**SRS & Procedure Analysis**

**Problem**: The call and confirm process in estates is slow and flawed in terms of security.

**Current Strategy Steps**:

* Visitor arrives at destination and meets security at the gate.
* Awaits a call from the resident to security to allow.
* Resident gives information on the visitor.
* Visitor is allowed to go into the estate

**Flaws**:

* Creates Traffic in entrances due to the slow process
* Could possibly affect returning residents
* Violates social distancing rules (still related to traffic)

**Proposed System**:

A platform where residents can register their guests prior to an arrival and security can access the records for verification when the guest arrives.

**Strategy Steps:**

* After arrangement, resident registers expected guest and fills in necessary information on the application. (Should visitor data be stored permanently?)
* When visitor arrives, he informs security on the address (block number)
* Security checks the records for that address, verifies guest’s entry.
* Guest is allowed to proceed.
* On departure security unchecks visitor
* Entries are stored for reference

**Flaws:**

* May not be ‘automatic’ enough and seamless as querying data may consume time especially if personnel aren’t properly trained.
* Tedious for operative personnel

**Alternative Procedure:**

* Resident applies for visitor (on-boarding for visitors)
* Enters details including contact details.
* Specify time frame of visit
* 24-hour token is generated on the specified day
* Token/Ticket is sent to visitor’s email.
* Processed at entry by QR code reader either deployed or from security personnel mobile phone account.
* Valid? Add visitor to logs and store time of entry/ acceptance.
* Optional step for resident to confirm arrival
* As long token has been used for entry without exit, it remains active.
* On departure, token is fulfilled and deleted from ‘activeTokens’ table database but its reference can remain in the main tokens table for reference.

This method would be more effective as it would be able to accommodate visitor-initiated requests efficiently, is less person-dependent and significantly faster.

**Proposed Stack**

* **Node.js** (Runtime environment)
* **NestJS** (Server-side framework)
  + **TypeORM**: For mapping code to database (so we can go through without writing any SQL or creating any tables on postgres. Just typescript.
  + **Nunjucks**: Render Engine for templates
  + **Swagger**: Automatic API documentation (If necessary) \*\*
  + **Fastify**: ‘Fastify can be a better choice when you place high value on very fast performance.’ – NestJS documentation.
  + **QR-code**: <https://thecodebarbarian.com/creating-qr-codes-with-node-js.html> - Library for generating QR codes.
* **React Native** (Frontend framework): React-Bulma components?

**Backend Modules:**

*Definition of terms:*

***MVC architecture****: Model–view–controller is a software design pattern commonly used for developing user interfaces that divides the related program logic into three interconnected elements. This is done to separate internal representations of information from the ways information is presented to and accepted from the user.*

***Module****: Component or part of a whole.*

***Model****(Entity): Where data is captured on an item in database.*

***Controller****: Controller acts as an interface between View and Model. Controller intercepts all the incoming requests.*

***Service****: The business logic or code which would be executed through the controller.*

*View: Frontend components of application.*

* **User Entity**: ID (generated), name, number, **email**, Role etc.
  + A User can be:
    - A Security Admin (The people standing by to process visitors):
    - A Visitor
    - A Resident
  + ***They will all have entities of their own***
* User Controller & Service
* **Homes module** (entity, controller service): For managing various house addresses, will be related to resident (*residentHomeAddress) &* Visitor (*HomeAddressVisited).*
* **Resident Module** (Related to User, Security Admin and visitor)
* Security Admin Module (Related to User (is a user), Resident(*authorizedVisitToWhom*) & **Visitor**(whoWasAuthorized): For security admin app based token scanning if on-ground qr scanners are busy or not functional (power outage)
* **VisitLog**: Visitors and their respective residents as well as *SecurityAdminOfficerThatAuthorized* (when app based scanning is used) can be stored here for reference. It is related to all the above modules.
* **RolesModule**: A module to facilitate role-based authentication to certain endpoints in order to restrict some functionality and accessibility to *admins* or any other party such as *Residents*.
* **AuthModule**: A module to facilitate authentication all through the application. Plays a big part in role-based-auth with the help of roles module.
* **JwtStrategy:**  A module backed by passport.js to specify authentication strategies in our application and to set up javascript web token (JWT) strategies for passing credentials which the server compares with its ‘permissions’ for authentication.

***Functionality for each Party***

**User**:

*Refers to the base functionality for a verified user that does not have any role.*

* Sign In
* Change Password
* Search for estate \*\*
* Sign Up (Visitor ONLY)

*There will be no sign up mechanism for residents as the estate or we (if employed) will handle the creation (and deletion) of resident accounts. Same applies to security admins, a superadmin or pre-exisiting security admin would handle their account creation.*

**Resident:**

*Resident role functionality.*

* Sign In
* Change Password
* Register Visitor
* Request Visit log

**Visitor:**

* Sign In
* Change Password
* Apply for visit

***Relationships***

|  |  |
| --- | --- |
| **Module** | **Relationships** |
| Auth |  |
| Global |  |
| Homes | Residents (Many To Many) |
| Residents |  |
| Roles |  |
| Security Admins |  |
| Users |  |
| Visitors |  |

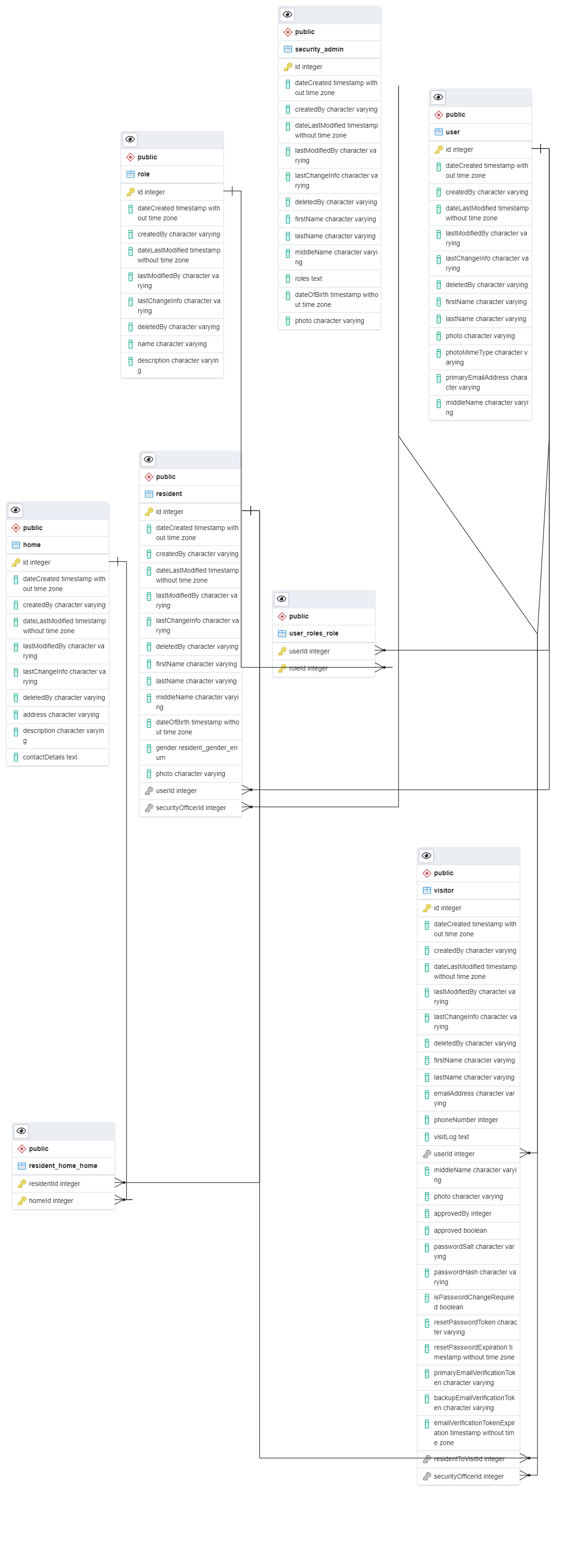


Figure 1: Entity Relationship Diagram

***Dependencies***

"dependencies": {

"@nestjs/common": "^7.0.0",

"@nestjs/core": "^7.0.0",

"@nestjs/mapped-types": "\*",

"@nestjs/platform-express": "^7.0.0",

"@nestjs/platform-fastify": "^7.6.15",

"@nestjs/swagger": "^4.8.0",

"dotenv": "^8.2.0",

"fastify-cookie": "^5.3.0",

"fastify-multipart": "^4.0.3",

"nodemailer": "^6.5.0",

"reflect-metadata": "^0.1.13",

"rimraf": "^3.0.2",

"rxjs": "^6.5.4",

"typeorm": "^0.2.31"

},

"devDependencies": {

"@nestjs/cli": "^7.0.0",

"@nestjs/schematics": "^7.0.0",

"@nestjs/testing": "^7.0.0",

"@types/express": "^4.17.3",

"@types/jest": "26.0.10",

"@types/node": "^13.9.1",

"@types/supertest": "^2.0.8",

"@typescript-eslint/eslint-plugin": "3.9.1",

"@typescript-eslint/parser": "3.9.1",

"eslint": "7.7.0",

"eslint-config-prettier": "^6.10.0",

"eslint-plugin-import": "^2.20.1",

"jest": "26.4.2",

"prettier": "^1.19.1",

"supertest": "^4.0.2",

"ts-jest": "26.2.0",

"ts-loader": "^6.2.1",

"ts-node": "9.0.0",

"tsconfig-paths": "^3.9.0",

"typescript": "^3.7.4"

},

***Resources***

* [***https://auth0.com/blog/developing-a-secure-api-with-nestjs-adding-authorization/#Set-Up-****API-Authorization*](https://auth0.com/blog/developing-a-secure-api-with-nestjs-adding-authorization/#Set-Up-API-Authorization)
* [***https://typeorm.io/#/many-to-one-one-to-many-relations***](https://typeorm.io/#/many-to-one-one-to-many-relations)
* [***https://orkhan.gitbook.io/typeorm/docs/relations***](https://orkhan.gitbook.io/typeorm/docs/relations)
* ***https://auth0.com/blog/adding-salt-to-hashing-a-better-way-to-store-passwords/#:~:text=A%20cryptographic%20salt%20is%20made,the%20salts%20for%20each%20user.***