Autonomeet Project Documentation
SW2

Carlos Honrado Cristobalena Daniel Torres Montoya

April 2025

Contents

1 Introduction

This document outlines the project structure, authentication process, registration process, and database schema for **Autonomeet**, developed as part of the SW2 course.

2 Project Architecture

The architecture of Autonomeet is divided into frontend, backend, database, and payment gateway components. Below is a diagram illustrating the system structure, derived from the provided Mermaid diagram.

2.1 Component Description

- **Frontend**:
  - *Client*: Provides a view and interface for clients to interact with the system.
  - *Freelancer*: Offers a view and interface for freelancers to manage their services and appointments.
- **Backend**:
  - *API Backend*: Handles requests from the frontend, manages data, and processes payments.
  - *Database*: Stores user information, business details, and appointment records.
- **Database**: Contains tables for users, business information, and appointments.
- **Payment Gateway**: Processes payments through an external service.

3 Authentication Process

The authentication process in Autonomeet allows users to log in using email/password credentials, Google, or GitHub. The frontend manages user state with a React context, while the backend handles authentication logic and issues JSON Web Tokens (JWTs) for secure access.

3.1 Overview

Authentication involves:

- **Frontend**: A React context manages user state, storing JWTs in local storage and decoding them to extract user data (`id`, `email`).
- **Backend**: A Flask-based service verifies credentials, integrates with Google and GitHub OAuth, and generates JWTs using a utility function.
- **API Endpoints**: RESTful endpoints (`/auth/login`, `/auth/google`, `/auth/github`) handle authentication requests and return JWTs upon success.

3.2 Authentication Flow

The login process is illustrated below for email/password authentication. Google and GitHub flows follow a similar pattern, with OAuth token validation replacing password checks.

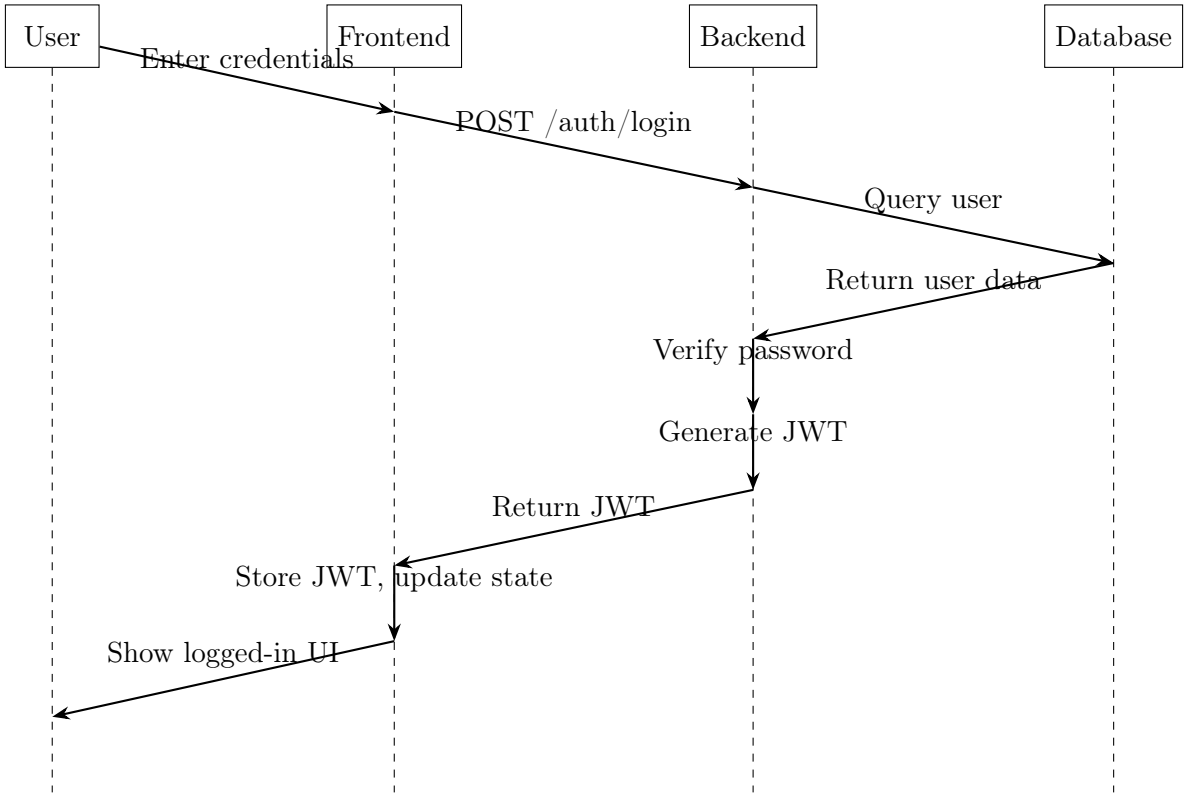

Figure 1: Authentication Sequence Diagram (Email/Password)

3.3 Frontend Authentication

The frontend uses a React context to manage authentication state. The provider component initializes the user state from a stored JWT and provides functions for login and logout. The authentication service handles API calls for email/password login, Google OAuth, and GitHub OAuth, sending requests to the backend and processing responses to update the user state.

3.4 Backend Authentication

The backend authentication service validates credentials and integrates with OAuth providers. For email/password login, it checks the user's email and hashed password against the database. For Google OAuth, it verifies the provided token using Google's OAuth library, creating or retrieving a user based on the email. For GitHub OAuth, it exchanges an authorization code for an access token, fetches user data, and creates or updates a user based on the GitHub ID and email.

3.5 API Endpoints

The backend exposes RESTful endpoints for authentication. The login endpoint accepts email and password, returning a JWT upon successful verification. The Google endpoint processes OAuth tokens, and the GitHub endpoint handles authorization codes, both returning JWTs for valid authentications.

3.6 Authentication Mechanisms

- **Email/Password**: Users submit credentials, which the backend verifies against hashed passwords in the `users` table. A JWT is issued upon success.
- **Google OAuth**: The frontend sends a Google token, which the backend verifies using Google's OAuth library. If valid, a user is created or retrieved, and a JWT is issued.
- **GitHub OAuth**: The frontend sends an authorization code, which the backend exchanges for an access token to fetch user data. A user is created or updated, and a JWT is issued.

4 Registration Process

The registration process in Autonomeet enables new users to create accounts by providing an email, password, and user type (client or freelancer). The frontend collects and validates user input, while the backend ensures unique accounts and securely stores credentials.

4.1 Overview

Registration involves:

- **Frontend**: A React component presents a form for users to enter their email, password, confirm password, and select whether they are a freelancer or client. It validates input locally before sending it to the backend.
- **Backend**: A Flask-based service checks for existing users, hashes the password, and creates a new user record in the database, issuing a JWT upon success.
- **API Endpoint**: A RESTful endpoint (`/auth/register`) processes registration requests, returning a JWT and user details if successful.

4.2 Registration Flow

The registration process is illustrated below, showing the interaction between the user, frontend, backend, and database.

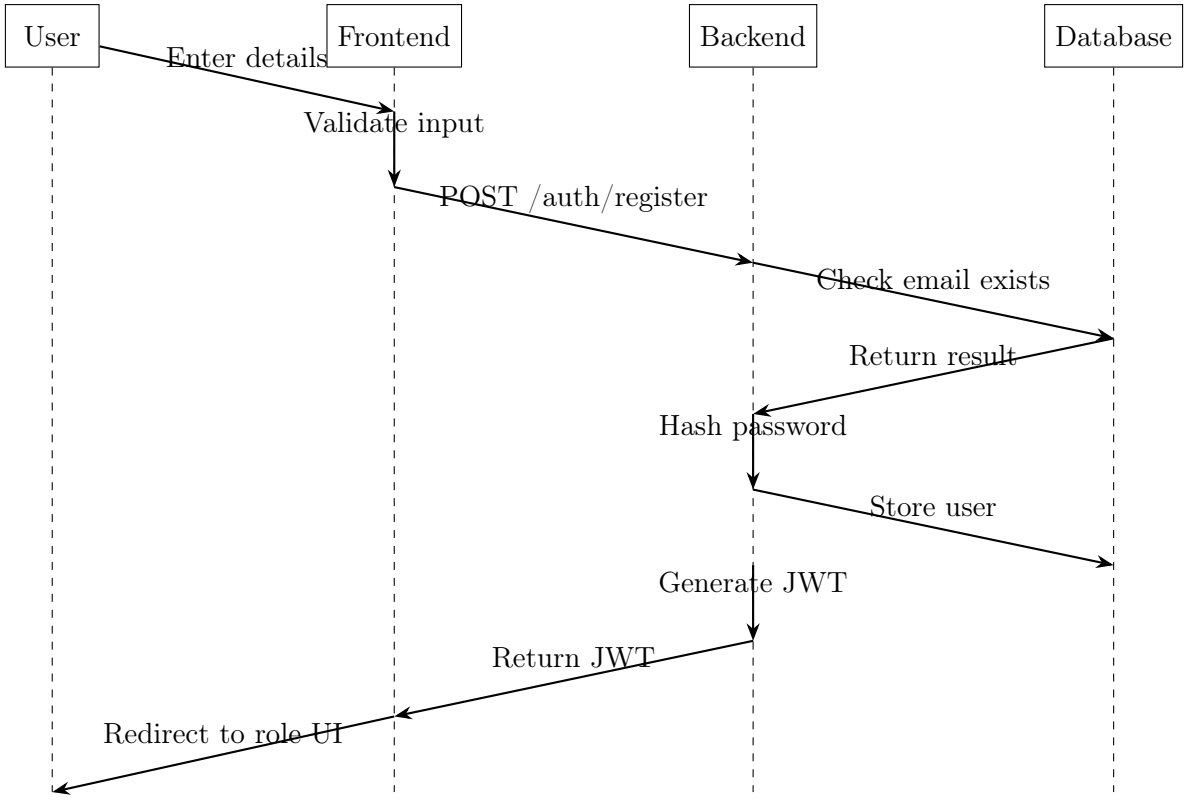

Figure 2: Registration Sequence Diagram

4.3 Frontend Registration

The frontend registration component collects user input through a form, including email, password, password confirmation, and a choice between client or freelancer roles. It validates that passwords match and meet a minimum length requirement before sending the data to the backend. Upon receiving a successful response, it logs the user in by storing the JWT and redirects them to a role-specific interface (freelancer or client dashboard).

4.4 Backend Registration

The backend registration service processes the submitted email, password, and freelancer status. It checks the database to ensure the email is not already registered. If unique, it hashes the password for security, creates a new user record with the specified role, and stores it in the database. A JWT is generated and returned to the frontend for immediate authentication.

4.5 API Endpoint

The backend provides a RESTful endpoint for registration, accepting email, password, and freelancer status. It returns a JWT, user ID, email, and role information upon successful user creation, or an error message if the email is already in use.

4.6 Registration Mechanism

The registration process ensures secure and user-friendly account creation:

- **Input Validation**: The frontend checks for matching passwords and minimum length, reducing invalid submissions.
- **Email Uniqueness**: The backend verifies that the email is not already registered, preventing duplicate accounts.
- **Password Security**: Passwords are hashed before storage in the `users` table, ensuring data protection.
- **Role Selection**: Users choose between client or freelancer roles, which are stored in the database and used to tailor their experience post-registration.
- **Immediate Access**: A JWT is issued upon registration, allowing users to access the system without a separate login step.

5 Database Schema

The database for Autonomeet is implemented in PostgreSQL. Below is a description of the schema, including table definitions and relationships.

5.1 Schema Overview

The database consists of the following tables:

- `users`: Stores user information, including email, hashed password, and freelancer status.
- `freelancer_profiles`: Contains freelancer-specific details, such as headline, description, and availability.
- `categories`: Defines service categories.
- `services`: Lists services offered by freelancers, including title, price, and category.
- `appointments`: Records appointments between clients and freelancers, with details like scheduled time and service.
- `reviews`: Stores client reviews of freelancers, including rating and comments.

5.2 Schema Description

The `users` table holds core user data, with unique email constraints and optional fields for OAuth users (e.g., GitHub ID). The `freelancer_profiles` table links to `users` and stores professional details. The `categories` table organizes services, which are detailed in the `services` table with pricing and freelancer associations. The `appointments` table tracks bookings, referencing clients, freelancers, and services. The `reviews` table captures feedback, linking reviewers and freelancers.

5.3 Table Relationships

- `users` is linked to `freelancer_profiles` via a user ID.
- `freelancer_profiles` is referenced by `services` and `appointments` through a freelancer ID.
- `categories` is linked to `services` via a category ID.
- `services` is referenced by `appointments` through a service ID.
- `reviews` connects to `users` (via an author ID) and `freelancer_profiles` (via a freelancer ID).