**INSTITUTO POLITÉCNICO DE VISEU**
**ESTGV – Escola Superior de Tecnologia e Gestão de Viseu**



Engenharia Informática

Unidade Curricular: Base de Dados II

# Projeto Prático – 2025/2026

# Tema: **Fast Office**

Rodrigo Rolo estgv18757

David Gonzalez pv33971

Diego Alonso Laguillo pv33986

Viseu, 08/02/2026

# Índice



# Introduction – Project Overview

O objetivo principal desta ficha é trabalhar com interrupções e saídas/entradas digitais. Vamos desenvolver programas e construir circuitos de modo a implementar vários tipos de semáforos.

Na fase inicial vão ser criados semáforos “individuais”, isto é, sem nenhuma interação entre si.

Ao longo do trabalho, irão ser implementadas “melhorias”, de forma a criar um sistema mais complexo com interações entre os semáforos (por exemplo, ligar um enquanto se desliga outro).

# Architecture Overview

**ORM for User Management**

**PostgreSQL and Database-Level CRUD Logic**

**MongoDB for Notifications**

## ORM for User Management

User data is managed exclusively through the Django ORM. This choice is motivated by the requirements of authentication and authorization, which are central to user-related functionality. The ORM integrates directly with Django's security framework, making it well suited for handling login processes, role management, and permissions.

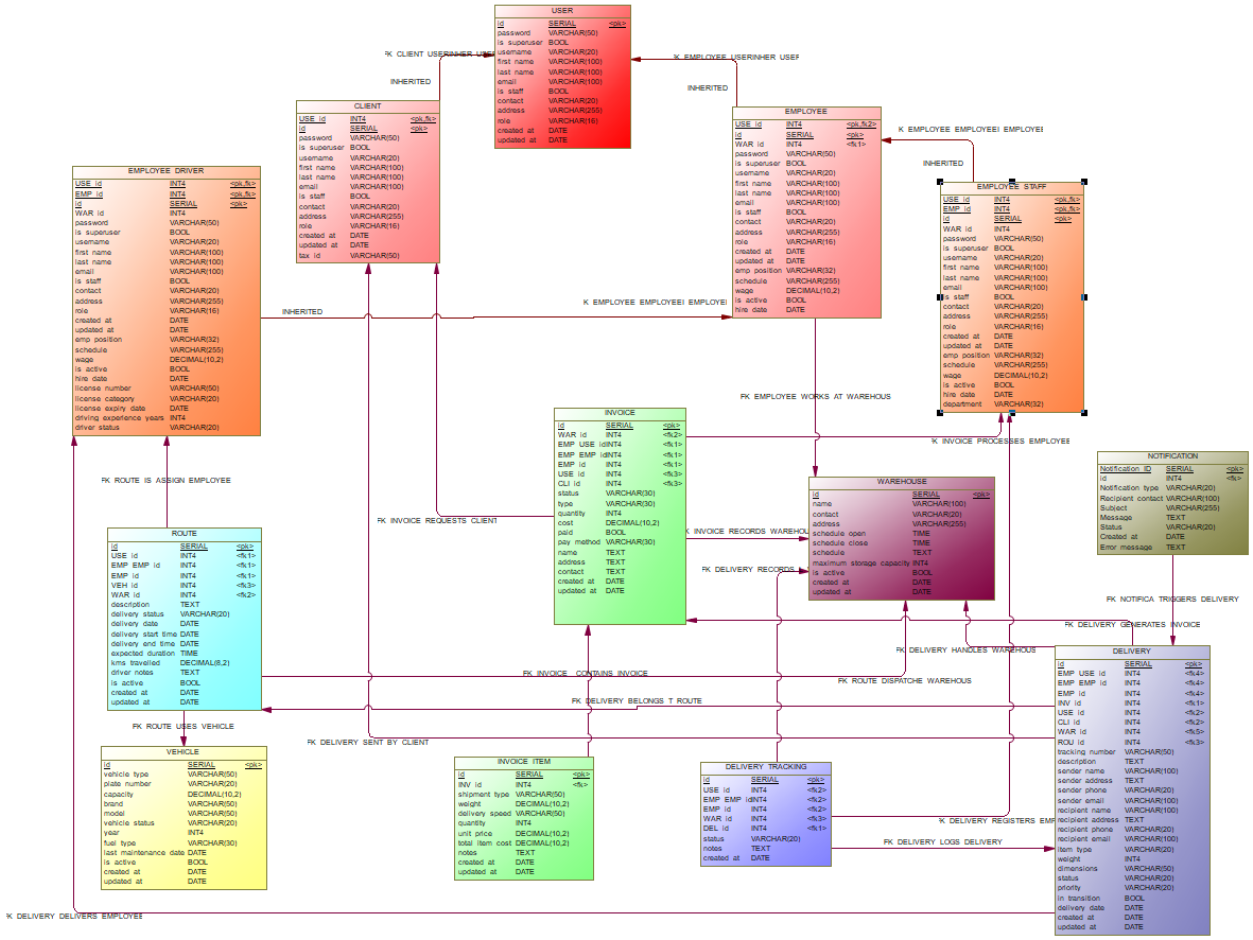
Using the ORM improves development efficiency for common user operations and reduces boilerplate code. Since user data is relatively stable and does not involve complex transactional logic, the ORM provides an appropriate level of abstraction. For this reason, its use is intentionally limited to the User entity, avoiding unnecessary complexity in other parts of the system.

## PostgreSQL and Database-Level CRUD LOGIC

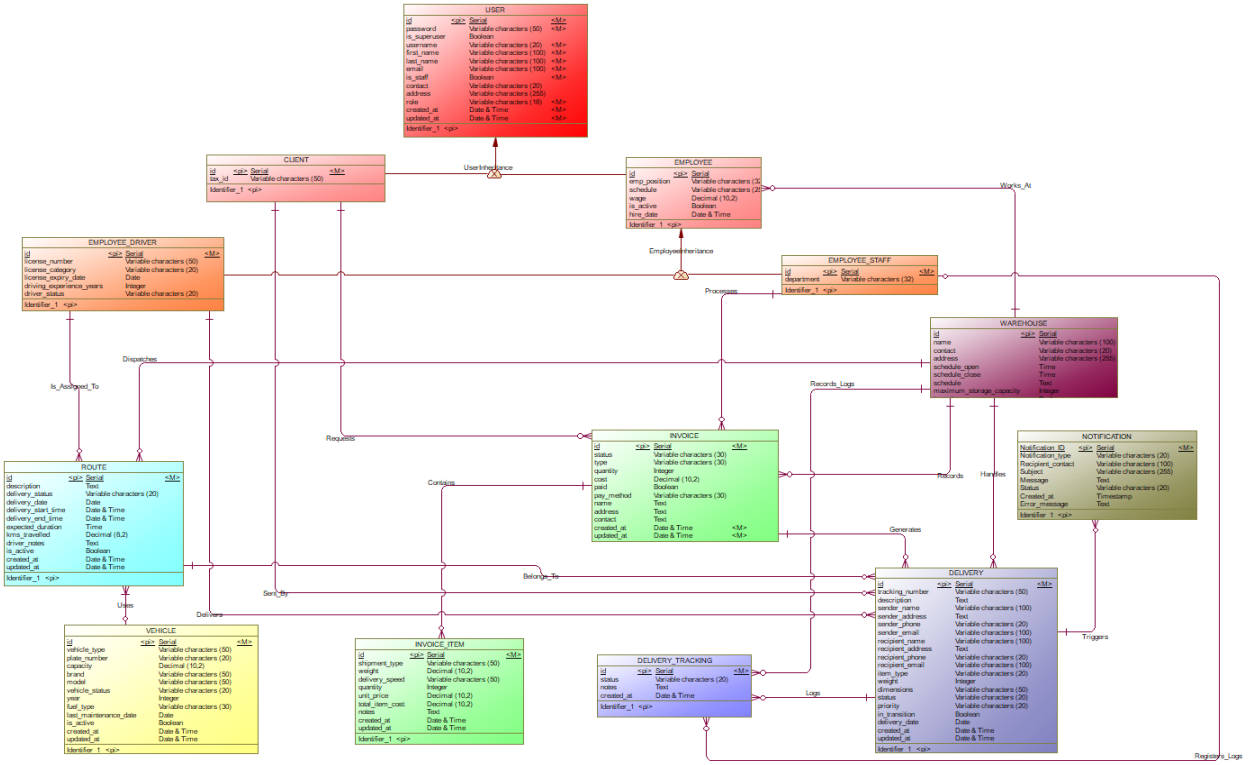
All other core entities are managed using PostgreSQL, where business logic is enforced directly at the database level. The system uses relational tables with strict integrity constraints, ensuring consistency between related entities.

CRUD operations are implemented through stored procedures, rather than direct table manipulation. This approach centralizes business rules within the database, making the system independent of the client implementation. Additional validations are handled using constraints, while triggers are employed where automated behavior is required.

This design improves performance for complex operations, prevents inconsistent or invalid data states, and increases robustness against incorrect API usage.

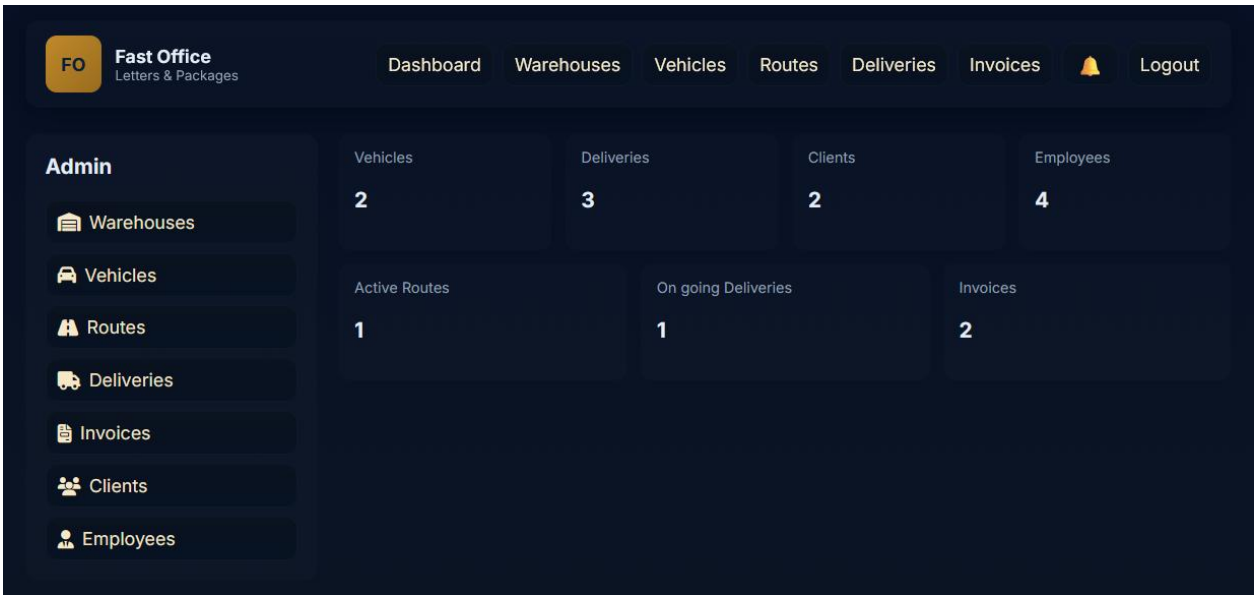


pdm

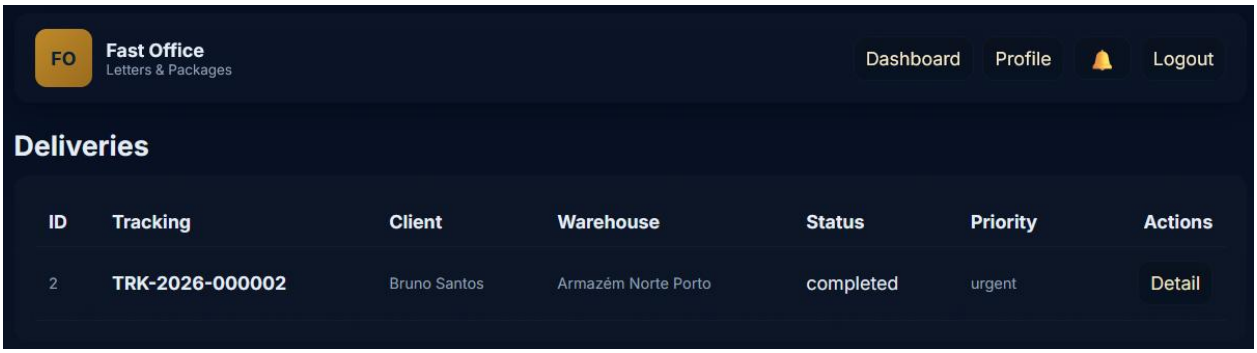


cdm

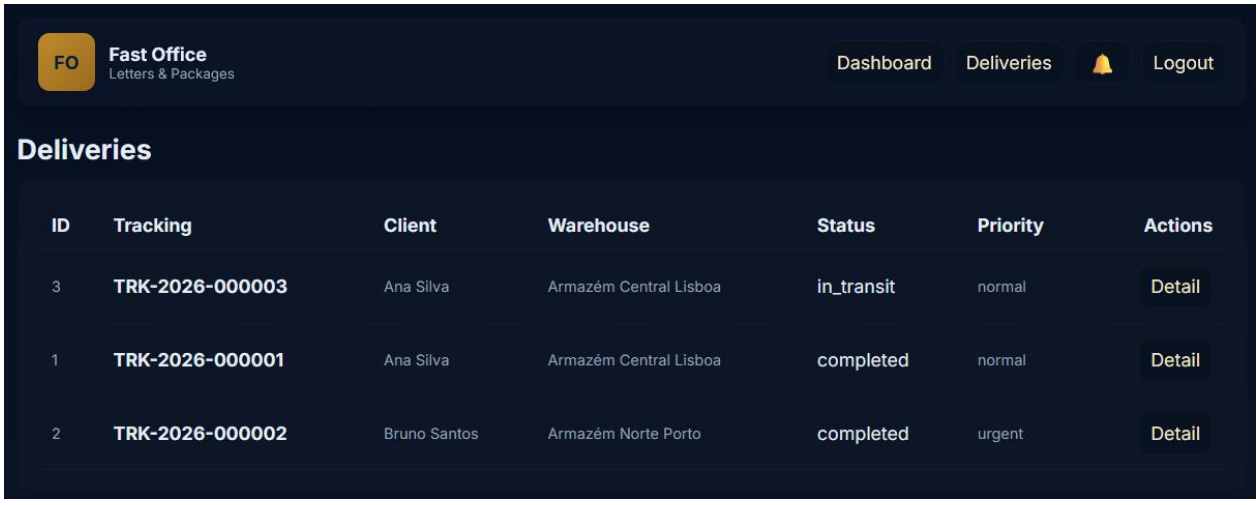
# Website Printscreens




Admin dashboard




Client dashboard




Driver Dashboard

# MongoDB for notifications

The notification system is implemented using MongoDB, which is used exclusively for this purpose. Notifications are semi-structured and event-driven, often requiring flexible schemas and additional metadata such as timestamps, read status, or custom payloads.

MongoDB is well suited for handling high write volumes and does not require strong relational constraints. Since notifications are not critical transactional data, they are intentionally decoupled from the relational database, keeping the core business logic clean and focused.

# Main Objects CRUD Analysis

## User

They represent the people who can log in and interact with the platform, and they carry the general profile and access role (admin/client/driver/staff/manager). In the project, a user becomes operational through role entities (Client / Employee).

### Read (R)

User reads are mostly about role access and administration. Instead of querying raw user rows everywhere, the database provides two views that make common read needs cleaner:

- v_clients acts as a ready dataset to list only users that behave as clients (useful for admin selection, invoice assignment, client listings, etc.). It avoids repeatedly filtering the same population at application level.

- v_potential_employees is used when the system needs to show who could become an employee. In practice, it supports UI flows where you pick a user and then create an employee profile for them, without mixing already-employed users in the same list.

So the read logic here is not heavy, but it is structured to reduce repeated filtering and keep role based selections consistent.

### Create (C)

User creation is centralized in sp_create_user, which acts as the database-level entry point for registering a new account. The main advantage of doing it through a procedure is that it keeps

creation consistent: a user gets created with the correct structure, and the application doesn't have to manually coordinate multiple inserts or default values.

In your Django layer, this is the operation behind user registration flows (or admin creation), and it's also the "first step" before creating employees.

**Update (U)**

User updates go through sp_update_user, which keeps profile changes and account updates consistent at database level (contact/address updates, role changes, activation-related fields, etc.). This matters because user data affects authorization decisions all over the system, so updates need to be centralized rather than scattered as random UPDATE "USER" statements.

There is also an important indirect update mechanism in the project: employee changes can automatically synchronize the user role. That is handled by the employee trigger trg_employee_sync_user_role (explained in Employee → Update), but it affects the User entity because it can update USER.role as a side-effect when someone becomes a driver or staff member.

**Delete (D)**

User deletion is supported through sp_delete_user. In systems like this, deletion is normally treated carefully because users are referenced by many operational records. Even when the UI does not expose a "delete account" button, having a controlled procedure is useful because it ensures deletion (or deactivation) follows a consistent rule instead of breaking history.

**User–Employee relationship**

The project clearly separates identity from operational roles.
A User represents a system identity capable of authentication and authorization, while an Employee represents an operational workforce role.

The relationship between User and Employee is optional and one-to-one:

- A user may exist without being an employee (e.g. client, admin).

- An employee must always be associated with exactly one user.

This design avoids role duplication and allows identity data (login, permissions) to evolve independently from operational employment data.

**Self vs administrative updates**

The project distinguishes between two types of user updates:

- Self updates, where a user edits their own profile data (name, contact information, address).

- Administrative updates, where an administrator manages users and employees through controlled procedures.

Self profile editing is intentionally limited for not sensitive fields and does not allow role changes or privilege escalation.
This ensures that authorization logic remains centralized and cannot be bypassed through the UI.

## Employee

Employees represent internal workforce profiles. While a user is identity, an employee is the operational role that participates in warehouses, logistics tasks, and internal processes. Employees carry work-related attributes such as position type (driver/staff), schedules, wage, activity status, and warehouse assignment.

**Read (R)**

Employee read operations are meant to be ready for the UI, because employee pages usually need identity labels plus operational context.

That is exactly what v_employees_full provides: a unified view that combines the employee record with the information needed to display it properly (typically the user's name and the warehouse context). In practice, this view is what you would use for employee listing pages, staff selection, and admin inspection.

**Create (C)**

Employee creation is handled by sp_create_employee, and this is one of the most important procedures in your "people" model because it doesn't only create an employee row: it also builds the correct specialization depending on the position.

If the employee is created as a driver, the procedure is responsible for inserting the corresponding driver profile data; if it is created as staff, it inserts the staff specialization row instead. That keeps the model consistent: you don't end up with drivers without driver data or staff without department.

On top of that, driver-specific validation is backed by the function fn_is_license_valid, which exists specifically to keep license rules centralized and reusable.

**Update (U)**

Employee updates are done via sp_update_employee, which centralizes changes like warehouse assignment, schedule, wage, activity status, and the employee position.

The key piece here is that the database also protects role consistency between Employee and User by using:

- trg_employee_sync_user_role (trigger)

- fn_sync_employee_user_role() (trigger function)

Whenever an employee is inserted or when their position changes, this trigger ensures the user's role stays aligned with that employment reality. In other words: if someone becomes a driver/staff operationally, the identity layer reflects that automatically, instead of relying on Django to remember to update roles in a separate query.

That design is very "database-project correct": business constraints are enforced where the data lives.

**Delete (D)**

Employee deletion is managed through sp_delete_employee. As with users, deletion is typically treated as "controlled removal" rather than freehand SQL, because employees are deeply tied to operational history (routes, tracking logs, invoice processing, etc.). A procedure ensures whatever deletion policy you follow remains consistent.

## Warehouse

Warehouses represent the physical storagement of the logistics system. They store operational information such as address/contact, schedule rules, capacity, active status, and audit timestamps. Warehouses are central because they provide location context for employees, deliveries, routes, and tracking logs.

**Read (R)**

Warehouse reads are built around views depending on the purpose:

- v_warehouses_full is the UI-friendly operational dataset (includes the relevant fields needed for listing and management pages).

- v_warehouses_export exists for export scenarios, keeping exports stable and structured independently from how the UI displays data.

That split is the same pattern you used in Deliveries: one “nice for UI”, one “flat for exports”.

**Create (C)**

Warehouse creation is centralized in sp_create_warehouse, which ensures that new logistics hubs are inserted consistently (including timestamps and required operational values). Using a procedure makes warehouse creation safer because it prevents incomplete warehouses from being created by accident.

**Update (U)**

Warehouse updates go through sp_update_warehouse, and they are also protected by schedule validation logic at the database level:

- trg_warehouse_schedule_check (trigger)

- fn_trg_warehouse_schedule_check() (trigger function)

This ensures that warehouse schedule information remains valid and consistent without relying purely on front-end validation. It’s a typical “DB integrity guardrail”: even if the UI lets something weird through, the database refuses invalid schedule data.

**Delete (D)**

Warehouse deletion is handled through sp_delete_warehouse. In logistics systems it’s common to avoid hard deletes; even if you remove a warehouse operationally, you usually want to keep historical deliveries and tracking readable. A controlled procedure allows you to implement that policy cleanly.

**Bulk Create (Import)**

Warehouses also support bulk loading via sp_import_warehouses, which is useful for populating environments quickly (initial dataset, migrations, testing). This mirrors your deliveries import approach: bulk operations are still centralized at DB level instead of being a pile of per-row inserts in application code.

## Deliveries

Deliveries are the core operational entity of the project. Each delivery represents a real shipment that gets registered, possibly assigned to a warehouse/route/driver, and then progresses through a lifecycle until it is completed or cancelled.

A key idea in the design is that the DELIVERY table holds the current snapshot of the shipment (current status, priority, sender/recipient details, assignments), while DELIVERY_TRACKING preserves the full history of how that status evolved over time. That separation is what makes it possible to show both "where it is now" and "how it got here", which is essential for logistics systems.

**Read (R)**

Reading deliveries is intentionally not done as raw table queries. The UI needs context (driver name, client name, warehouse, route status, etc.) and the system also needs role-based access (clients should only see their own deliveries, drivers only theirs). Because of that, read operations are built around functions and views that already return "ready-to-display" datasets.

For a client user, deliveries are retrieved through fn_get_client_deliveries(p_client_id), which returns only deliveries belonging to that client and includes operational context like the warehouse name and (if assigned) driver/route information. This means the application doesn't have to manually filter or build joins, and it also avoids exposing unrelated data.

For drivers, the equivalent perspective is handled by fn_get_driver_deliveries(p_driver_id). It returns the deliveries that are assigned to a given driver and enriches them with client and warehouse context so that a driver can see what matters operationally without needing to query separate tables.

Whenever the system needs a complete operational view (for staff/admin screens or detail pages), it relies on v_deliveries_full. This view is basically the "unified delivery dataset" of the project: it combines delivery fields with the most relevant labels and context from drivers, clients, routes and warehouses. The main advantage is consistency: the UI always renders deliveries from the same representation, rather than each endpoint doing its own joins.

Tracking reads follow the same idea: the goal is not to read a single status column, but to retrieve a full timeline. For that, tracking can be accessed through fn_get_delivery_tracking(p_tracking_number), which returns the delivery's events in chronological order and includes information such as the staff member (when available) and warehouse context for each event. In parallel, v_delivery_tracking exists as a reusable dataset-style view of the tracking timeline, which is very convenient for internal queries and admin pages.

For exports, deliveries are not taken from the UI view; instead, the system uses v_deliveries_export, which is designed to be flat and stable for JSON/CSV output. That keeps exports independent from how the UI decides to enrich or display deliveries.

Finally, because tracking is a high-read feature, the project also includes mv_delivery_tracking as a materialized view. The benefit here is performance and fast lookup by tracking number, especially when the tracking page is queried frequently. Since it's materialized, it needs explicit refreshes when we want it to reflect the newest tracking rows (that ties directly into the Update flow).

**Create (C)**

Creating a delivery in this project is not just "insert a row". A new shipment should be valid (at least regarding basic constraints), should have a consistent tracking number, and should become trackable immediately with an initial event.

The official entry point is sp_create_delivery(...), which centralizes the creation logic in the database. It validates important constraints early (for example, weight must be at least 1 if provided) and it checks that referenced entities like client/driver/route exist when their IDs are provided. It also applies sensible defaults: if no status is specified, the delivery starts as registered, and if priority is missing it falls back to normal.

One of the nicest details is tracking number generation: if the caller doesn't provide a tracking number, the procedure generates one automatically in the format PO-YYYYMMDD-XXXXX, which makes it both human-friendly and unique enough for the project's needs.

Two triggers reinforce the consistency of creation. trg_delivery_timestamp_check ensures timestamps are always valid and present (created/updated), and prevents impossible cases such as updated time being earlier than created time. Then trg_delivery_tracking_log automatically inserts the first tracking entry ("Delivery registered") right after insertion. That guarantees that every delivery has a timeline from the start without requiring the application to manually insert tracking rows.

**Update (U)**

Updates are separated into two categories because they mean different things in a logistics workflow.

**Updating delivery fields (not status)**

When the goal is to edit the delivery's mutable details (addresses, assignments, priority, delivery date, transition flag, etc.), the system uses sp_update_delivery(...). This procedure updates fields safely using a "keep old value if NULL" approach and enforces basic constraints such as weight being valid. It also fails clearly if the delivery doesn't exist, which is important to prevent silent inconsistencies.

As with creation, trg_delivery_timestamp_check runs automatically and updates updated_at consistently, making sure the record always reflects when it was last modified.

**Updating delivery status (workflow + audit trail)**

Status changes are treated as operational events, so they are handled through a dedicated procedure: sp_update_delivery_status(...). This procedure updates only the status and accepts optional context (staff id, warehouse id, notes) so that tracking events can carry meaningful operational information.

Before a status change is accepted, the database enforces the workflow through trg_delivery_status_workflow, which uses fn_is_valid_status_transition(old, new). This function contains the allowed lifecycle transitions (registered → ready → pending → in_transit → completed, with cancellation allowed along the way). If someone tries an illegal jump, the trigger raises an exception that explains the allowed transitions, preventing invalid lifecycle states from ever being stored.

Once a status update succeeds, trg_delivery_tracking_log automatically inserts the corresponding tracking row. Immediately after that, sp_update_delivery_status locates the most recent tracking record and enriches it with the staff/warehouse/notes that were provided. The result is both robust

and realistic: you always get a tracking event, and when the update comes from staff operations you also keep the operational context.

If the UI reads tracking data through the materialized view, then the system needs to refresh mv_delivery_tracking after status changes to ensure the new event appears instantly. That's why refresh logic exists alongside status updates in the application layer.

**Delete (D)**

Deletion is implemented as cancellation rather than physical removal, which is the correct choice for a delivery system: you want to preserve history for auditing, customer support, and reporting.

The deletion entry point is sp_delete_delivery(p_id), which issues a delete statement, but the actual behavior is controlled by trg_delivery_soft_delete. This trigger intercepts the delete and converts it into a soft delete by setting the delivery status to cancelled and updating the timestamp, while keeping the row in the table. The procedure then verifies that the delivery exists and ended up cancelled, so "delete" always produces a consistent business outcome.

**Bulk Create (Import)**

Although it is not part of classic CRUD, deliveries also support bulk creation through sp_import_deliveries(JSONB). This procedure loops through a JSON array and inserts deliveries one by one, applying the same principles as single creation: it generates tracking numbers when missing, applies default status/priority values, and allows triggers to fire per row so timestamps and initial tracking events are always created consistently.

## Invoice

Invoices are the financial representation of the service: they capture what is being charged, who is being charged (client), where it was processed (warehouse), and which staff member handled it. The model supports a header (invoice) plus detailed line items (invoice_item), and the total is kept consistent automatically whenever items change.

A key design decision here is that invoice totals are not recalculated in the application layer. Instead, the database takes responsibility for keeping the invoice consistent, so cost and quantities cannot become stale or contradictory.

**Read (R)**

Invoice reads are designed around "ready-to-display" datasets, because in real billing screens you rarely want the raw invoice row alone.

For most list and detail pages, the system relies on v_invoices_with_items, which returns invoices already enriched with warehouse name, staff name, client name, and a computed item count. The advantage is obvious: one query gives you everything the UI needs to show an invoice list that actually makes sense to a human.

When the purpose is exporting data (JSON/CSV), reads use v_invoices_export instead. It's intentionally flat, stable, and UI-agnostic, so exports don't break if the UI view changes.

For reporting or lightweight summaries, v_invoice_totals provides a compact view of each invoice's cost, quantity and item count, which is useful for dashboards, quick validations, or admin stats.

**Create (C)**

Invoice creation is split into two parts: creating the invoice "header" and then adding invoice items.

The header is created via sp_create_invoice. This keeps invoice creation consistent (defaults like status falling back to pending and cost starting from a safe baseline). It also means your app doesn't need to do manual insert statements for invoices — it calls the procedure and gets a clean invoice row created.

Items can be inserted either through normal inserts (e.g., import) or through sp_add_invoice_item, which is the safe "API" for adding a line item to an invoice. The crucial part is that item totals and invoice totals are not trusted to the caller: the database recalculates them.

Two triggers make this work smoothly:

- trg_invoice_item_calc_total ensures each item automatically computes total_item_cost as quantity * unit_price using fn_calculate_item_total. This avoids inconsistent totals if someone changes quantity/unit_price.

- trg_invoice_update_cost runs after inserts/updates/deletes of items and recalculates the parent invoice cost and quantity using fn_invoice_total (which itself relies on fn_invoice_subtotal and fn_calculate_tax).

**Update (U)**

Updating an invoice header (status, payment method, address, paid flag, etc.) is handled by sp_update_invoice. This keeps "administrative changes" in one place and prevents the UI from doing ad-hoc updates.

However, the most important updates in the invoice subsystem are actually item-driven: any time invoice_item rows change, the triggers ensure the invoice header is recalculated. That's what makes the billing layer robust: even if the application edits items, totals still remain correct.

So conceptually, invoice updates happen at two levels:

- header-level updates through sp_update_invoice,

- financial recalculations driven automatically by item triggers and the subtotal/tax functions.

**Delete (D)**

Invoices follow the same philosophy as Deliveries: deletion is not meant to erase history.

The system uses sp_delete_invoice, but the actual behavior is enforced by trg_invoice_soft_delete, which converts deletion into a business cancellation by setting status = 'cancelled' and preventing the row from being physically removed. This preserves billing history and avoids breaking references or audit trails.

**Bulk operations (Import)**

For initial datasets and testing, invoices can be bulk-imported using sp_import_invoices(JSONB). This procedure supports importing invoice headers and (optionally) nested items. Once items are inserted, the triggers still apply, which keeps totals consistent even in bulk operations.

## Dashboard

The dashboard is the "management window" of the system. Instead of computing all metrics in Django every time, the database exposes aggregated statistics so the UI can fetch them quickly and consistently.

**Read (R)**

The base dataset is v_dashboard_stats, which provides global counts such as total active vehicles, total deliveries, total clients, total active employees, active routes, pending deliveries, and total invoices.

On top of that, the system exposes fn_get_dashboard_stats(p_user_id, p_role), which adapts what is returned depending on the user's role:

- admins/managers get the full operational overview (all the stats from v_dashboard_stats),

- drivers see a "my_deliveries" count filtered to them,

- clients see "my_deliveries" filtered to them,

- staff/others get a simpler total deliveries metric.

This keeps the dashboard logic centralized: the app doesn't need to implement different queries per role; it just asks the DB for the right stats.

**Create/Update/Delete (C/U/D)**

Dashboard doesn't have classic CRUD because it's not an entity users "edit". It's a read-only layer built from existing operational data. Any "changes" happen indirectly when you create/update deliveries, routes, users, invoices, etc., and the dashboard reflects that automatically because it is defined as aggregated queries.

**Vehicle**

Vehicles represent the fleet resources used in transport operations. Vehicles are important because they connect planning (routes) with real-world constraints (capacity, maintenance, availability).

The vehicle subsystem strongly emphasizes validation at database level, especially regarding year correctness and safe deletion rules.

**Read (R)**

For day-to-day screens, reads rely on v_vehicles_full, which provides all vehicle fields needed for lists and detail displays.

For exports, v_vehicles_export provides a flat dataset designed for JSON/CSV output.

**Create (C)**

Vehicles are created using sp_create_vehicle. Before insertion, the procedure validates the vehicle year through fn_is_valid_year, ensuring you can't register nonsense like a vehicle from 1800 or 2099. This keeps fleet data realistic and prevents silent garbage from entering the system.

If the vehicle status is not provided, the procedure defaults it sensibly (e.g., available), so new vehicles enter the system in a usable state.

**Update (U)**

Vehicles are updated via sp_update_vehicle, which supports partial updates while still enforcing validation when needed (especially year validation using the same fn_is_valid_year). This makes updates consistent with creation rules.

**Delete (D)**

Vehicle deletion is intentionally restricted. sp_delete_vehicle prevents deleting a vehicle if it is currently assigned to an active route (not_started or on_going). That rule matters because deleting such a vehicle would leave operational routes in a broken state.

So deletion is allowed only when it does not compromise ongoing operations, which is exactly how a real logistics system would behave.

**Bulk operations (Import)**

Vehicles can be bulk-loaded with sp_import_vehicles(JSONB), and the same year validation is applied consistently. That avoids having "clean data in manual entries" but "dirty data in imports".

## Route

Routes represent planned and executed delivery runs: a driver + vehicle + warehouse context, plus timing, status, and operational notes. They are the planning layer that supports deliveries moving in an organized way.

**Read (R)**

Operational screens use v_routes_full, which enriches routes with driver identity (including license info), vehicle identity (plate and brand/model), and warehouse name. That makes it ideal for list pages and route management screens where you need human-readable context.

Exports use v_routes_export, which remains flat and stable for JSON/CSV.

**Create (C)**

Routes are created via sp_create_route. The procedure sets sensible defaults (such as not_started if no status is given) and inserts the scheduling fields.

The critical consistency rule here is enforced through trg_route_time_check. It guarantees that if both start and end times exist, the end must be later than the start. Without this trigger, you could store impossible routes that would break reports and planning logic.

**Update (U)**

Route updates go through sp_update_route, allowing changes to driver assignment, vehicle assignment, status, and scheduling details.

The same time consistency rule is always enforced again by trg_route_time_check on updates, so a route cannot "become invalid" after edits.

**Delete (D)**

Routes can only be deleted if they are not actively supporting deliveries. sp_delete_route blocks deletion when the route has deliveries that are not completed or cancelled. This prevents breaking the delivery workflow and preserves operational integrity.

**Bulk operations (Import)**

Routes can be imported in bulk through sp_import_routes(JSONB), which is useful for testing datasets and initial population. As with the rest of the system, the route trigger still applies, meaning time consistency remains enforced even on imported data.

# Conclusion

This project helped us understand, just how a real information system is constructed, not just theoretically but also in practice. As we worked out how to separate users, roles, and various other entities like employees and clients, we started to grasp why this separation was important.

Working with database views, stored procedures, and triggers was particularly useful, since it demonstrated the number of decisions that could actually be accommodated safely at the database level, thereby untangling the application significantly in the process. At the same time, using Django as the application layer provided us with a clear view of how the backend frameworks coordinate without them having all the business logic bundled in the process.