Data Engineering and Distributed Systems
Community-Based University Space Monitoring System: Is It Full?

January 2025

University of Primorska, FAMNIT; Dmytro Tupkalenko (89252101); GitHub repository

1 Introduction

This paper presents a system designed to assist university students in checking the availability of commonly used public spaces, including libraries, study rooms, and other places like cafeterias. Students often face uncertainty regarding the occupancy of shared study spaces. A wrong answer to a common question - "Is the library full right now?" - leads to inefficient planning, wasted travel time, and frustration when preferred study areas or other "university-associated" spaces are at their capacity. The proposed system aims to provide community-based platform to answer this question.

Existing Solutions

There are already several products that address similar problem. The ones we could find are either sensor-based, or some variation of reservation-systems.

1. *Sensor-Based:* OpenRoom, Waitz
   These systems rely on hardware installed in specific rooms or buildings to automatically track occupancy.
   - **Advantage:** The occupancy data is precise up to precision of the "sensor data" - the precision can be controlled by improving the hardware.
   - **Disadvantage:** Requires installation of physical hardware, meaning adoption depends on university management approving and funding the deployment.
   - **Disadvantage:** Monitoring is limited to spaces where sensors are installed; adding new rooms requires additional hardware.
   - **Disadvantage:** Only university-owned spaces can typically be monitored. If students study in third-party locations (e.g., coworking spaces) - those areas cannot be included unless an agreement is made with the space owner to install hardware.
   - **Disadvantage:** Works only for supported universities, and in order to add a new university - the university management should contact the provider.
   - **Disadvantage**: These systems have usage fee + costs of hardware.
2. *Reservation-Based:* MIDAS, Whatspot
   These systems allow users to reserve spaces and see the time when the space is reserved.
   - **Advantage:** Occupancy can be known in advance through reservations.
   - **Disadvantage:** These systems are limited to spaces that can be reserved. Open or informal study areas (e.g., general library seating) are unsuitable for reservations - consequently cannot be supported.

There are also solutions for individual universities. An example is the study-space availability system used at University College London - however they generally share all of the above mentioned advantages and disadvantages (except the price, as they are implemented at the organisations that use them).

Our Solution

We propose a **crowd-sourced, community-managed system** that addresses the limitations of existing approaches. Our solution does not rely on hardware installation, university buy-in, or formal reservation protocols. Instead - it relies on the student community to create a user-generated dashboard of study space availability. The application's primary function is to allow users to quickly check or report whether a study space is **free**, **busy**, or **full**.

The system operates on the following principles:

- **User-Generated Spaces:** Users should be the one who define the spaces. That is the spaces that are tracked are user defined only.
- **Crowd-Sourced Status Updates:** The availability of each space is determined by the most recent user reports. Upon arriving at or leaving a location, a user can instantly update its status with a single tap.
- **Recency:** Each space's listing displays not only its current status but also the timestamp of the last update.
- **Community Closeness:** The data for spaces can be seen and updated by the community members only.
- **Community Incentives:** To ensure accuracy of the data, the system incorporates reputation mechanisms.

This report is structured in the following way: Section 2 describes the requirements of our system - both functional and technical. Section 3 justifies the choices of the selected technologies for our project. Section 4 describes architecture of the solution (framework, database schema, diagrams, design patterns). Section 5 discusses advantages and disadvantages of our approach. Section 6 summarizes the achieved work, main learning and suggests future work.

2 Requirements

Functional Requirements

The system shall provide the following functionalities (the ones marked with *, have some additional points of discussion that will be addressed in Section 5):

1. **University Management***
   - Users shall be able to submit a request to add a new university to the platform.
   - Approved universities shall have their own dedicated space management interface, which can be accessed by authenticated users (that belong to the university) only.
2. **Space Management***
   - Authenticated Users shall be able to **Read/View** all spaces in their university with occupancy status. Each space shall display: current occupancy status, last updated timestamp, location and space type-specific details.
   - Authenticated users shall be able to **"Create"** new study spaces within their university's domain.
   - Users shall be able to **"Update"** space information (name, location, type) and occupancy status.
   - Users shall be able to **"Delete"** spaces.
3. **User Management**
   - Users shall be able to register and login with email and password. Email and Hashed Password is to be stored in the persistent layer.
   - Users shall only be able to register and login using a valid university email address (with a specified university email address domain).
   - The system shall verify email ownership through an email confirmation.
   - Users shall only access and modify spaces belonging to their verified university domain.
   - Users shall be able to manage their profile information (change password, delete account).
4. **Occupancy Reporting***
   - Users shall be able to report occupancy status (Free/Busy/Full) for any space in their university.
5. **User Interface Requirements**
   - The system shall provide a list-based view of study spaces.
   - The interface shall clearly indicate data recency (e.g., "Updated 5 minutes ago").

Technical Requirements

The system shall adhere to the following technical specifications:

1. **Records Management** – All records shall be stored in a relational database, structured with the following core tables:
   - Universities table
   - Spaces Table
   - Users table
2. **University Management** – The Universities table shall serve as the central table. All existing spaces and users must be linked to the Universities table via foreign keys.
3. **Space Management** – The spaces shall be implemented using the Composite design pattern. For example, if a library (a study space) contains child spaces (e.g., a study room on the first floor, a study room on the second floor), the system must allow all required functionality (e.g., updating availability) to be called on the child spaces, and the parent record (the library) must stay synchronized with the state of its children.
4. **User Management** – The Users table shall store the raw email address. For security, only a hashed password may be stored; raw passwords must never be stored.
5. **Occupancy Reporting** – Occupancy shall be reported by updating a corresponding field in the Space table record. Occupancy data must be retrievable through a database query.
6. **Responsive Interface** - Interface shall be adapted to the device used.

3 Technological choices

Framework: Django

Django, a high-level Python web framework, was selected as the development platform for this project.

Advantages For Our Project

- **Appropriate Scope Match:** The project's requirements - a simple CRUD operations, and a single primary interface (the dashboard) - align well with Django's "batteries-included" philosophy.
- **Rapid Development:** Django's feature set, including its admin interface, authentication system, and form handling, significantly accelerates development time.
- **Robust ORM:** Django's Object-Relational Mapping (ORM) system provides an intuitive, Pythonic interface for database operations. This simplifies implementation of the Database specific parts of code of our project (specifically Composite design pattern (see Subsection MODEL: Composite Design pattern) - which without ORM would be harder to implement using plain SQL.
- **Built-in Security Features:** Django includes protection against common web vulnerabilities (CSRF, XSS, SQL injection) by default, which is essential for our system where the major parts of it depend on user input.
- **Migration Path:** If future requirements demand greater flexibility (support of mobile apps), the codebase can be relatively quickly refactored toward API + frontend Microservice architecture with DjangoREST framework - which is an extension of Django to develop REST APIs.

Considered Disadvantages For Our Project

- **Monolithic Architecture:** Django's coupled structure (backend + frontend) may introduce unnecessary complexity if the system needs to scale, though this is negligible given the project's current scale.
- **Performance Overhead:** The framework's abstraction layers and synchronous nature may introduce worse latency compared to lighter frameworks, though this is negligible given the project's current scale.

Database: PostgreSQL

The database selection process mainly considered PostgreSQL and MariaDB, since both are fully open-source and provide native support through the Django ORM. **PostgreSQL** was chosen for this system for these reasons:

Advantages For Our Project

- **Native Support for Hierarchical Data:** Our *Space* model implements a composite pattern via a recursive `parent` foreign key, forming a tree structure. PostgreSQL provides first-class support for querying such hierarchies using **Recursive Common Table Expressions (CTEs)**, allowing for operations like fetching all descendants of a space - a task that is central to our application's logic.
- **Robust Concurrency via MVCC:** Features like occupancy update make it necessary to handle simultaneous reads and writes. PostgreSQL's is built on **Multi-Version Concurrency Control (MVCC)**, which allows high-throughput operations with minimal locking conflicts.
- **Strict Standards Compliance & Data Integrity:** While both databases support foreign key actions like *CASCADE*, PostgreSQL's stricter adherence to SQL standards ensures more predictable and reliable enforcement of relational integrity. This is critical for maintaining the clean dependency hierarchy (`University` → `Space`).
- **Django Integration:** As mentioned before, PostgreSQL is the recommended production database for Django applications, ensuring optimal ORM compatibility.[1]

Considered Disadvantages For Our Project

- **Operational Complexity & Initial Overhead:** For an initial prototype, a lighter database like SQLite might have sufficed. However, given the relative complexity of our data model and the cost of a future database migration, adopting PostgreSQL from the beginning in our opinion pays-off in the future.

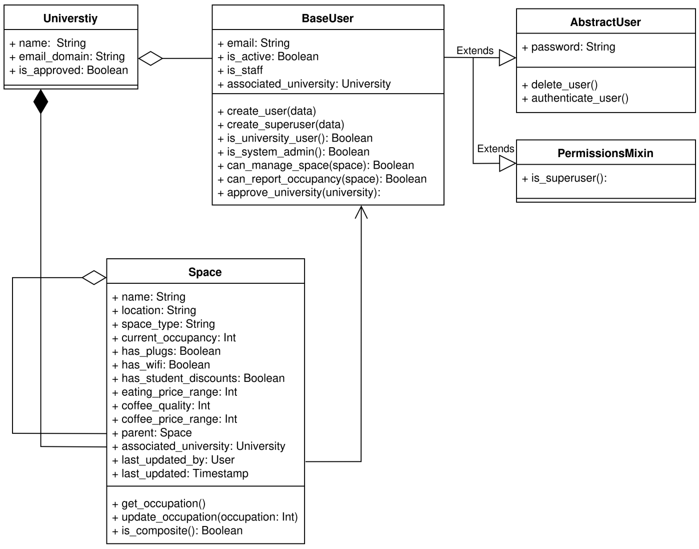
4 Architecture

Architectural Design Pattern: MVT

The system follows Django's **Model-View-Template (MVT)** pattern, a variant of the Model-View-Controller (MVC) design pattern. MVT enforces separation of concerns by dividing the application into three interconnected layers:

- **Model Layer:** The Model layer encapsulates the data structure and business logic. Our implementation defines three core entities - *University*, *Space*, and *User* - using Django's Object-Relational Mapping (ORM). Key architectural decisions at this layer include:
  - The Composite design pattern implementation for the *Space* model, enabling hierarchical parent-child relationships.
  - Database constraints and validation logic to ensure referential integrity.
  - Business rules, such as occupancy calculation algorithms that aggregate child space occupancy when querying composite spaces.
- **View Layer:** The View layer serves as the application's control mechanism, processing HTTP requests and coordinating between Models and Templates. Views implement:
  - Authentication and authorization checks (e.g., verifying users can only access spaces within their university domain).
  - Request handling for CRUD operations on spaces and occupancy reporting.
  - Data retrieval and transformation before passing to Templates.
  - Form validation and error handling.
- **Template Layer:** The Template layer handles presentation logic through server-side rendered HTML. Templates receive context data from Views and render dynamic content. This layer includes:
  - Responsive UI components that adapt to different screen sizes.
  - Conditional rendering based on user authentication state and permissions.
  - Reusable template inheritance structures (e.g., `base.html`) that other templates can built on top of.

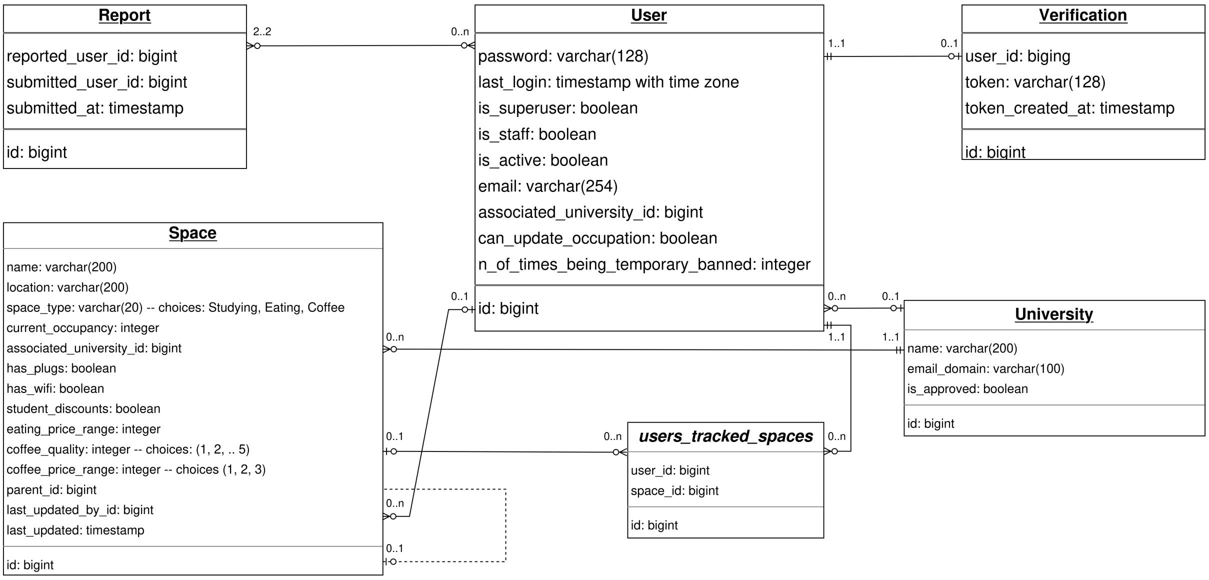
MODEL: Class Diagram + Database Schema



BaseUser is a concrete user class inheriting from two Django built-in classes, which provide authentication and administration functionality. It has an **aggregation (weak)** relationship with `University` via the `associated_university` attribute (the weak association is needed to allow system administrators to exist independently of any university).

Space represents a university space and has a **composition (strong)** relationship with `University`, meaning it cannot exist without one. It also has a **self-aggregation** relationship through the `parent` attribute, implementing the Composite pattern (see Subsection MODEL: Composite Design pattern). Additionally, **Space** is **unidirectionally associated** with `BaseUser` via the `updated_by` attribute, which records the user responsible for the latest occupancy update.

University represents the universities registered in the application.



- **Space** ↔ **University** (`0..n` ↔ `1..1`) Each `University` instance can have 0 or more associated instances of `Space`. Each `Space` instance belongs to exactly 1 `University` instance (via `associated_university_id`).
- **Space** ↔ **Space** (`0..1`) Each `Space` instance can have 0 or 1 parent `Space` instances.
- **Space** ↔ **User** (`0..n` ↔ `0..1`) Each `User` instance can update 0 or more instances of `Space`. Each instance of `Space` can be last updated by either 0 or exactly 1 instances of `User` (via `last_updated_by_id`).
- **User** ↔ **University** (`0..n` ↔ `0..1`) Each `University` instance can be associated with 0 or more instances of `User`. Each `User` instance is linked to at most 1 instance of `University` (via `associated_university_id`) - 0 if the user is admin, 1 if the user is a general user.
- **User** ↔ **Verification** [**MODEL + VIEW: Authentication System**] (`1..1` ↔ `0..1`) Each `Verification` instance can be associated with exactly one `User` (via user_id field). Each `User` is linked to at most one `Verification`.
- **User** ↔ **Reports** [**Potential Misuse Scenarios**] (`0..n` ↔ `2..2`) Each `Report` instance can be associated with exactly 2 instances of `User` (via reported_user_id and submitted_user_id fields). Each `User` instance can participate in many reports.
- **User** ↔ **Space** [**TEMPLATE: Dashboard**] (`0..n` ↔ `0..n`) Captured through users_tracked_spaces.

MODEL: Composite Design pattern

The *Space* model implements the Composite pattern to support hierarchical space structures. This allows modeling scenarios such as a library (parent) containing multiple reading rooms (children) or a building floor with several study spaces. The implementation uses a self-referential foreign key (can be seen in the DB schema). The interface is common for both parent and children:

- `get_occupancy()`: Recursively calculates average occupancy from child spaces. Or retrieves the value if the Space is not composite.
- `get_all_descendants()`: Traverses the tree structure to retrieve all nested child spaces. t

MODEL + VIEW: Authentication System

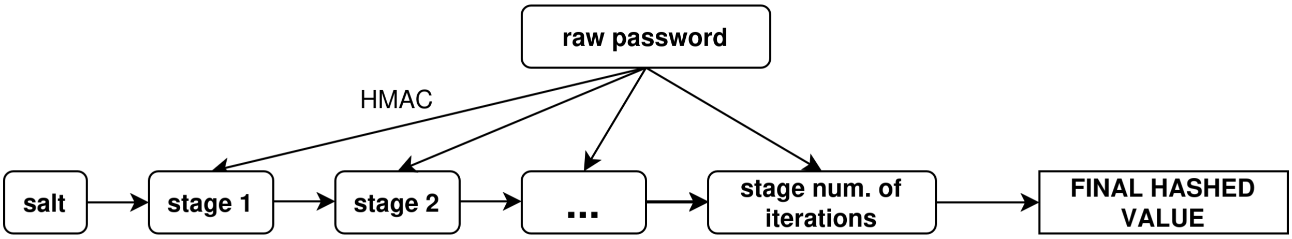
We begin by describing the built-in authentication and password management system provided by Django, which serves as the foundation of the overall authentication architecture. Then we outline the complete authentication flow and discuss what security is achieved with the described authentication architecture.

Built-in Authentication System of Django

Django uses the **PBKDF2 (Password-Based Key Derivation Function 2)** algorithm with HMAC-SHA256 hashing by default.[2] [3]. Passwords in Django are stored using the following format:

`<algorithm>$<iterations>$<salt>$<hash>`
e.g. `pbkdf2_hmac_sha256$720000$random_salt$hash_output` (default for current version)

- **Algorithm**: Specifies the hashing algorithm used:
  - SHA-256 is the hashing algorithm
  - HMAC extends the hashing by taking two arguments (previous step hash + password) and encrypting the previous step hash by using password as the secret key - salt is used at the first stage.
- **Iterations**: The number of times the hashing algorithm is applied. This makes brute-force attacks significantly more expensive (especially in the way that the encryption is now sequential, because HMAC takes hash computed at the previous stage as an argument - so it is harder to parallelize).
- **Salt**: A "cryptographically random" string unique to each password. It is used to ensure that identical passwords produce different hashes, preventing rainbow table attacks. [4]
- **Hash**: The final output of applying PBKDF2-HMAC-SHA256 to the combination of password, salt, and iterations. This is the value actually compared during authentication.



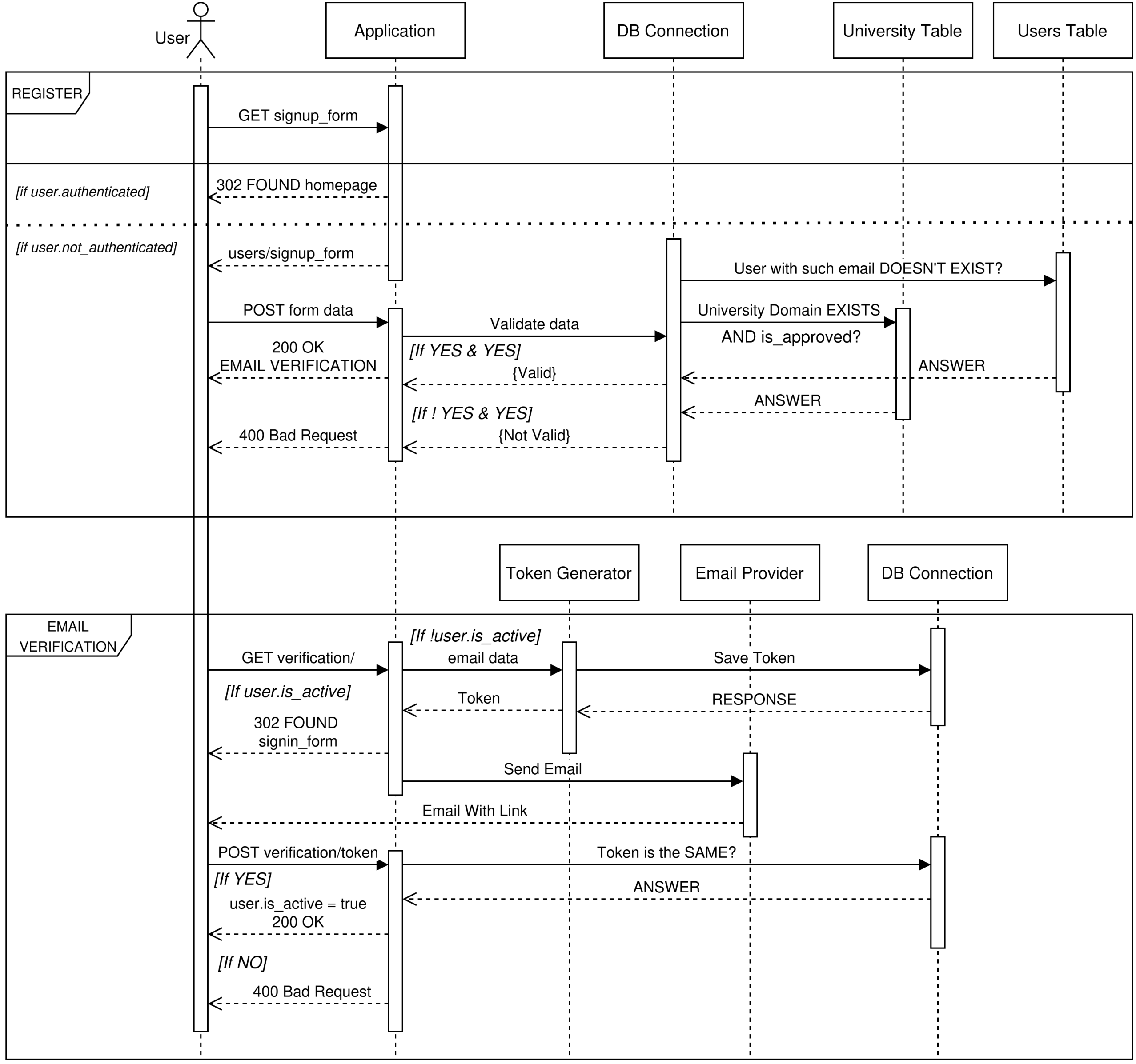
Authentication Process When a user attempts to sign in, Django:

1. Retrieves the stored password hash from the database
2. Extracts the algorithm, iterations, and salt from the stored string
3. Applies the same hashing process to the submitted password using the extracted parameters
4. Performs a constant-time comparison between the newly computed hash and the stored hash to prevent timing attacks [5]

Implementation in Our System Our system uses Django's `AbstractBaseUser` class, which automatically integrates this password management system. The `User` model also inherits password validation methods that enforce minimum complexity requirements (minimum 10 characters, no common passwords, non-numeric-only).

Authentication Flow

The authentication system implements a two-phase verification process to ensure secure user registration with university email validation. The following sequence diagram shows the complete authentication process:



Phase 1: User Registration When a user initiates registration by accessing the signup form:

1. **Initial Request**: The user sends a GET request to `users/signup_form`
   - If the user is already authenticated, the system redirects (302) to the homepage
   - If not authenticated, the application returns the signup form (200 OK)
2. **Form Submission**: The user submits registration data via POST request containing email and password
3. **Server-Side Validation**: The Application validates the submitted data through database queries:
   - *Email Uniqueness Check*: Queries the Users Table to verify no existing account uses this email
   - *University Domain Verification*: Queries the University Table to confirm a university with the matching email domain exists and has `is_approved = true`
4. **Validation Outcomes**:
   - *If validation fails* (user exists OR university doesn't exist/isn't approved): Database returns negative response, and Application returns 400 Bad Request to user
   - *If validation succeeds* (new email AND approved university exists): Database confirms validity, Application proceeds to email verification phase, and returns 200 OK with email verification notice

Phase 2: Email Verification After successful form validation, the system initiates email verification:

1. **User Verification**:
   - User clicks verification link in email
   - Browser sends GET request to `verification/` endpoint (if user still inactive)
   - If user is already active, system redirects (302) to signin form
2. **Token Generation**:
   - Application forwards email data to Token Generator component
   - Token Generator creates a unique verification token
   - Token is returned to Application
3. **Token Persistence**:
   - Application sends token to DB Connection for storage
   - Database stores token associated with the pending user account
   - Confirmation response returned to Application
4. **Email Dispatch**:
   - Application sends verification email request to Email Provider
   - Email Provider delivers email containing verification link to user
   - The verification link includes the generated token
5. **Token Validation**:
   - User submits verification token via POST to `verification/token`
   - Application queries DB Connection to validate token
   - Database confirms token matches stored value
6. **Account Activation**:
   - *If token is valid*: User account status updated (`is_active = true`), Application returns 200 OK, and user can now sign in
   - *If token is invalid*: Application returns 400 Bad Request, and user must request new verification email

Achieved Security

The described authentication system achieves the following security measures: first Django's built-in password hashing ensures passwords are managed securely. Second, a two-step email verification process restricts account access to users with valid university email addresses only, ensuring that only authorized members can access the corresponding university dashboard.

We thus achieve the architecture where independent users have their data stored securely, and where users of the dashboard can be sure that the data they share (description of spaces, occupation) is shared only among users who belong to the community.

TEMPLATE: Dashboard

The application's main user interface is organized into two distinct dashboard views. The **public dashboard** that displays all currently active spaces within a given university - that is, all spaces that have been created by users and have not been deleted. This provides a community-wide overview of spaces. The **private dashboard** that presents a personalized view for each authenticated user, listing only the spaces that they have explicitly chosen to track or created themselves. This is done through the junction table users_tracked_spaces depicted in the DB diagram.

Each dashboard presents spaces in a list-based layout. Spaces are visually distinguished by their occupancy state (Free / Busy / Full), with clear indicators of the last update timestamp, and other descriptive information.

5 Discussion

Potential Misuse Scenarios

NOTE: This subsection can be considered a part of functional requirement section, but since a major parts are up to refinement through user feedback - it is put in the discussion section.

As described previously, both the dashboards and the list of supported universities are populated and maintained by the user community. As a result, the system may be exposed to misuse, such as denial-of-service–style behavior or the submission of inaccurate or misleading information.

(DDOS): denial-of-service-style attacks

To mitigate denial-of-service–style attacks [6] in which users repeatedly submit requests to add new universities, the system should enforce a limit on the number of pending university submissions. In practice, this can be implemented by capping the number of unapproved universities stored at any given time, for example by maintaining pending requests in an in-memory cache (e.g. by maintaining a "cache layer" through Redis).

A similar form of abuse is possible with respect to the spaces within a university. To address this, the system should impose a limit on the number of spaces that can be created by a single user (e.g. 20) - this can be implemented by a form validation rule where we first check the aggregated count of the created_by field of Spaces for the given university.

Misreport

First potential source of misreport arises from the ability of users to edit existing space metadata, such as amenities (e.g. Wi-Fi availability) or pricing information. To prevent incorrect or malicious modifications, the system should employ a community-based validation mechanism. Proposed edits would be submitted as suggestions and only applied after receiving approval from a sufficient proportion of users who track the given space (e.g. 10%) - implemented in cache layer (Redis).

A second form of misuse concerns the intentional or negligent misreporting of occupancy status. To mitigate this, the system should include a reporting mechanism that allows users who observe an incorrect occupancy update to flag it within a limited time window (e.g. within five minutes after submission of occupation). If a user receives multiple verified reports (e.g. 3), their ability to submit occupancy updates should be temporarily suspended by disabling the reporting field - these time-based suspensions can be enforced via scheduled background tasks (e.g. a cron job), and repeated temporary bans (e.g. 3 occurrences) should result in a permanent restriction on submitting occupancy updates - both temporary and permanent bans can be triggered on user report and the type can be determined through the aggregation on corresponding records.

Implementation Plan

Since the application can be modularized into multiple Django applications, the implementation can be planned by taking into account the dependencies between these modules:

(core) - application containing the central business logic, (users + universities) - application managing user accounts and universities data respectively, (reports) - application that implements the community rules and security mechanisms. The core application can itself be subdivided into pre-authentication components (such as public views) and post-authentication components (such as the main dashboard and space management). Since the authenticated core features rely on a fully functional user and university system, their development must follow. Similarly, the reports system and other misuse preventions (e.g., rate limiting with Redis or scheduled tasks for ban management) can be implemented only on top of the core user-space interactions they are designed to monitor.

Therefore, the implementation should proceed in the following order:

- Build the pre-authentication core, alongside the foundational **User** and **University** models.
- Implement user authentication, including email verification.
- Develop the authenticated core features for university and space management.
- Add the reports system and misuse prevention measures.

6 Conclusion

In this paper we described a community-based system for university space management that aims for balance between the freedom of users, and the security with anti-misuse measures. We considered all the main, from the author's point of view, architectural and technological aspects; discussed potential flaws and points of failure of the system and established the foundation for future work. In general, we believe that similar considerations can be applied to a broader range of space tracking systems, that are not limited just to universities; some parts of it, especially the security and community incentives, can be used in any system where the major parts of it are left to the users to manage.

Key takeaways that can be applied to other community-based systems are the need to balance user freedom with mechanisms that prevent misuse, and the importance of choosing appropriate authentication and verification methods. When users are given control over shared data, some level of accountability (such as basic reputation, rate limits, or community reporting) becomes essential to keep the system reliable. Similarly, in contexts where access should be limited to a specific group, simple email/password authentication is often not enough; lightweight identity checks, such as domain-based email verification, can significantly improve trust without adding much complexity. These ideas provide a useful starting point for extending and refining the system in future iterations.

For future work, the thresholds and numerical parameters for community incentives, such as the number of votes required to validate edits or the frequency limits for occupancy updates should be carefully determined trough user feedback or experiments. Additionally, even though we did some considerations on how to "bound" the system to prevent misuse and to promote fair use, it can definitely be improved and extended. Finally, although this is a "technical" consideration, extending the application to a mobile interface is likely a necessary future step, as we expect most users to prefer accessing the product on their phones rather than via a web browser. However, before development begins, this assumption should be thoroughly validated and carefully reconsidered.

References

[1] *Database configuration*. Django Software Foundation. URL: https://docs.djangoproject.com/en/6.0/ref/databases/.

[2] *Password management in Django*. Django Software Foundation. URL: https://docs.djangoproject.com/en/stable/topics/auth/passwords/.

[3] *PKCS #5: Password-Based Cryptography Specification Version 2.0*. Internet Engineering Task Force (IETF). URL: https://datatracker.ietf.org/doc/html/rfc2898.

[4] *Password Storage Cheat Sheet*. OWASP Foundation. URL: https://cheatsheetseries.owasp.org/cheatsheets/Password_Storage_Cheat_Sheet.html#salting.

[5] Paul C. Kocher. "Timing Attacks on Implementations of Diffie-Hellman, RSA, DSS, and Other Systems". In: *Advances in Cryptology — CRYPTO '96* (1996). DOI: 10.1007/3-540-68697-5_9.

[6] Saman Taghavi Zargar, James Joshi, and David Tipper. "A Survey of Defense Mechanisms Against Distributed Denial of Service (DDoS) Flooding Attacks". In: *IEEE Communications Surveys & Tutorials* (2013). DOI: 10.1109/SURV.2013.031413.00127.