Pedro Montereiro 14366
David Gonzalez pv33971
Diego Alonso Laguillo pv33986
Rodrigo Rolo 18757

https://github.com/Diego-Alonso-05/Post_Office

Databases II – Project Report

Fast Office (Post Office & Logistics Management System)

1. Project Overview

Fast Office is a web-based system designed to manage the core operations of a post office or delivery company. The system centralizes operational data (deliveries, routes, vehicles, warehouses, invoices, users/employees) in PostgreSQL, while keeping MongoDB exclusively for notifications.

From a user perspective, the application behaves like a complete system: users can authenticate, navigate through modules, and perform CRUD actions according to their role.

2. Technologies and Architecture

2.1 Django + PostgreSQL (Core Data)

The main operational entities are modeled using Django ORM and stored in PostgreSQL. This ensures strong relational consistency through foreign keys and validation rules.

Implementation details:

The core entities of the system and their relationships are defined in the models.py file. Validation rules and user-friendly form behavior are implemented using Django forms in forms.py.
The routing of the different application modules and endpoints is configured in the urls.py file.

2.2 MongoDB (Notifications Only)

Notifications are stored in MongoDB to keep them lightweight and decoupled from transactional data. The app writes notifications without blocking the main workflow (fail silently if Mongo is unavailable). The connection to MongoDB and the helper functions responsible for creating, retrieving, and updating notification records are implemented in the notifications.py file.

2.3 Modular Web Structure

Instead of one huge views.py, the project splits views by domain module (deliveries, routes, vehicles, warehouses, invoices, users, etc.). URL mapping connects each section.

3. Worksheet 5 — Requirements, User Types, and Web Structure

3.1 Functional Requirements (What the system does)

We implemented the following core functionalities:

- Authentication (login/register/logout)
  - Endpoints exist for login/register/logout in the URL config.
  - File: urls.py (paths login/, register/, logout/)
    urls
- Role-based behavior
  - Users have a role field with choices (admin/client/driver/staff/manager).
  - File: models.py (User.ROLE_CHOICES, role field)
    models
- User/Employee hierarchy
  - Employee is linked 1–1 to User, and employees can be specialized as Driver or Staff via separate tables.
  - Automatic alignment of user.role with employee position is implemented in model logic.
  - File: models.py (Employee.clean() and Employee.save(), plus EmployeeDriver, EmployeeStaff)
    models

- CRUD for core entities
    - Warehouses, vehicles, routes, deliveries, invoices, users/clients are exposed via dedicated endpoints.
    - File: urls.py includes list/create/edit/delete routes for each module
      urls
- Import/Export
    - Dedicated endpoints exist for JSON import and JSON/CSV export for multiple entities.
    - File: urls.py shows /import/json/, /export/json/, /export/csv/ routes for warehouses/vehicles/routes/deliveries/invoices
      urls
    - Import forms exist for JSON upload.
    - File: forms.py (VehicleImportForm, WarehouseImportForm, DeliveryImportForm, RouteImportForm)
      forms
- Notifications
    - Notifications can be created, fetched, and marked as read.
    - Files: Mongo helper notifications.py
      notifications and notification endpoints in urls.py (notifications/, notifications/read/<id>/)
      urls

3.2 Non-Functional Requirements (How it behaves)

We paid special attention to integrity and maintainability:

- Data integrity & validation
    - Models include validators, ordering, PROTECT FKs to avoid accidental deletion.
    - File: models.py (validators on year/weight/capacity; on_delete=models.PROTECT used in relations)
      models
- Form-level business rules
    - We enforce constraints such as:
        - wage must be positive,
        - license expiry must be in the future,
        - warehouse open/close times must be consistent,
        - route end time must be after start time,
        - delivery weight positive and updated_at after registered_at, etc.
    - File: forms.py (clean_* and clean() methods across forms)
      forms
- Separation of concerns
    - Models, forms, and URLs are separated, and domain logic is grouped by module.

3.3 User Types

The system supports these user types:

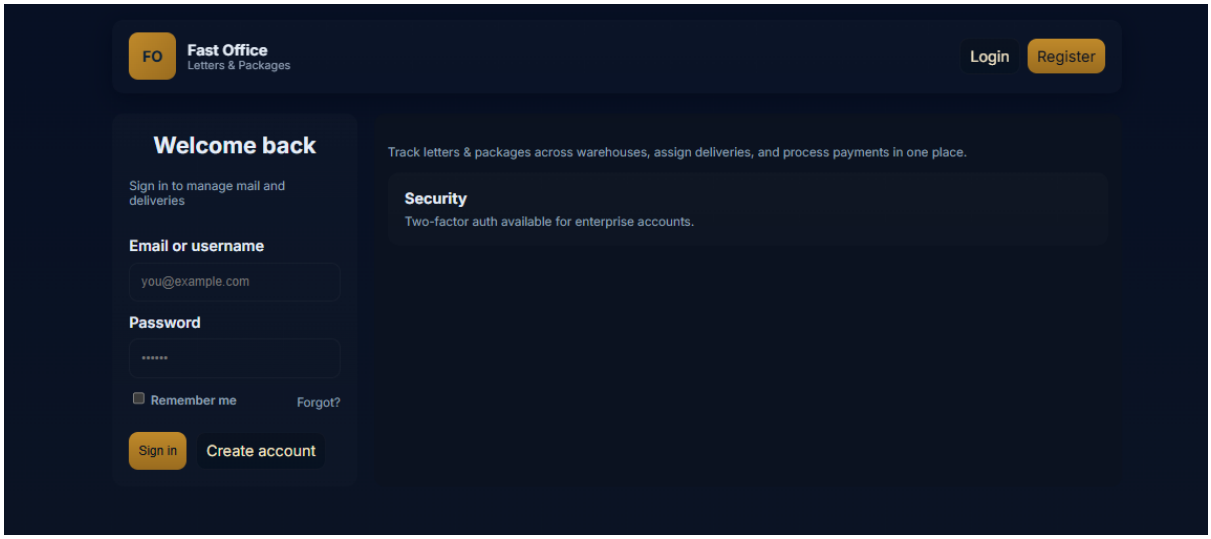
- Admin: full control of the system
- Manager: supervisory access
- Staff: operational support
- Driver: route/delivery operational role
- Client: track personal deliveries / invoices

Roles are stored in the user table and used across the app.

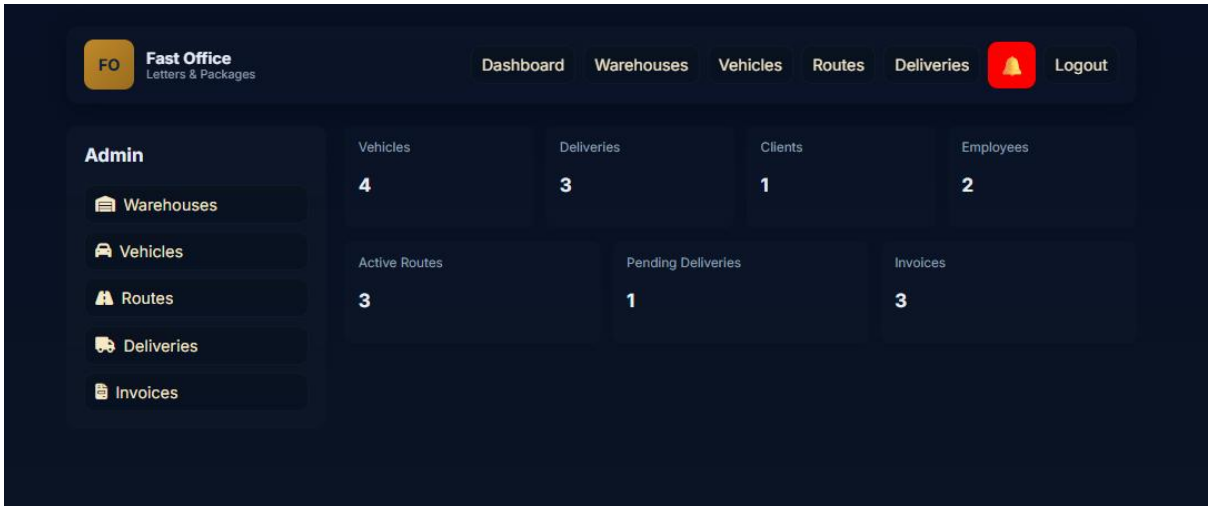
3.4 Web Structure (Main Pages)

Based on routing, the app is organized into modules:

- Dashboard: /
- Home: /home/
- Authentication: /login/, /register/, /logout/
- Profile: /profile/
- Warehouses: /warehouses/… (CRUD + import/export)
- Vehicles: /vehicles/… (CRUD + import/export)
- Routes: /routes/… (CRUD + import/export)
- Deliveries: /deliveries/… (CRUD + detail + import/export)
- Invoices: /invoices/… (CRUD + import/export)
- Users/Clients admin pages: /users/…, /clients/…
- Notifications: /notifications/…



The application provides an authentication entry point that allows users to securely access the system according to their role.



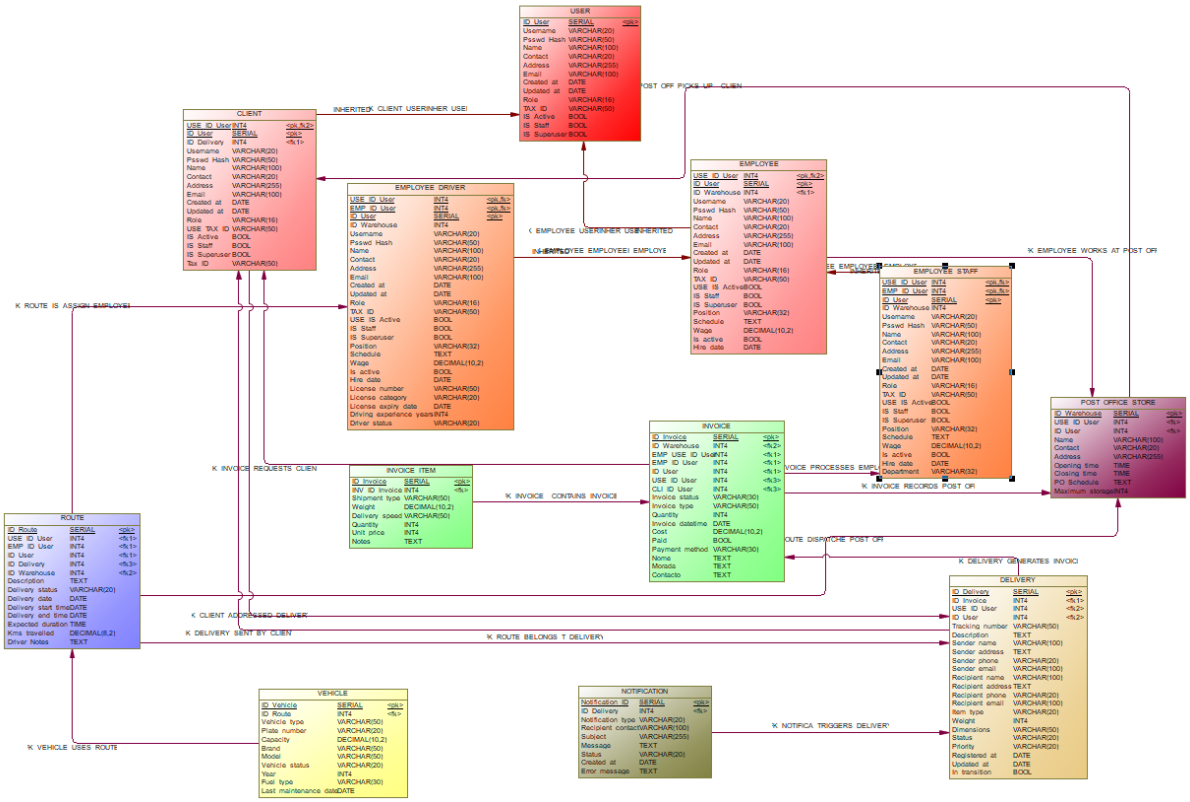
Admin dashboard displaying role-based navigation and summary information for the main system modules.

4. Worksheet 7 — Data Models and Database Objects

4.1 Conceptual Model (High level)

At a conceptual level, the system is centered around Deliveries and their operational context:

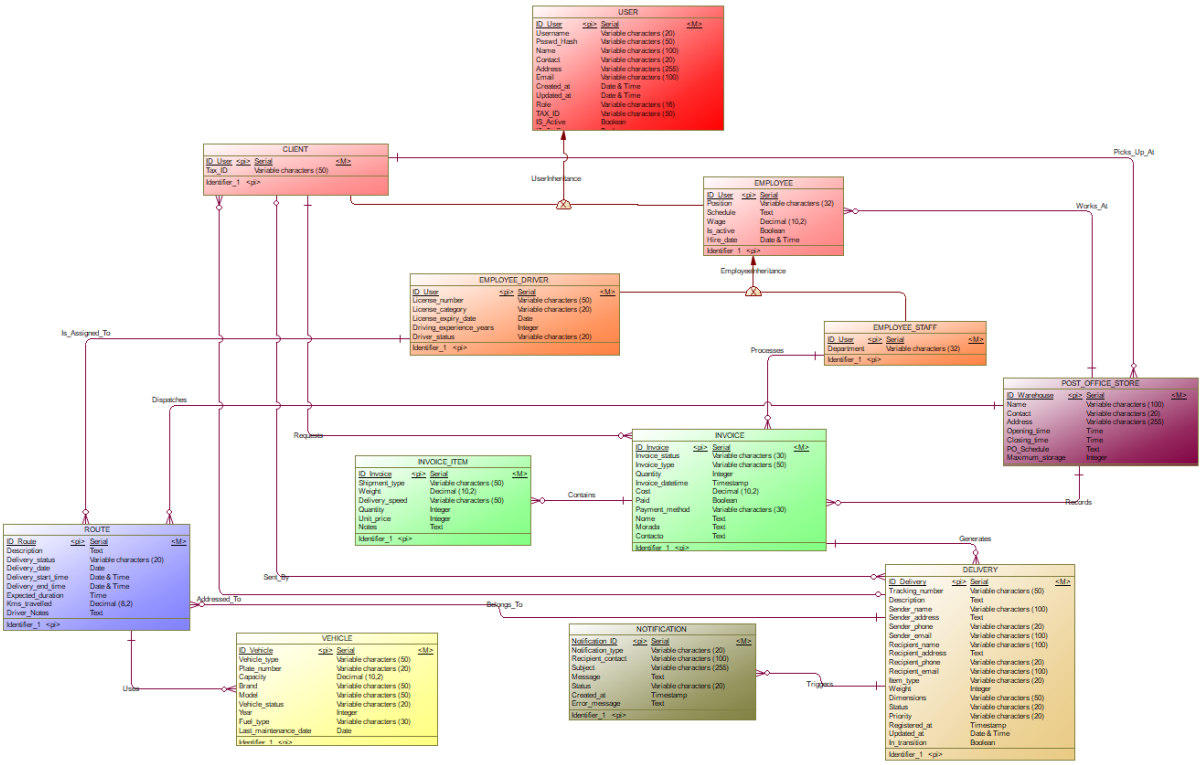
PDM



- Users (clients) create/own deliveries and invoices.
- Employees (drivers/staff) operate deliveries and routes.
- Routes organize deliveries and link drivers + vehicles.
- Warehouses support storage/operations.

4.2 Logical Model (Relationships)

CDM



Main relationship patterns:

- User ↔ Employee: 1–1 (Employee.user)
- Employee ↔ Driver/Staff info: 1–1 (specialization)
- Invoice → Deliveries: 1–N (delivery linked to invoice)
- Route → Deliveries: 1–N (delivery linked to a route)
- Employee → Routes/Deliveries: 1–N (driver assigned)
- Vehicle → Routes: 1–N

4.3 Physical Model (PostgreSQL via ORM)

The physical model is implemented through Django models, including constraints and validation rules.

Examples of integrity rules:

- tracking_number is unique (Delivery)
- plate_number is unique (Vehicle)
- Route has a uniqueness constraint for (driver, vehicle, delivery_date) to avoid double-booking

4.4 Database Objects Implemented

Our DB-related objects include:

- Tables (via ORM)
- Views/abstractions (Worksheet 12)
- Export functions (Worksheet 13)
- Triggers (Worksheet 13)

5. Worksheet 9 — Search Attributes and Reports

5.1 Search Attributes (By Entity)

We identified the following attributes as the main “search keys”:

- User: username, email, full_name, role
- Employee: user, position, is_active
- Warehouse: name, address, contact
- Vehicle: plate_number, vehicle_type, vehicle_status
- Route: delivery_date, driver, vehicle, delivery_status
- Delivery: tracking_number, status, priority, delivery_date, client, driver
- Invoice: id_invoice, invoice_status, invoice_type, user

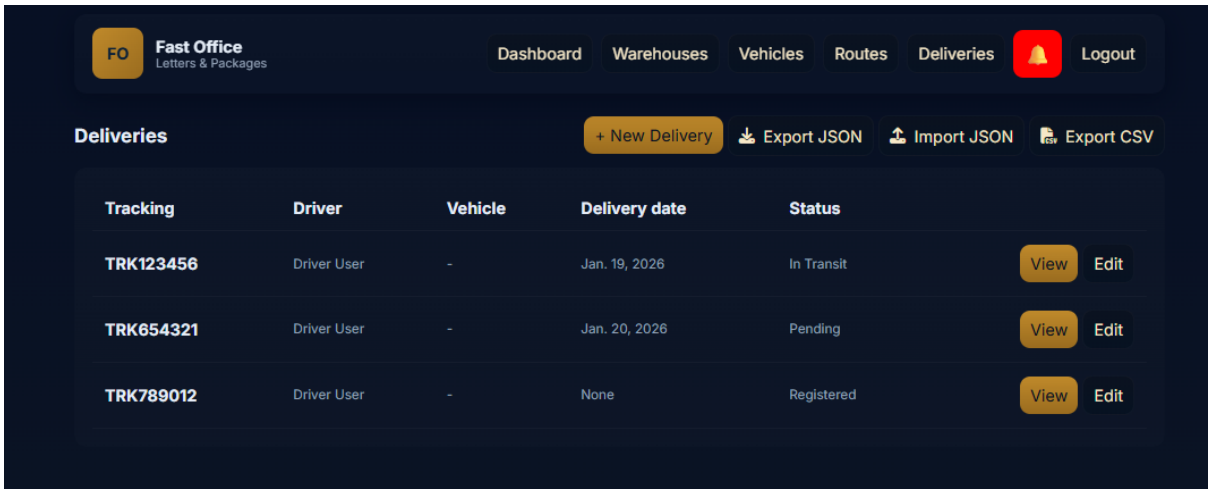
These fields exist directly in the models and are displayed/used in list and detail pages.

5.2 Reports the Application Can Generate

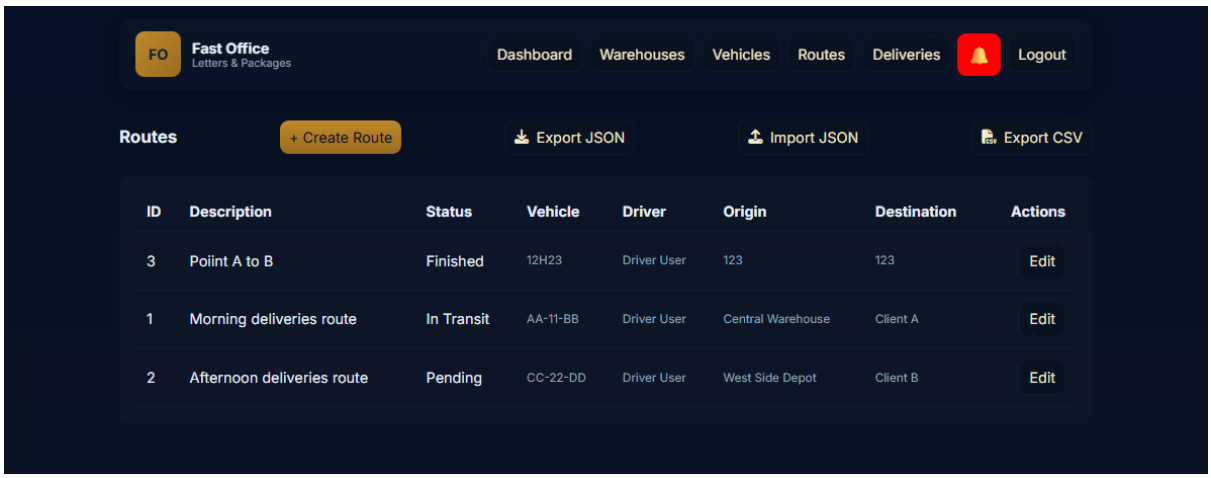
We treat “reports” as structured lists/exports that support decisions and monitoring:

- Delivery report: deliveries by status, priority, date; also track by tracking number
- Routes report: routes per day, driver assignment, vehicle usage, kms traveled
- Vehicles report: vehicle fleet status and availability
- Warehouses report: warehouse capacity and operational info
- Invoices report: invoice status (paid/unpaid), type, costs and billing activity
- Users report: list users by role and activity

The existence of JSON/CSV export endpoints confirms reporting-oriented output capabilities.



The application provides structured list views that function as operational reports for monitoring system activity.



Routes list page showing route status, assigned vehicles and drivers, used as an operational report.

Generated invoice report summarizing shipment details, pricing, taxes, and total cost.

6. Worksheet 10 — Django Prototype, DB Users, and JSON Exchange

6.1 Django Prototype

We implemented a navigable prototype where the user can move through modules as if the system were fully complete.

- File pointers (structure)
    - URL routing: urls.py
      urls
    - Domain forms: forms.py
      forms
    - Domain models: models.py
      models

6.2 Database Users (SGBD users)

In our context, the application has multiple “user types” at the application level, supported by a role stored in PostgreSQL. In practice, different permissions are enforced through application logic and role checks.

(If you created actual PostgreSQL roles/users externally, you can mention them here; otherwise, it’s valid to describe the application roles, because the prototype’s access control is implemented that way.)

6.3 JSON Import/Export Scenarios

We support JSON exchange mainly for interoperability and for easy population / backup:

- Export (JSON): data can be exported from main modules through specific endpoints.
- Import (JSON): authorized users upload a JSON file and the system inserts valid records.
- Where it is implemented

- Import form definitions: forms.py
  forms
- Import/export endpoints per entity: urls.py
  urls

We also added PDF Export for invoices to get receipts in a professional manner.

FAST OFFICE

Rua Afonso Ribeiro

Lisboa, Portugal

contact@fastoffice.com

Client User
Client Street
912345679

Shipment Type	Weight	Delivery Speed	Quantity	Unit Price (€)	Total (€)
Package	7.77	Express	1	7.77	7.77
Letter	9.99	Standard	1	9.99	9.99
Package	5.55	Overnight	1	5.55	5.55
Letter	4.44	Express	1	4.44	4.44
				Subtotal (€)	27.75
				Tax 23% (€)	6.3825
				Total (€)	34.1325

7. Worksheet 11 — CRUD Testing with Pytest

To validate our CRUD processes, we implemented automated tests using **pytest**, focusing on key collections and operations. The test file demonstrates inserting, reading, updating, and deleting records and asserting expected results.

- Where it is implemented
  - test_mongo.py contains CRUD tests for deliveries, notifications, postoffice, routes, users, and vehicles (Mongo collections).
    test_mongo

8. Worksheet 12 — Database Views (Simple and Materialized)

For reporting and structured retrieval, we identified queries that combine multiple entities (e.g., deliveries with invoice/user information, routes with driver/vehicle assignments). The worksheet requires implementing simple and materialized views in PostgreSQL based on previously identified queries.

In our project, these database-level query abstractions are part of the “reporting and export” approach: rather than duplicating complex joins repeatedly in application code, we centralize reusable query logic at the database layer whenever it makes sense for reporting.

9. Worksheet 13 — Database Export Functions and Triggers

9.1 Export Functions (CSV)

We implemented PostgreSQL functions to export key entity data to CSV format:

- export_warehouses_csv()
- export_vehicles_csv()
- export_routes_csv()
- export_deliveries_csv()
- export_invoices_csv()

9.2 Triggers

We implemented triggers to enforce integrity and automate actions such as timestamps, validations, and status synchronization. The project notes the following trigger functions:

- fn_update_delivery_timestamp()
- fn_log_delivery_created()
- fn_log_delivery_status_change()
- fn_validate_delivery()
- fn_validate_invoice()
- fn_validate_driver()
- fn_route_status()

10. Practical Notes (How to Run / Test Users)

The repository includes a run guide and also test credentials for different roles (admin/client/driver/staff/manager), which makes it easier to demonstrate role-based behavior during evaluation.

Procedures

Vehicle & Driver Scheduling Check Procedure

We want to ensure that a driver or vehicle is not assigned to overlapping routes on the same day.

```
-- ========================================
-- PROCEDURE: check_driver_vehicle_availability
-- ========================================
CREATE OR REPLACE FUNCTION check_driver_vehicle_availability(
    p_driver_id INT,
    p_vehicle_id INT,
    p_delivery_date DATE
) RETURNS BOOLEAN AS $$
DECLARE
    conflicting_routes INT;
BEGIN
    SELECT COUNT(*) INTO conflicting_routes
    FROM "PostOffice_App_route"
    WHERE delivery_date = p_delivery_date
      AND (driver_id = p_driver_id OR vehicle_id = p_vehicle_id);

    IF conflicting_routes > 0 THEN
        RAISE EXCEPTION 'Driver or Vehicle is already assigned on
%', p_delivery_date;
        RETURN FALSE;
    ELSE
        RETURN TRUE;
    END IF;
END;
$$ LANGUAGE plpgsql;
```

We want to prevent invalid status changes. For example, you shouldn't move a delivery from `Registered` directly to `Completed`.

```
-- ========================================
-- PROCEDURE: validate_delivery_status_transition
-- ========================================
CREATE OR REPLACE FUNCTION validate_delivery_status_transition(
    p_delivery_id INT,
    p_new_status VARCHAR
) RETURNS BOOLEAN AS $$
DECLARE
    current_status VARCHAR;
BEGIN
    SELECT status INTO current_status
    FROM "PostOffice_App_delivery"
    WHERE id = p_delivery_id;

    IF current_status IS NULL THEN
        RAISE EXCEPTION 'Delivery not found';
    END IF;

    CASE current_status
        WHEN 'Registered' THEN
            IF p_new_status NOT IN ('Ready', 'Cancelled') THEN
                RAISE EXCEPTION 'Invalid transition from Registered
to %', p_new_status;
            END IF;
        WHEN 'Ready' THEN
            IF p_new_status NOT IN ('In Transit', 'Cancelled') THEN
                RAISE EXCEPTION 'Invalid transition from Ready to
%', p_new_status;
            END IF;
        WHEN 'In Transit' THEN
            IF p_new_status NOT IN ('Completed', 'Cancelled') THEN
                RAISE EXCEPTION 'Invalid transition from In Transit
to %', p_new_status;
            END IF;
        WHEN 'Completed' THEN
            RAISE EXCEPTION 'Completed deliveries cannot change
status';
        WHEN 'Cancelled' THEN
            RAISE EXCEPTION 'Cancelled deliveries cannot change
status';
    END CASE;
```

```
    RETURN TRUE;
END;
$$ LANGUAGE plpgsql;



-------------------------- functions.sql -------------------------
-
-----------------------------
-- Export .csv for WAREHOUSES
-----------------------------

CREATE OR REPLACE FUNCTION public.export_warehouses_csv(
      )
    RETURNS TABLE(line text)
    LANGUAGE 'plpgsql'
    COST 100
    VOLATILE PARALLEL UNSAFE
    ROWS 1000
AS $BODY$
        BEGIN
            RETURN QUERY
            SELECT CONCAT_WS(',',
                id,
                name,
                address,
                contact,
                po_schedule_open,
                po_schedule_close,
                maximum_storage_capacity
            )
            FROM "PostOffice_App_warehouse"
            ORDER BY id;
        END;

$BODY$;

-----------------------------
-- Export .csv for VEHICLES
-----------------------------
```

```
CREATE OR REPLACE FUNCTION public.export_vehicles_csv(
      )
    RETURNS TABLE(line text)
    LANGUAGE 'plpgsql'
    COST 100
    VOLATILE PARALLEL UNSAFE
    ROWS 1000
AS $BODY$
    BEGIN
        RETURN QUERY
        SELECT CONCAT_WS(',',
            id,
            plate_number,
            brand,
            model,
            capacity,
            vehicle_status,
            year,
            fuel_type,
            last_maintenance_date,
            vehicle_type
        )
        FROM "PostOffice_App_vehicle"
        ORDER BY id;
    END;
$BODY$;
.
--------------------------
-- Export .csv for ROUTES
--------------------------
CREATE OR REPLACE FUNCTION public.export_routes_csv(
      )
    RETURNS TABLE(line text)
    LANGUAGE 'plpgsql'
    COST 100
    VOLATILE PARALLEL UNSAFE
    ROWS 1000
AS $BODY$
    BEGIN
        RETURN QUERY
        SELECT CONCAT_WS(',',
```

```
            id,
            description,
            delivery_status,
            vehicle_id,
            driver_id,
            origin_name,
            origin_address,
            origin_contact,
            destination_name,
            destination_address,
            destination_contact,
            delivery_date,
            delivery_start_time,
            delivery_end_time,
            kms_travelled,
            expected_duration,
            driver_notes
        )
        FROM "PostOffice_App_route"
        ORDER BY id;
    END;
$BODY$;

-----------------------------
-- Export .csv for DELIVERIES
-----------------------------
CREATE OR REPLACE FUNCTION public.export_deliveries_csv(
      )
    RETURNS TABLE(line text)
    LANGUAGE 'plpgsql'
    COST 100
    VOLATILE PARALLEL UNSAFE
    ROWS 1000
AS $BODY$
    BEGIN
        RETURN QUERY
        SELECT CONCAT_WS(',',
            id,
            tracking_number,
            description,
            sender_name,
```

```
            sender_address,
            sender_phone,
            sender_email,
            recipient_name,
            recipient_address,
            recipient_phone,
            recipient_email,
            item_type,
            weight,
            dimensions,
            status,
            priority,
            registered_at,
            updated_at,
            in_transition,
            destination,
            delivery_date,
            driver_id,
            invoice_id,
            route_id,
            client_id
        )
        FROM "PostOffice_App_delivery"
        ORDER BY id;
    END;
$BODY$;

-----------------------------
-- Export .csv for INVOICES
-----------------------------
CREATE OR REPLACE FUNCTION public.export_invoices_csv(
      )
    RETURNS TABLE(line text)
    LANGUAGE 'plpgsql'
    COST 100
    VOLATILE PARALLEL UNSAFE
    ROWS 1000

AS $BODY$
BEGIN
    RETURN QUERY
```

```
    SELECT
        COALESCE(id_invoice::TEXT, '') || ',' ||
        COALESCE(invoice_status, '') || ',' ||
        COALESCE(invoice_type, '') || ',' ||
        COALESCE(quantity::TEXT, '') || ',' ||
        COALESCE(TO_CHAR(invoice_datetime, 'YYYY-MM-DD HH24:MI:SS'),
'') || ',' ||
        COALESCE(cost::TEXT, '') || ',' ||
        CASE WHEN paid THEN 'true' ELSE 'false' END || ',' ||
        COALESCE(payment_method, '') || ',' ||
        COALESCE(REPLACE(name, ',', ';'), '') || ',' ||
        COALESCE(REPLACE(address, ',', ';'), '') || ',' ||
        COALESCE(contact, '') || ',' ||
        COALESCE(user_id::TEXT, '')
    FROM "PostOffice_App_invoice"
    ORDER BY
        invoice_datetime DESC NULLS LAST,
        id_invoice;
END;
$BODY$;
```

Material Views

Number of INVOICES and total revenue grouped by day

```
CREATE MATERIALIZED VIEW mv_daily_sales AS
    SELECT
        DATE(invoice_datetime) AS day,
        COUNT(*) AS total_invoices,
        SUM(cost) AS total_revenue
    FROM public."PostOffice_App_invoice"
    GROUP BY day
    ORDER BY day;
```

All-time number of INVOICES and total revenue grouped by payment method

```
CREATE MATERIALIZED VIEW mv_payment_methods_stats AS
    SELECT
        payment_method,
        COUNT(*) AS total_invoices,
```

```
        SUM(cost) AS total_value
    FROM public."PostOffice_App_invoice"
    GROUP BY payment_method;
```

Top 10 customers by spending (all-time)

```
CREATE MATERIALIZED VIEW mv_top_customers AS
    SELECT
        name AS customer_name,
        COUNT(*) AS total_invoices,
        SUM(cost) AS total_spent
    FROM public."PostOffice_App_invoice"
    GROUP BY name
    ORDER BY total_spent DESC
    LIMIT 10;
```

Number of INVOICES and total revenue grouped by month

```
CREATE MATERIALIZED VIEW mv_monthly_sales AS
    SELECT
        DATE_TRUNC('month', invoice_datetime) AS month,
        COUNT(*) AS total_invoices,
        SUM(cost) AS total_revenue
    FROM public."PostOffice_App_invoice"
    GROUP BY month
    ORDER BY month;
```

Simple views

INVOICES listing (all-time

```
CREATE OR REPLACE VIEW vw_all_invoices AS
    SELECT
        id_invoice,
        name AS customer_name,
        invoice_type,
        invoice_status,
        quantity,
        cost,
        paid,
        payment_method,
        invoice_datetime
    FROM public."PostOffice_App_invoice";
```

All unpaid INVOICES (all-time)

```
CREATE OR REPLACE VIEW vw_unpaid_invoices AS
    SELECT *
    FROM public."PostOffice_App_invoice"
    WHERE paid = FALSE;
```

All-time number of INVOICES and total revenue grouped by invoice status (Pending, Completed, Cancelled or Refunded)

```
CREATE OR REPLACE VIEW vw_invoices_totals_by_status AS
    SELECT
        invoice_status,
        COUNT(*) AS total_invoices,
        SUM(cost) AS total_value
    FROM public."PostOffice_App_invoice"
    GROUP BY invoice_status
    ORDER BY total_invoices DESC;
```

INVOICES from the last 7 days (rolling weekly window)

```
CREATE OR REPLACE VIEW vw_recent_invoices AS
SELECT *
FROM public."PostOffice_App_invoice"
WHERE invoice_datetime >= NOW() - INTERVAL '7 days';
```

Triggers

Update timestamp for DELIVERY

```
CREATE OR REPLACE FUNCTION fn_update_delivery_timestamp()
RETURNS TRIGGER AS $$
BEGIN
    NEW.updated_at = CURRENT_TIMESTAMP;
    RETURN NEW;
END;
$$ LANGUAGE plpgsql;

CREATE TRIGGER trg_update_delivery_timestamp
BEFORE UPDATE ON "PostOffice_App_delivery"
FOR EACH ROW
EXECUTE FUNCTION fn_update_delivery_timestamp();
```

Shows in pgadmin4 logs that a DELIVERY was created

```
CREATE OR REPLACE FUNCTION fn_log_delivery_created()
RETURNS TRIGGER AS $$
BEGIN
    RAISE NOTICE 'Delivery % created with status %', NEW.id,
NEW.status;
    RETURN NEW;
END;
$$ LANGUAGE plpgsql;

CREATE TRIGGER trg_delivery_created
AFTER INSERT ON "PostOffice_App_delivery"
FOR EACH ROW
EXECUTE FUNCTION fn_log_delivery_created();
```

Shows in pgadmin4 logs that a DELIVERY status has changed

```
CREATE OR REPLACE FUNCTION fn_log_delivery_status_change()
RETURNS TRIGGER AS $$
BEGIN
    IF NEW.status IS DISTINCT FROM OLD.status THEN
        RAISE NOTICE 'Delivery % changed status from % to %',
NEW.id,
        OLD.status, NEW.status;
    END IF;
RETURN NEW;
END;
$$ LANGUAGE plpgsql;

CREATE TRIGGER trg_status_change
AFTER UPDATE ON "PostOffice_App_delivery"
FOR EACH ROW
EXECUTE FUNCTION fn_log_delivery_status_change();
```

Shows in pgadmin4 error in case of DELIVERY fields are not being updated properly

```
CREATE OR REPLACE FUNCTION fn_validate_delivery()
RETURNS TRIGGER AS $$
BEGIN
    IF NEW.status = 'Completed' AND NEW.delivery_date IS NULL THEN
        RAISE EXCEPTION 'Cannot mark delivery as Completed without
        delivery_date';
    END IF;
```

```
    IF NEW.weight <= 0 THEN
        RAISE EXCEPTION 'Weight must be greater than 0';
    END IF;

    RETURN NEW;
END;
$$ LANGUAGE plpgsql;

CREATE TRIGGER trg_validate_delivery
BEFORE INSERT OR UPDATE ON "PostOffice_App_delivery"
FOR EACH ROW
EXECUTE FUNCTION fn_validate_delivery();
```

Shows in pgadmin4 error in case of INVOICE fields are not being updated properly

```
CREATE OR REPLACE FUNCTION fn_validate_invoice()
RETURNS TRIGGER AS $$
BEGIN
    IF NEW.cost < 0 THEN
        RAISE EXCEPTION 'Invoice cost cannot be negative';
    END IF;

    IF NEW.quantity IS NOT NULL AND NEW.quantity <= 0 THEN
        RAISE EXCEPTION 'Invoice quantity must be greater than
zero';
    END IF;

    IF NEW.paid = TRUE THEN
        NEW.invoice_status = 'Paid';
    END IF;

    RETURN NEW;
END;
$$ LANGUAGE plpgsql;

CREATE TRIGGER trg_validate_invoice
BEFORE INSERT OR UPDATE ON "PostOffice_App_invoice"
FOR EACH ROW
EXECUTE FUNCTION fn_validate_invoice();
```

Shows in pgadmin4 error in case of driver license expires or is insert incorrectly

```
CREATE OR REPLACE FUNCTION fn_validate_driver()
RETURNS TRIGGER AS $$
BEGIN
    IF NEW.license_expiry_date < CURRENT_DATE THEN
        RAISE EXCEPTION 'Driver license expired';
    END IF;

    IF NEW.driving_experience_years < 0 THEN
        RAISE EXCEPTION 'Driving experience cannot be negative';
    END IF;
RETURN NEW;
END;
$$ LANGUAGE plpgsql;

CREATE TRIGGER trg_validate_driver
BEFORE INSERT OR UPDATE ON "PostOffice_App_employeedriver"
FOR EACH ROW
EXECUTE FUNCTION fn_validate_driver();
```

Updates all DELIVERies of a route as completed when the route is completed

```
CREATE OR REPLACE FUNCTION fn_route_status()
RETURNS TRIGGER AS $$
BEGIN
    IF NEW.delivery_status = 'Completed' THEN
        UPDATE "PostOffice_App_delivery"
        SET status='Completed', delivery_date = CURRENT_DATE
        WHERE route_id = NEW.id;
    END IF;
RETURN NEW;
END;
$$ LANGUAGE plpgsql;

CREATE TRIGGER trg_route_completed
AFTER UPDATE ON "PostOffice_App_route"
FOR EACH ROW
EXECUTE FUNCTION fn_route_status();
```

Updates INVOICES total cost adding the item costs

```
CREATE OR REPLACE FUNCTION update_invoice_cost()
RETURNS TRIGGER AS $$
BEGIN
    UPDATE "PostOffice_App_invoice"
    SET cost = (
        SELECT COALESCE(SUM(quantity * unit_price), 0)
        FROM postoffice_app_invoice_items
        WHERE invoice_id = COALESCE(NEW.invoice_id, OLD.invoice_id)
    )
    WHERE id_invoice = COALESCE(NEW.invoice_id, OLD.invoice_id);
    RETURN NEW;
END;
$$ LANGUAGE plpgsql;

DROP TRIGGER IF EXISTS trigger_invoice_cost ON
postoffice_app_invoice_items;

CREATE TRIGGER trigger_invoice_cost
AFTER INSERT OR UPDATE OR DELETE ON postoffice_app_invoice_items
FOR EACH ROW EXECUTE FUNCTION update_invoice_cost();
```

11. Conclusion

Fast Office is a coherent prototype that integrates:

- structured relational data modeling in PostgreSQL (via Django ORM),
- modular web organization with clear CRUD flows,
- role-based access logic at the application level,
- JSON/CSV data exchange for portability and reporting,
- MongoDB usage for notifications,
- automated CRUD testing using pytest,
- database-level functions and triggers to support integrity and exports.

Overall, the system fulfills the objectives of the practical worksheets and demonstrates an end-to-end approach to database-backed web application development.