Gated Community Application: Technical Documentation

Group #1

Kelsey A., Jordani A., Arthur B., Aiyesha C., Aiden P.

University of Belize

CMPS4131- Software Engineering

Mr. Medina Manuel

April 11 2025

Table of Content

Introduction

This technical document is intended for developers and system administrators working with the Gated Community Application. It provides a structured overview of the system, including what the application does, the environment it operates in, and the technical details needed to install, configure, and maintain it. The Gated Community Application is designed to manage and monitor visitor access within a residential community, offering features such as visitor scheduling, QR code scanning, blacklist management, and feedback collection.

The document is organized to support both development and administrative tasks. It begins with a description of the system's main functionalities and outlines the hardware, software, and network requirements necessary for deployment. It then provides step-by-step instructions for installing and setting up the system in a production or development environment. Following this, developers can refer to detailed sections on the application's architecture, source code organization, APIs, and database structure. Finally, the document includes testing information, covering test strategies, use case validations, and known issues to ensure the reliability and stability of the system.

System Overview

The Gated Community Application is a web-based system designed to streamline the management of visitor access within a residential community. It provides a secure and organized way for residents to schedule visits, for security personnel to verify visitor access, and for administrators to monitor and manage system activity. The application aims to enhance community safety, simplify gate entry processes, and maintain accurate visitor records.

Key Features

- **Visitor Scheduling**: Residents can create and manage visit requests for guests, including details such as visit date, time, and purpose. This could be done for a single visitor or a group of visitors.
- **QR Code Generation and Scanning:** Approved visit requests automatically generate QR codes that can be scanned by security personnel at the gate for quick verification and access control.
- **Blacklist Management:** Administrators can add or remove individuals from a blacklist to prevent unauthorized or problematic visitors from entering the premises.
- **User Role Management:** The system supports multiple user roles (Admin, Resident, and Security) with access permissions tailored to each role's responsibilities.
- **Visit History and Logs:** The system maintains logs of all visits and logins , allowing users to review visit history and track entry and exit records.

- **Feedback Submission:** Residents can submit feedback about their experience with the application or visitor interactions, which can be reviewed by administrators.

High-Level Architecture

The Gated Community Application is built using Next.js, utilizing both its frontend and backend capabilities to deliver a full-stack, server-rendered experience. The application uses built-in API routes to manage all server-side logic, enabling seamless communication between the interface and backend processes.

At the backend, the system uses Prisma, a modern Object-Relational Mapping (ORM) tool, to interact with a PostgreSQL database. Prisma simplifies complex queries, ensures type safety, and enhances security by reducing the risk of SQL injection. Combined with Next.js API routes, it allows the system to maintain a modular and maintainable structure, ideal for scalable community management.

The backend is organized into the following core API modules:

- **Users** – Manages user accounts, profile information, and role assignments. It supports multiple user roles including Admin, Resident, and Security personnel.
- **Visitors** – Handles visitor scheduling, visit tracking, and QR code generation for secure access at community entry points.
- **Blacklist** – Allows authorized users to add or remove individuals from a restricted list, helping prevent unauthorized access to the community.
- **Feedback** – Enables residents to submit feedback related to visits, security experiences, or app functionality, which administrators can review.
- **Auth** – Handles authentication and session management. This module is responsible for login, logout, and maintaining secure user sessions. It directly interacts with the users model to verify credentials and enforce role-based access control.

The application architecture ensures that frontend components communicate securely with these backend APIs. All sensitive operations, such as authentication or database mutations, are handled server-side to maintain data integrity and protect user information. This setup supports a clean separation of concerns, with clearly defined data models and corresponding API routes, making the application robust and developer-friendly.

System Requirements

Our system relies on several critical resources to ensure smooth operation and optimal performance. The server must have at least 32GB of memory and a RAID 5 configuration with three 500GB hard drives to support data redundancy, security, and efficient storage management. A stable and high-speed network connection is essential for seamless communication between users and the database. Security personnel will utilize 10-inch tablets for scanning visitor QR codes, which require responsive touchscreens and reliable internet connectivity for real-time verification. Additionally, the system depends on modern web browsers for residents and administrators to access the platform, with a recommended minimum screen resolution of 1280 × 720 pixels for an optimal user experience.

Installation Guide

Installation

Before beginning, ensure you have the following installed:

- Git
- Next.js and npm
- PostgreSQL
- TSX (TypeScript execution)
- Prisma CLI

Developers must also have a github account and be a part of the repository.

Step 1: Clone the Repository

```
git clone https://github.com/jordani-alpuche/software-engineering_final.git
cd software-engineering_final
```

Step 2: Install Dependencies

Use the legacy peer dependencies flag to avoid conflicts:

```
npm install --legacy-peer-deps
```

Step 3: Set Up PostgreSQL Database

1. Create a new database.

```
sudo -u postgres psql

-- In psql shell:
CREATE DATABASE [database name];
```

2. Create a user and assign a password.

```
-- In psql shell:
\c database name
CREATE USER username WITH PASSWORD 'password';
```

Replace username, password, database with your actual PostgreSQL credentials.

3. Grant the user access to the software database:

```
-- In psql shell:
ALTER DATABASE software OWNER TO software;
GRANT CREATE ON DATABASE software TO software;
```

Step 4: Create the .env File

Create a .env file in the root directory.

```
touch .env
```

Add the following:

```
NEXTAUTH_SECRET="SuperSecretPasswordSIB"
DATABASE_URL="postgresql://[username]:[password]@localhost:5432/[database name]?schema=public&connection_limit=1"
NEXTAUTH_URL="http://localhost:3000"
```

Replace username, password, database with your actual PostgreSQL credentials.

Step 5: Run Database Migration

Generate and push the initial Prisma schema to your PostgreSQL database:

```
npx prisma migrate dev --name init
```

```
npm i tsx --legacy-peer-deps
npx prisma db push
```

Step 6: Create an Admin User

1. Navigate to the user utility script:

```
cd src/app/utils
```

2. Open createUser.ts and update the following:

```
...
7 const hashedPassword = await hashPassword("your-password");
...
12 username: "your-username",
```

These credentials are for administrative login into the frontend of the application; therefore, it does not have to coordinate with your postgres database credentials.

3. Then run the script:

```
npx tsx createUser.ts
```

This will create your admin user account with the credentials you just defined.

Step 7: Start the Development Server

```
npm run dev
```

Visit http://localhost:3000 in your browser to access the application.

Done!
Your development environment is now up and running. You can log in using the admin credentials you set in createUser.ts.

Configuration

Step 1: Still navigate to the project folder

```
cd software-engineering_final
```

Step 2: Check remote connection (just to see if it's linked)

```
git remote -v
```

Step 3: If not linked yet, add remote

```
git remote add origin
https://github.com/jordani-alpuche/software-engineering_final.git
```

Step 4: Add, commit, and push

```
git add .
git commit -m "comment"
git push -u origin main
```

Edit comment to a relevant description of code.

Developer Documentation

Database Schema

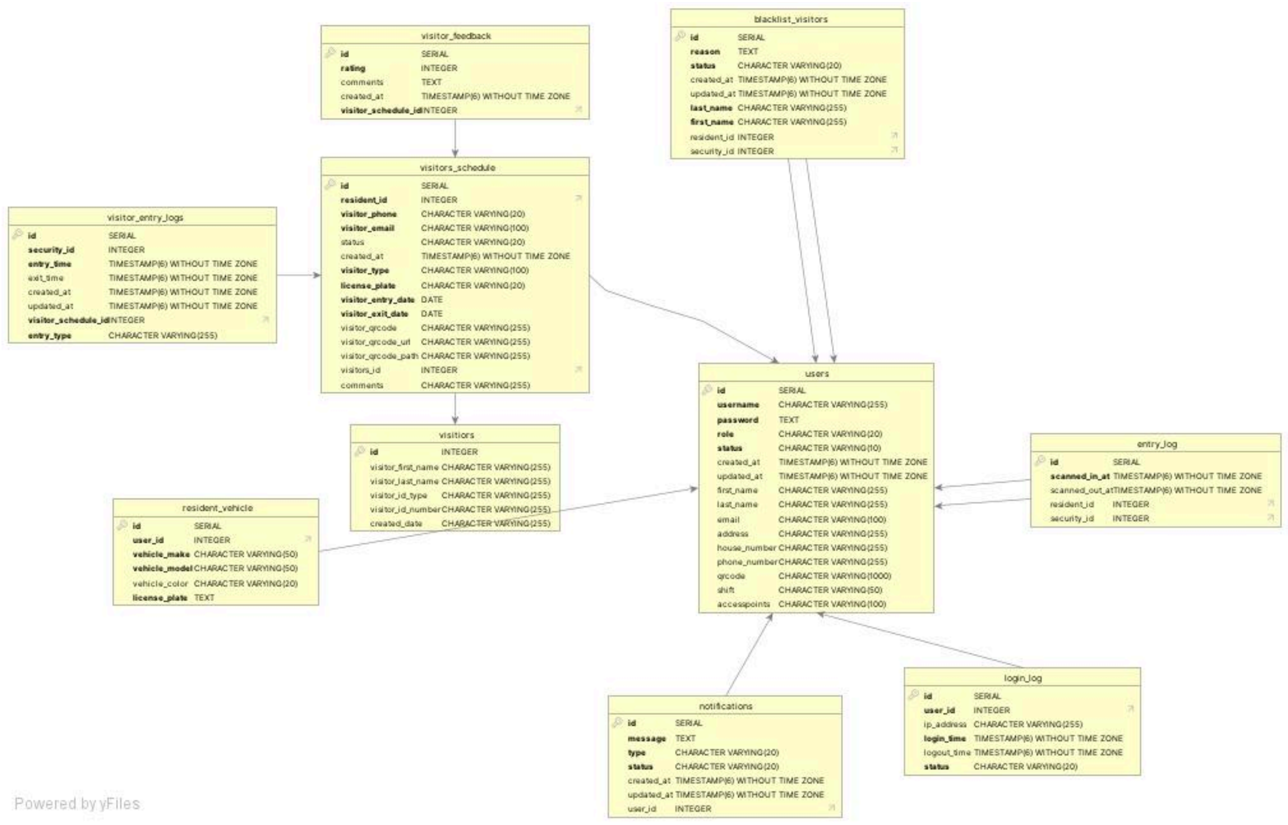
visitor_feedback
id SERIAL
rating INTEGER
comments TEXT
created_at TIMESTAMP(6) WITHOUT TIME ZONE
visitor_schedule_id INTEGER
blacklist_visitors
id SERIAL
reason TEXT
status CHARACTER VARYING(20)
created_at TIMESTAMP(6) WITHOUT TIME ZONE
updated_at TIMESTAMP(6) WITHOUT TIME ZONE
last_name CHARACTER VARYING(255)
first_name CHARACTER VARYING(255)
resident_id INTEGER
security_id INTEGER
visitors_schedule
id SERIAL
resident_id INTEGER
visitor_phone CHARACTER VARYING(20)
visitor_email CHARACTER VARYING(100)
status CHARACTER VARYING(20)
created_at TIMESTAMP(6) WITHOUT TIME ZONE
visitor_type CHARACTER VARYING(100)
license_plate CHARACTER VARYING(20)
visitor_entry_date DATE
visitor_exit_date DATE
visitor_qrcode CHARACTER VARYING(255)
visitor_qrcode_url CHARACTER VARYING(255)
visitor_qrcode_path CHARACTER VARYING(255)
visitors_id INTEGER
comments CHARACTER VARYING(255)
visitor_entry_logs
id SERIAL
security_id INTEGER
entry_time TIMESTAMP(6) WITHOUT TIME ZONE
exit_time TIMESTAMP(6) WITHOUT TIME ZONE
created_at TIMESTAMP(6) WITHOUT TIME ZONE
updated_at TIMESTAMP(6) WITHOUT TIME ZONE
visitor_schedule_id INTEGER
entry_type CHARACTER VARYING(255)
visitiors
id INTEGER
visitor_first_name CHARACTER VARYING(255)
visitor_last_name CHARACTER VARYING(255)
visitor_id_type CHARACTER VARYING(255)
visitor_id_number CHARACTER VARYING(255)
created_date CHARACTER VARYING(255)
users
id SERIAL
username CHARACTER VARYING(255)
password TEXT
role CHARACTER VARYING(20)
status CHARACTER VARYING(10)
created_at TIMESTAMP(6) WITHOUT TIME ZONE
updated_at TIMESTAMP(6) WITHOUT TIME ZONE
first_name CHARACTER VARYING(255)
last_name CHARACTER VARYING(255)
email CHARACTER VARYING(100)
address CHARACTER VARYING(255)
house_number CHARACTER VARYING(255)
phone_number CHARACTER VARYING(255)
qrcode CHARACTER VARYING(1000)
shift CHARACTER VARYING(50)
accesspoints CHARACTER VARYING(100)
entry_log
id SERIAL
scanned_in_at TIMESTAMP(6) WITHOUT TIME ZONE
scanned_out_at TIMESTAMP(6) WITHOUT TIME ZONE
resident_id INTEGER
security_id INTEGER
resident_vehicle
id SERIAL
user_id INTEGER
vehicle_make CHARACTER VARYING(50)
vehicle_model CHARACTER VARYING(50)
vehicle_color CHARACTER VARYING(20)
license_plate TEXT
notifications
id SERIAL
message TEXT
type CHARACTER VARYING(20)
status CHARACTER VARYING(20)
created_at TIMESTAMP(6) WITHOUT TIME ZONE
updated_at TIMESTAMP(6) WITHOUT TIME ZONE
user_id INTEGER
login_log
id SERIAL
user_id INTEGER
ip_address CHARACTER VARYING(255)
login_time TIMESTAMP(6) WITHOUT TIME ZONE
logout_time TIMESTAMP(6) WITHOUT TIME ZONE
status CHARACTER VARYING(20)
Powered by yFiles

Figure 1. Database Schema

Modules

Blacklist Module

Purpose: Manages the blocking and unblocking of visitors who have violated rules or are not welcome at a property.

Key Functionalities:

- Add to Blacklist: Allows admins or security to block a visitor based on their ID.
- Remove from Blacklist: Unblock a visitor and allow them to be scheduled again.
- View Blacklist: Retrieve a list of all currently blacklisted visitors.

Feedback Module

Purpose: Collects and manages feedback submitted by residents regarding their visitor experience or any complaints.

Key Functionalities:

- Submit Feedback: Residents can submit feedback on their visitor interactions.
- View Feedback: Admins can view all submitted feedback for analysis or review.

Users Module

Purpose: Handles authentication, roles, and profile data of all system users including admins, security, and residents.

Key Functionalities:

- Add/Edit/Delete Users: Admin can manage user records and assign roles (admin, security, resident).
- Authentication & Session Management: Uses next-auth for session-based login.
- Role-Based Access: Role ID system is in place for different permissions.

Visitors Model

Purpose: Allows residents to schedule, view, and manage visitor access while also enabling security to track visitor entry/exit.

Key Functionalities:

- Create Visitor Schedule: Residents can register expected visitors and set entry/exit times.
- View Visitor List: Retrieves visitor schedules per resident (ordered by entry date).
- Log Entry/Exit: Security logs entry and exit times for each visitor.Entry Log History: View all entry/exit logs for auditing.
- Validate One-time vs Recurring Visitors: Different workflows for temporary and frequent visitors.

API Documentation

Method	Endpoint	Description	Parameters	Response
GET	/api/auth/[...nextauth]	Fetch session status or redirect to sign-in provider	Cookies (for session), Query string (for callback info)	200 OK (session info) or redirects
POST	/api/auth/[...nextauth]	Handle sign-in/sign-out and callbacks	Request body depends on the action (e.g., credentials, OAuth data)	200 OK (session), 302 Redirect, or 401 Unauthorized

Table 1. Authentication API Endpoint

Method	Endpoint	Description	Parameters	Response
POST	/api/blacklist/create-update	Create a new blacklist entry or update an existing one based on type	resident_id, visitor_id, security_id?, reason, status, type (true for create, false for update), id (for update)	200 OK, 400 Bad Request, or 500 Server Error
DELETE	/api/blacklist/[id]	Delete blacklist entry by ID	id (number, required)	200 OK, 401 Unauthorized, or 500 Server Error
GET	/api/blacklist/list	Retrieve all blacklist entries	(Session required)	List of all entries or error
GET	/api/blacklist/list/:visitor_id	Get blacklist entry by visitor ID	visitor_id (URL param or function arg)	Single entry or null or error

Table 2. Blacklist API Endpoints

Method	Endpoint	Description	Parameters	Response
POST	/api/feedback/create	Create feedback for a visitor schedule	scheduleId (number), rating (number), comments (string)	200 OK, 400 Bad Request, or 500 Server Error
GET	/api/feedback/[id]	Checks if feedback exists for a visitor schedule. Return schedule if not.	id (schedule ID)	"exists" or schedule object or null
GET	/api/feedback/all	Retrieves all visitor feedbacks with schedule info	None	List of feedback or null

Table 3. Feedback API Endpoints

Method	Endpoint	Description	Parameters	Response
POST	/api/user/create	Registers a new user	username, password, role, first_name, last_name, etc.	201 Created, 409 Conflict, or 500 Internal Server Error
GET	/api/user/[id]	Fetch a single user	id (user ID)	User object or null
PUT	/api/user/[id]	Update user details	id, and user fields in data object	200 OK, 400 Bad Request, 404 Not Found, 500 Server Error
DELETE	/api/user/[id]	Delete a user record	id (user ID)	200 OK or error message
GET	/api/user/list	Fetch all users	None	List of user objects or error

Table 4. User API Endpoints

Method	Endpoint	Description	Parameters	Response
GET	/api/visitors/[id]	Fetches a visitor's details by their ID.	id (integer, required) – Visitor ID	200 OK – { id, first_name, last_name, phone, email, entry_logs, schedule } 404 Not Found – Visitor does not exist
GET	/api/visitors	Retrieves all visitors, excluding blacklisted ones.	None	200 OK – [{ id, first_name, last_name, id_type, id_number }] 500 Server Error
GET	/api/visitors/blacklist-check/{id}	Checks if a visitor is on the blacklist.	id (integer, required) – Visitor ID	200 OK – "exists" if blacklisted, visitor details otherwise 404 Not Found – Visitor does not exist
POST	/api/visitors/create/individual	Creates an individual visitor entry and schedules a visit.	resident_id (integer, required) visitor_first_name (string, required) visitor_last_name (string, required) visitor_phone (string, required) visitor_id_type (string, required) visitor_id_number (string, required) visitor_email (string, required) status (string, required) visitor_type (string, required) visitor_entry_date (string, required, format: YYYY-MM-DD) visitor_exit_date (string, required, format: YYYY-MM-DD) license_plate (string, required) comments (string, optional) sg_type (integer, required)	200 OK – { success: true, message: "Visitor created successfully", visitorScheduleId } 400 Bad Request – Missing fields 403 Forbidden – Visitor is blacklisted 500 Server Error

POST	/api/visitors/create/group	Creates a group visitor entry and schedules a visit.	resident_id (integer, required) visitors (array, required) – Each visitor has: → visitor_first_name (string, required) → visitor_last_name (string, required) → visitor_id_type (string, required) → visitor_id_number (string, required) visitor_phone (string, required) visitor_email (string, required) status (string, required) visitor_type (string, required) visitor_entry_date (string, required, format: YYYY-MM-DD) visitor_exit_date (string, required, format: YYYY-MM-DD) license_plate (string, required) comments (string, optional) sg_type (integer, required)	200 OK – { success: true, message: "Group visitors scheduled successfully", visitorScheduleId } 400 Bad Request – Missing fields or invalid visitor list 403 Forbidden – One or more visitors are blacklisted 500 Server Error
PUT	/api/visitors/update/individual/[id]	Updates an individual visitor schedule.	id (integer, required) – Schedule ID resident_id (integer, required) visitor_phone (string, required) visitor_email (string, required) status (string, required) visitor_type (string, required) visitor_entry_date (string, required, format: YYYY-MM-DD) visitor_exit_date (string, required, format: YYYY-MM-DD) license_plate (string, required) comments (string, optional)	200 OK – { success: true, message: "Schedule updated successfully" } 400 Bad Request – Missing fields 500 Server Error
PUT	/api/visitors/update/group/[id]	Updates a group visitor schedule, adding/removing visitors as needed.	id (integer, required) – Schedule ID resident_id (integer, required) visitor_phone (string, required) visitor_email (string, required)	200 OK – { success: true, message: "Schedule successfully updated." } 400 Bad Request – Missing fields

			status (string, required) visitor_type (string, required) visitor_entry_date (string, required, format: YYYY-MM-DD) visitor_exit_date (string, required, format: YYYY-MM-DD) license_plate (string, required) comments (string, optional) visitors (array, required) – List of visitors to update	500 Server Error
DELETE	/api/visitors/delete/[id]	Deletes a visitor schedule and its linked visitors.	id (integer, required) – Schedule ID	200 OK – { success: true, message: "Schedule deleted successfully" } 500 Server Error
POST	/api/visitors/entry-exit	Handles logging entry/exit or updating status for one-time and recurring visitors.	visitorId (string or number), scheduleId (string or number), securityId (string or number), action ("logEntry" | "logExit" | "updateOneTime"), entryChecked?, exitChecked?, visitorExit, status	success, code, message, optional data. E.g. "Recurring visitor exit logged." or error codes like 400, 403, 404, 500
GET	/api/visitors/entrylog (function: visitorsLog)	Retrieves all visitor entry logs (currently not filtered by resident).	None	Array of entry log records, each including visitor details.
GET	/api/visitors/list (function: visitorsInfo)	Fetches all visitor schedules for the currently logged-in resident.	None (user ID is taken from session)	Array of visitor schedule records including visitor info, ordered by entry date (desc).

Table 5. Visitor API Endpoints

Code Structure

src/ (Main source code folder)

- **app/** – Next.js App Router structure; includes route folders (e.g., login, dashboard) with layout.tsx, and route pages.
- **components/** – Feature-based reusable UI components.
- **hooks/** – Custom hooks like useIsMobile for responsive behavior.
- **lib/** – Shared config or utilities.
- **__tests__/** – Contains automated test files.

prisma/

- **schema.prisma** – Defines your database schema.
- **migrations/** – Tracks database changes using Prisma migrations.

public/

- Static assets served directly (e.g., **file.svg**, **globe.svg**, etc.).

reports/

- Test or build reports (e.g., **junit.xml** for CI/CD test results).

Root Configuration Files

- next.config.ts, tailwind.config.ts, postcss.config.mjs – App and styling setup.
- tsconfig.json – TypeScript config.
- jest.config.js – Testing config.
- middleware.ts – Auth or redirect logic.
- package.json – Project metadata and dependencies.
- README.md – Project usage and overview.

Testing Information

To ensure the application functions as intended, several key test cases were designed and executed. These tests were focused on verifying the core modules of the Gated Community system, including resident registration, visitor check-ins, gate access logging, and data validations.

Summary of Test Cases

	Test Case	Description	Expected Result	Actual Outcome	Status
1	Register New Resident	Add a new resident with valid data	Resident added and listed	Successfully added	✅Passed
2	Duplicate Email on Registration	Try registering with an existing email	Error message shown, no record added	Validation error triggered	✅Passed
3	Create Visit Request	Submit a visit request for a guest	Visit appears in pending visits	Record created with status	✅Passed
4	Visit Approval by Guard	Approve a scheduled visit	Visit status updated to "Approved"	Status updated correctly	✅Passed
5	Invalid Visit (no resident match)	Submit visit with unknown resident	Form shows validation error	Prevented and error shown	✅Passed
6	Add Feedback	Resident submits feedback after visit	Feedback stored and linked to resident	Feedback logged	✅Passed
7	View Feedback History	Admin views feedback submissions	Feedback list loads properly	All records visible	✅Passed
8	8. Blacklist Visitor	Admin blacklists a visitor	Visitor is marked as "blacklisted"	Entry blocked for future visits	✅Passed
9	Attempted Visit by Blacklisted Person	Blacklisted visitor tries to check in	Check-in is denied	Access blocked and logged	✅Passed
10	View Visit History	Resident/admin views visit logs	Correct chronological records shown	Accurate logs displayed	✅Passed
11	Edit User Info	Admin edits resident details	Data updates successfully	Info updated in database	✅Passed
12	Delete User	Admin deletes a resident	Resident removed from system	Record deleted properly	✅Passed

Table 6. Summary of Application Testing

- All form fields were validated for correct input types and required values.
- Blacklisting logic prevents both visit creation and gate access for flagged individuals.
- System timestamps were checked for accuracy in visit logs and feedback records.
- The feedback system ties responses to specific residents and visit instances.
- Historical data is preserved even when users are edited or removed, ensuring auditability.

All primary modules of the Gated Community application performed as expected under manual testing. Key flows such as visit creation, user administration, feedback, and security actions (like blacklisting) worked smoothly. The system is stable and prepared for user deployment, pending optional stress and UI testing in larger environments.