Database

GWIIT

General Workplace Inventory & Inspection Tool

# Introduction

## Purpose of the Database

This document outlines the database design and structure for the General Workplace Inventory & Inspection Tool (GWIIT) web application. The application will utilize three separate PostgreSQL databases to manage different aspects of the system on release, corresponding to the 'users', 'tickets', and 'fire\_extinguishers' Django apps. The databases are separated to correspond to the different web apps to allow for growth of the overall project as new Django apps are released.

## Scope

At release, the web ap will utilize three separate databases outlined below:

Users Database: Manages user information, roles, and permissions.

Tickets Database: Handles ticket management, including creation, assignment, and tracking.

Fire Extinguishers Database: Manages the inventory and inspection records of fire extinguishers, along with compliance tracking.

### Design Reasoning

Each Django app has its’ dedicated database to maintain a clear separation of concerns, ensuring modularity, ease of maintenance, and scalability.

### Data Integrity and Security:

#### Front-End Validation:

HTML5 Validation: HTML5 attributes like ‘required’, ‘pattern’, ‘minlength’, and ‘maxlength’ on input fields offer basic validation directly in the browser.

Alpine.js: Handles simple validation logic, such as showing error messages or disabling the submit button until the form is correctly filled out.

#### Backend Validation:

Form Validation: Django provides a ‘forms’ framework that includes built-in validation. When a user submits a form, Django automatically checks the data against the form's validation rules before processing. This includes validating field types, enforcing required fields, and applying custom validation logic.

Model Validation: Django also supports validation at the model level. Model validation occurs when data is created or modified outside of form submission, such as through APIs, batch imports, or admin interfaces.

Custom Validators: Django allows definition of custom validators for specific fields. Custom validators can enforce complex validation rules that go beyond basic checks.

Error Handling and User Feedback: Django can detect invalid data during form or model submission, and prevent the data from being saved, while providing detailed error messages. Errors are returned to the user, allowing correction to input.

Security and Consistency: By handling validation on the server side, common security issues, such as SQL injection or cross-site scripting (XSS) attacks, are avoided by Django since all data entering the system is consistently validated on the backend.

Integration with the Database: Django's validation is integrated with its ORM (Object-Relational Mapping) layer, meaning any constraints defined at the database level (e.g., unique fields, foreign key constraints) are respected and enforced by Django. This integration ensures that the data in the database remains consistent and reliable.

### Normalization and Constraints:

#### Normalization:

For the GWIIT web application, normalization is applied to ensure that databases are organized efficiently. Data is stored only once, which simplifies updates and maintains consistency across the system.

Key Normalization Practices:

First Normal Form (1NF): Ensures that each table column in the databases contains atomic values, making the data straightforward to query and manage.

Second Normal Form (2NF): Builds on 1NF by ensuring that all non-key columns are fully dependent on the primary key, eliminating partial dependencies.

Third Normal Form (3NF): Used to ensure that each non-key column is dependent only on the primary key, preventing data anomalies.

Benefits:

Data Integrity: Normalization maintains accurate data across all databases by eliminating duplicate data and ensuring that related data is properly linked.

Simplified Updates: With less redundancy, making changes to the data is easier and less error-prone.

#### Database Constraints:

Constraints are critical to ensuring that the data stored in each database is valid and consistent. Constraints prevent the entry of incorrect or incomplete data, which is vital for maintaining data compliance and accurate reporting within the application.

Types of Constraints Used:

UNIQUE and NOT NULL Constraints: Applied to fields like usernames in the 'users' database or ticket IDs in the 'tickets' database to ensure that these critical identifiers are always unique and present.

PRIMARY KEY and FOREIGN KEY Constraints: Ensure that relationships between tables, such as linking a ticket to a user or a fire extinguisher to its location, are maintained correctly.

CHECK Constraints: Used for fields like ticket priority or fire extinguisher status to ensure that only valid values are entered, enhancing data reliability.

ENUM Constraint: Used to restrict the possible values of a field to a predefined set. This is particularly useful for fields that should only accept specific values, such as statuses or categories.

### Data Migration:

It is possible that a larger client of the web app will have their own database for employees. It would be logical to create a connection to this external database to feed the ‘users’ database in the web app. Some considerations would need to be made first, but the connection is possible and would assist in integrating the web app into use for the web app client.

#### Assess Existing Database:

Database Structure: Analyze structure of existing employee database. Identify the tables, columns, data types, and relationships that correspond to the data for 'users' database.

Data Quality: Evaluate the quality of the existing data, checking for inconsistencies, missing values, or outdated records to avoid migrating flawed data into the web app database system.

#### Mapping Data Fields:

Field Mapping: Create a map between the fields in the client’s database and the ‘users’ database. For example, map the client's ‘employee\_id’ to the ‘user\_id’ in the ‘users’ database. Ensure that all necessary fields in the ‘users’ database are accounted for in the client's database.

Handling Differences: Address any differences in data types or structures. For instance, if the client’s database uses a different format for dates or stores user roles differently, data transformation will need to be done for migration.

#### Data Transformation and Validation:

Transformation Logic: Develop scripts or use ETL (Extract, Transform, Load) tools to transform the data from the client's format to the format required by 'users' database. This might include converting date formats, normalizing names, or merging multiple fields into one.

Validation Rules: Implement validation checks to ensure that the transformed data meets the constraints and requirements of the ‘users’ database, such as ensuring all usernames are unique and not null.

#### Integration Approaches:

Direct Integration: If the client's database will continue to be used and updated alongside the web application database, consider a direct integration approach. This could involve:

Database Views: Create database views that allow the web application to query the client's existing database directly, rather than duplicating data.

APIs: Use APIs to interact with the client’s database, pulling in employee data as needed.

Data Synchronization: If a direct integration is not feasible, set up a synchronization process where data from the client's database is periodically imported into the 'users' database. This could be done on a schedule or triggered by specific events.

#### Handling Integration with Django

##### Setting Up Database Connections:

Multiple Database Configuration: Django supports multiple databases, allowing connection to the client’s existing database alongside the web application’s database. This is configured in Django’s ‘settings.py’ file, where the database connections are defined using Django's DATABASES setting.

Defining Database Routers: Django allows definition of database routers that determine which database to use for specific models. This is useful for keeping the 'users' database separate but still interacting with the client’s database.

##### Managing Data Access with Models:

Custom Models for Client Data: Custom Django models that map to the tables in the client’s database. These models are read-only so the client’s database will not be modified by the web application.

Views and API Integration: For API-based integration, create Django views that pull data from the client's database via the API, process it, and then store it in the 'users' database as needed. Django's ‘requests’ library or third-party packages like ‘django-rest-framework’ can facilitate this interaction.

##### Data Synchronization Process:

Scheduled Tasks with Celery: If implementing periodic data synchronization, use ‘Celery’ with Django for managing scheduled tasks. Celery allows automating the data synchronization process, ensuring that the 'users' database stays up-to-date with the client’s database.

Django Management Commands: For manual control, create Django management commands that trigger synchronization processes. These commands can be executed as needed or set up as chronological jobs for regular execution.

Routing and URL Configuration:

URL Routing: Define URLs in Django’s ‘urls.py’ that correspond to views handling the data integration. For instance, routes like ‘/sync-users/’ that trigger a synchronization process or ‘/fetch-employee-data/’ that pulls in real-time data from the client’s database.

Middleware for Real-Time Sync: If real-time synchronization is required, middleware can be implemented to intercept certain requests and fetch the latest data from the client’s database before proceeding.

#### Security and Compliance:

Data Privacy: Ensure that migration and integration processes comply with data privacy laws and regulations. This includes securing data during transfer and ensuring that only authorized personnel have access to sensitive information.

User Access Control: Ensure that user roles and permissions are correctly mapped and enforced, preserving the integrity of the client's access control policies.

#### Testing and Verification:

Testing: Before going live with a client, conduct thorough testing of the migration process. This includes verifying that all data has been correctly migrated, that relationships between tables are intact, and that the application behaves as expected with the integrated data.

User Acceptance Testing (UAT): Involve the client in the testing process to ensure that the integrated system meets their expectations and that all necessary data is correctly reflected in your application.

#### Ongoing Maintenance:

Monitoring and Auditing: Ensure that the data integration continues to function correctly over time. Set up auditing to track any changes made to the data post-migration.

Handling Updates: Plan for updates to either the web application or client’s database to prevent data inconsistencies or integration failures.

# Database Overview

Database Name: ‘users’

Database Management System (DBMS): PostgreSQL

Version Information: 16.4 (released on 09/14/2023)

Database Name: ‘tickets’

Database Management System (DBMS): PostgreSQL

Version Information: 16.4 (released on 09/14/2023)

Database Name: ‘fire\_extinguisher’

Database Management System (DBMS): PostgreSQL

Version Information: 16.4 (released on 09/14/2023)

## Hosting Environment:

For early development, the databases are being hosted locally on the development machine. For later development and release there are options for hosting:

### On-Premises Hosting:

**Environment:** Databases are hosted on physical servers within the client organization.

#### Advantages:

**Control:** Complete control over hardware and software, including security, backups, and network configurations.

**Compliance:** Easier to meet specific regulatory requirements (e.g., data residency).

#### Considerations:

**Cost:** Higher costs for hardware, maintenance, and IT staff to the client.

**Scalability:** Limited by physical hardware; scaling may require additional infrastructure.

**Maintenance:** Requires in-house expertise to manage and maintain the servers.

### Cloud-Based Hosting:

**Options:**

**AWS RDS for PostgreSQL:** Amazon’s managed database service that offers automated backups, scaling, and patching.

**Google Cloud SQL for PostgreSQL:** Google’s managed service with similar features to AWS RDS.

**Azure Database for PostgreSQL:** Microsoft’s managed service with built-in security, automated backups, and scaling options.

#### Advantages:

**Scalability:** Easily scale resources up or down based on demand.

**Flexibility:** Access databases from anywhere with internet connectivity.

**Managed Services:** Providers handle backups, patching, and high availability.

**Ease of Use:** Managed services handle operational overhead, including automated backups, scaling, patching, and monitoring.

**High Availability:** Built-in redundancy and failover options to ensure uptime.

**Security:** Providers offer security features, including encryption at rest and in transit.

#### Considerations:

**Ongoing Costs**: Regular expenses based on usage (compute, storage, data transfer).

**Data Privacy:** Compliance with data privacy regulations.

**Vendor Lock-In:** Dependence on the service provider’s infrastructure and APIs.

**Customization:** Less flexibility in configuration compared to self-hosted options.

### Hybrid Hosting:

**Environment:** A combination of on-premises and cloud-based hosting.

#### Advantages:

**Flexibility:** Keep sensitive data on-premises while leveraging the cloud for scalability and redundancy.

**Cost Optimization:** Use on-premises resources for steady workloads and cloud resources for peak demands.

#### Considerations:

**Complexity:** Managing a hybrid environment can be complex, requiring integration between on-premises and cloud resources.

**Data Synchronization:** Ensuring data consistency across environments may require sophisticated strategies.

# Design Reasoning

## Strategic Separation of Databases

Modularity and Scalability: The decision to separate the ‘users’, ‘tickets’, and ‘fire\_extinguishers’ databases is strategically aligned with the project’s long-term goals. This modular approach allows each part of the system to grow independently, facilitating easier updates, maintenance, and potential scaling without affecting other components.

Specialized Functionality: Each database is designed to handle specific aspects of the application, ensuring that the data structure and constraints are optimized for the unique requirements of user management, ticketing, and the functions of additional Django web applications.

## Performance Optimization

Load Distribution: By distributing data across multiple databases, the system can better manage load, ensuring that no single database becomes a bottleneck. This separation is particularly important as the application scales, allowing targeted performance tuning for each database.

Indexing and Query Efficiency: Key fields within each database are indexed to ensure fast query response times. This is especially crucial for high-frequency operations like retrieving user details, searching tickets, or generating compliance reports.

## Flexibility and Future-Proofing

Adaptability to Client Needs: The database design is flexible enough to integrate with external systems if needed, such as a client's existing employee database. This ensures that the application can adapt to various deployment environments and client requirements.

Growth and Expansion: As the GWIIT web app evolves, the current design allows for the addition of new Django apps and corresponding databases without disrupting the existing architecture. This future-proofing ensures that the application can expand its functionality over time.

# Database Schema

## Tables Overview

### ‘Users Tables’

**‘users’:** Stores user account information

**‘roles’:** Defines user roles within the application.

**‘permissions’:** Specifies the permissions associated with each role.

**‘user\_roles’:** Maps users to their respective roles.

**‘role\_permissions’:** Maps roles to their associated permissions.

**‘mfa\_devices’:** Stores information about multifactor authentication devices associated with each user.

**‘sessions’:** Tracks user sessions, including login and logout times.

**‘user\_preferences’:** Stores user-specific preferences, such as theme, language, and other settings.

**‘audit\_logs’:** Keeps a record of important actions performed by users.

### Users Reference Tables

**‘job\_titles’**: Stores all possible job titles.

**‘departments’**: Stores all possible departments.

**‘locations’**: Stores all possible locations where users can be assigned.

**‘schedule\_codes’**: Stores predefined work schedules.

### ‘tickets’

**‘tickets’**: Table for storing information about each ticket.

**‘comments’**: Records comments made on tickets.

**‘attachments’**: Manages files attached to tickets.

**‘categories’**: Defines categories for ticket classification.

**‘priorities’**: Stores the priority levels available for tickets.

**‘tags’**: Manages tags associated with tickets.

**‘ticket\_tags’**: Junction table to manage the many-to-many relationship between tickets and tags.

**‘ticket\_history’**: Logs changes and actions taken on each ticket.

**‘sla\_policies’**: Stores the service level agreements (SLAs) that define the expected timeframes for ticket resolution.

**‘ticket\_dependencies’:** Manages dependencies between tickets, where one ticket might block another until it is resolved.

### ‘fire\_extinguishers’

**‘fire\_extinguishers’:** Manages inventory of fire extinguishers, including details like type, location, and inspection history.

**‘inspections’:** Records inspection details for each fire extinguisher.

**‘photos’:** Manages photos taken during inspections or maintenance activities for fire extinguishers.

**‘annotations’:** Stores any annotations or additional notes related to photos or inspection tasks.

**‘locations’:** Manages the locations where fire extinguishers are placed.

**‘tasks’:** Tracks tasks related to fire extinguisher maintenance and compliance.

**‘compliance\_reports’:** Generates reports based on inspection and compliance data.

**‘audit\_trails’:** Logs significant actions related to fire extinguisher records, such as updates to status, inspections, or location changes.

## Table Schema – ‘users’

Provide detailed descriptions of each table, including columns, data types, constraints, relationships, and references.

‘id’

Column ID: ‘id’

Type: UUID

Constraints: Primary Key, NOT NULL

Description: A unique identifier for each user.

‘username’

Column ID: ‘username’

Type: VARCHAR(255)

Constraints: NOT NULL, UNIQUE

Description: The username used by the user to log into the system.

‘email’

Column ID: ‘user\_email’

Type: VARCHAR(255)

Constraints: NOT NULL, UNIQUE

Description: The user's email address.

‘password\_hash’

Column ID: ‘password\_hash’

Type: VARCHAR(255)

Constraints: NOT NULL

Description: A hashed version of the user's password.

‘first\_name’

Column ID: ‘first\_name’

Type: VARCHAR(255)

Constraints: NOT NULL

Description: The user's first name.

‘last\_name’

Column ID: ‘last\_name’

Type: VARCHAR(255)

Constraints: NOT NULL

Description: The user's last name.

‘phone\_number’

Column ID: ‘phone\_number’

Type: VARCHAR(20)

Constraints: NULL

Description: The user's phone number.

‘is\_active’

Column ID: ‘is\_active’

Type: BOOLEAN

Constraints: NOT NULL, DEFAULT TRUE

Description: A flag indicating whether the user's account is active.

‘last\_login\_at’

Column ID: ‘last\_login\_at’

Type: TIMESTAMP WITH TIME ZONE

Constraints: NULL

Description: The timestamp of the user's last login.

‘created\_at’

Column ID: ‘created\_at’

Type: TIMESTAMP WITH TIME ZONE

Constraints: NOT NULL, DEFAULT CURRENT\_TIMESTAMP

Description: The timestamp when the user account was created.

‘updated\_at’

Column ID: ‘updated\_at’

Type: TIMESTAMP WITH TIME ZONE

Constraints: NOT NULL, DEFAULT CURRENT\_TIMESTAMP ON UPDATE CURRENT\_TIMESTAMP

Description: The timestamp of the last update to the user's account.

‘manager\_id’

Column ID: ‘manager\_id’

Type: UUID

Constraints: NULL, Foreign Key (users.id)

Description: The ID of the user's manager. This can be used for assigning tickets, managing approvals, and sending notifications.

‘job\_title’

Column ID: ‘job\_title’

Type: VARCHAR(100)

Constraints: NULL

Description: The user's job title, which may be relevant for role-based access control or reporting.

‘department’

Column ID: ‘department’

Type: VARCHAR(100)

Constraints: NULL

Description: The department to which the user is assigned, helping to manage roles, permissions, and reporting.

‘assigned\_location’

Column ID: ‘assigned\_location’

Type: VARCHAR(100)

Constraints: NULL

Description: An alphanumeric code representing the building or site the user is assigned to. This can be useful for filtering users by location.

‘badge\_id’

Column ID: ‘badge\_barcode’

Type: VARCHAR(255)

Constraints: NULL, UNIQUE

Description: The user's badge barcode ID, typically used for physical access control or identification.

‘rfid\_tag’

Column ID: ‘rfid\_tag’

Type: VARCHAR(255)

Constraints: NULL, UNIQUE

Description: The user's RFID tag, often used for quick identification or tracking within a facility.

‘employment\_start\_date’

Column ID: ‘employment\_start\_date’

Type: DATE

Constraints: NULL

Description: The date the user started employment, useful for calculating tenure and managing employee lifecycle events.

‘employment\_end\_date’

Column ID: ‘employment\_end\_date’

Type: DATE

Constraints: NULL

Description: The date the user's employment ended, if applicable, useful for managing offboarding processes.

‘schedule\_code’

Column ID: ‘schedule\_code’

Type: VARCHAR(50)

Constraints: NULL

Description: A code representing the user's work schedule. This can be used to link to a schedules table that defines specific work hours or shifts.

‘schedule\_start\_time’

Column ID: ‘schedule\_start\_time’

Type: TIME

Constraints: NULL

Description: The time when the user's work schedule starts. Useful for notifications and SLA management.

‘schedule\_end\_time’

Column ID: ‘schedule\_end\_time’

Type: TIME

Constraints: NULL

Description: The time when the user's work schedule ends. This can help in determining when the user is available for tasks and notifications.

‘ooto’

Column ID: ‘ooto’

Type: BOOLEAN

Constraints: NOT NULL, DEFAULT FALSE

Description: A flag indicating whether the user is currently out of the office. This can be used to manage notifications and task assignments when the user is unavailable.

Entity-Relationship Diagram (ERD): Include a visual representation of the database schema.

H2: 5. Data Integrity and Validation

Front-End Validation

Backend Validation

H2: 6. Security and Access Control

User Roles and Permissions: Outline the roles and permissions within the database.

Authentication and Authorization Mechanisms: Describe how security will be enforced.

H2: 7. Backup and Recovery

Backup Strategy: Explain how and when backups will be performed.

Recovery Plan: Describe the process for restoring the database in case of failure.

H2: 8. Data Migration

Migration Plan: If applicable, outline how data will be migrated from existing systems.

Tools and Techniques: Describe any tools or scripts that will be used for migration.

H2: 9. Performance Considerations

Indexing Strategy: Describe the indexing approach to optimize queries.

Query Optimization: Discuss how queries will be optimized to ensure efficient data retrieval.

H2: 10. Audit and Logging

Audit Trail Requirements: Specify what actions need to be logged.

Log Retention Policy: Define how long logs will be retained.

H2: 11. Compliance and Reporting

Compliance Requirements: Identify any legal or regulatory compliance needs (e.g., GDPR, HIPAA).

Reporting Capabilities: Describe how the database will support reporting requirements.

H2: 12. Conclusion

Summary of Key Points: Recap the critical aspects of the database design.

Next Steps: Outline the next steps in the implementation process.

H2: 13. References

Citations: Include any references or sources used in the planning process.