GET /admin/documents` : Review pending documents
- `PUT /admin/documents/<id>/verify` : Verify professional documents
- `POST /admin/services` : Create new service offerings
- `GET /admin/analytics` : System-wide statistics and reports

## API Security Implementation

- **JWT Authentication:** Tokens for secure stateless authentication
- **Role-Based Access Control:** Route protection based on user roles
- **Request Validation:** Input validation using Flask-RESTX models
- **Rate Limiting:** Protection against abuse
- **Error Handling:** Consistent error responses with appropriate HTTP status codes

# Architecture and Features

## Project Architecture

```
graph TD
    Client[Vue.js Frontend] → Nginx
    Nginx → API[Flask API]
    API → DB[(PostgreSQL)]
    API → Redis[(Redis)]
    API → Celery[Celery Workers]
    Celery → EmailService[Email Service]
    Celery → FileProcessor[File Processor]
```

## Backend Structure

- **Factory Pattern:** Application factory for configuration and initialization
- **Blueprints:** Modular organization of routes by domain
  - `/app/routes/auth.py` : Authentication endpoints
  - `/app/routes/customer.py` : Customer-specific endpoints
  - `/app/routes/professional.py` : Professional-specific endpoints
  - `/app/routes/admin.py` : Administrative operations
  - `/app/routes/service.py` : Service management
- **Service Layer:** Business logic separation
- **Background Tasks:** Asynchronous processing with Celery
  - Email notifications
  - Document verification processing
  - Scheduled tasks

## Frontend Architecture

- **Component-Based:** Reusable UI components
- **Stores Pattern:** Pinia stores for global state management

- AuthStore: User authentication state
- UserStore: User profile and preferences
- ServiceStore: Service catalog
- RequestStore: Service request management
- **Route Guards:** Navigation protection based on authentication status
- **API Services:** Axios-based service modules for API communication
- **Responsive Design:** Mobile-first approach with Bootstrap

## Key Features Implementation

### Authentication System

- JWT-based authentication with token refresh
- Role-based authorization with route guards
- Password hashing with secure algorithms
- Account recovery workflows

### Service Management

- Catalog browsing with search and filtering
- Request creation with scheduling
- Status tracking across the service lifecycle
- Professional selection and assignment

### Professional Verification

- Document upload and storage
- Admin review interface
- Multi-step verification process
- Email notifications on status changes

### Analytics and Reporting



- Dashboard with key metrics
- Service performance analytics
- User activity tracking
- Financial reporting for professionals

## Video

[tinyurl.com/mad2project](https://tinyurl.com/mad2project)