SYSTEM DOCUMENTATION - Farma Rapid

Daniel Royero

21/11/2025

#SYSTEM DOCUMENTATION

1. Project Information

- **Project Name:** Farma Rapid – Pharmacy Management System
- **Student Name:** Daniel Royero
- **Course:** Database II – Universidad de La Guajira
- **Semester:** 2025–6
- **Date:** *21/11/2025*
- **Instructor:** Jaider Quintero

Short Project Description

Farma Rapid is a small pharmacy management system that allows the user to register and consult medicines, customers, prescriptions and sales.
The system is divided into a REST API backend and an Angular frontend.
The interface focuses on a clean layout with a left navigation menu and four main options: **Catalog**, **Customers**, **Prescriptions** and **Sales**.

---

2. System Architecture Overview

2.1 Architecture Description

The solution follows a classic **client–server** architecture:

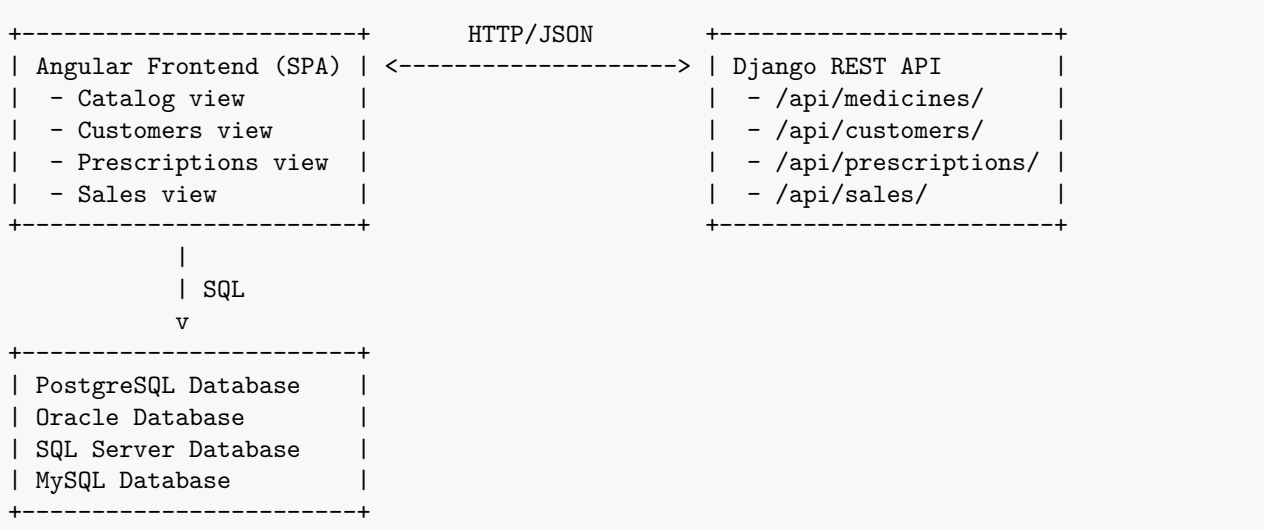
- The **frontend** is a Single Page Application (SPA) built with Angular.
  It is responsible for presenting the information, managing navigation between views and sending HTTP requests to the API.
- The **backend** exposes a **REST API** that implements the business logic and communicates with the relational database.
- The **database** stores the persistent information: medicines, customers, prescriptions, prescription details and sales.

All communication between frontend and backend is done over HTTP using JSON messages.

2.2 Technologies Used

- **Frontend**
  - Angular 17
  - TypeScript
  - HTML5 / CSS3
  - VS Code + Angular CLI
- **Backend**
  - Python 3.x
  - Django + Django REST Framework (DRF)
  - Virtual environment (`venv`)
  - VS Code
- **Database Engine**
  - PostgreSQL 14+ (relational database)
- **Additional Libraries / Tools**
  - Git & GitHub for version control
  - REST Client (VS Code extension) for testing the API (`.http` files)
  - pgAdmin / DBeaver for database administration

2.3 Visual Explanation of the System's Operation



3. Database Documentation (ENGLISH)

3.1 Database Description

The database is **relational** and **normalized**. Its goal is to store the main entities involved in a small pharmacy:

- **medicines**: products sold by the pharmacy.

- **customers**: people to whom medicines are sold.
- **prescriptions**: medical prescriptions associated with one customer.
- **prescription_items**: medicines and dosage that belong to each prescription.
- **sales**: invoices or sales records.
- **sale_items**: medicines and quantity sold in each sale.

Each entity has an integer primary key and the necessary foreign keys to keep referential integrity.

3.2 ERD – Entity Relationship Diagram

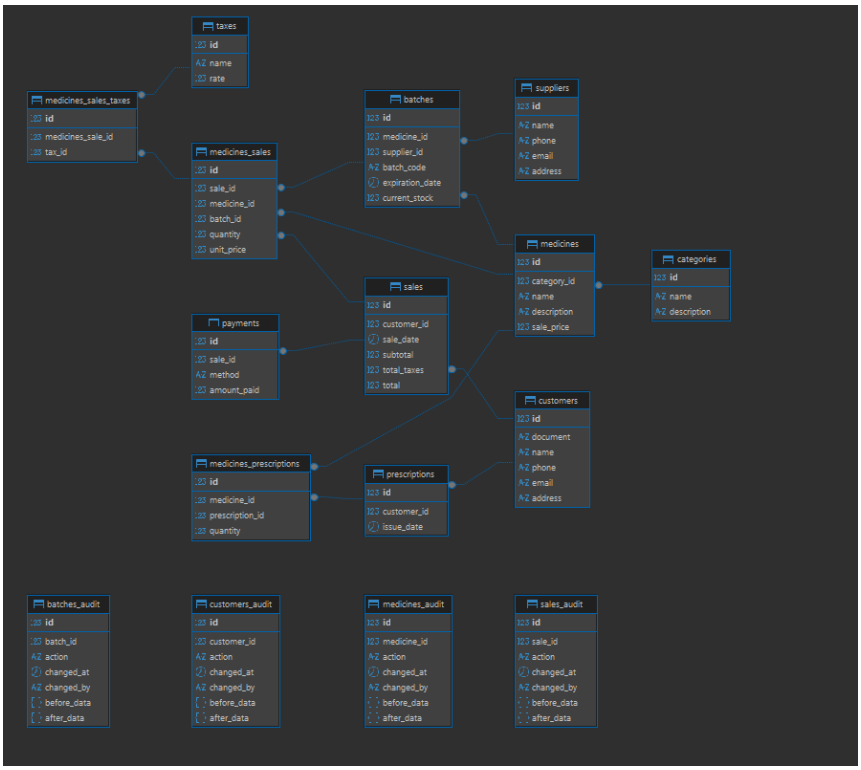


Figure 3.1 – Entity–Relationship Diagram of the Farma Rapid database.

3.3 Logical Model (Summary)

Main relationships between entities:

- **Customer (1) — (N) Prescription**
- **Prescription (1) — (N) PrescriptionItem — (N) Medicine**
- **Customer (1) — (N) Sale**
- **Sale (1) — (N) SaleItem — (N) Medicine**

These relationships ensure that each prescription and each sale is associated with exactly one customer, and that each item belongs to a single prescription or sale.

3.4 Physical Model (Tables)

Below is a simplified version of the tables.
Adapt field names and types to match your real implementation.

Table 3.1 – medicines

Column	Type	PK/FK	Description
id	serial	PK	Internal identifier
name	varchar(100)		Commercial name of the medicine
description	varchar(255)		Short description / use
sale_price	numeric(10,2)		Unit sale price
created_at	timestamp		Creation date
updated_at	timestamp		Last update

Table 3.2 – customers

Column	Type	PK/FK	Description
id	serial	PK	Internal identifier
document	varchar(20)		Identification number
full_name	varchar(150)		Customer full name
phone	varchar(30)		Phone number
email	varchar(100)		Email address
address	varchar(200)		Home address

Table 3.3 – prescriptions

Column	Type	PK/FK	Description
id	serial	PK	Internal identifier
customer_id	integer	FK	References `customers(id)`
doctor_name	varchar(150)		Name of the doctor
created_at	date		Date of the prescription

Table 3.4 – prescription_items

Column	Type	PK/FK	Description
id	serial	PK	Internal identifier
prescription_id	integer	FK	References `prescriptions(id)`
medicine_id	integer	FK	References `medicines(id)`
dosage	varchar(100)		Dosage / instructions
quantity	integer		Quantity recommended

Table 3.5 – sales

Column	Type	PK/FK	Description
id	serial	PK	Sale identifier
customer_id	integer	FK	References `customers(id)`
sale_date	timestamp		Date and time of the sale
subtotal	numeric(10,2)		Sum of line items (before taxes)
total_taxes	numeric(10,2)		Total taxes
total_amount	numeric(10,2)		Final total to pay

Table 3.6 – sale_items

Column	Type	PK/FK	Description
id	serial	PK	Line identifier
sale_id	integer	FK	References `sales(id)`
medicine_id	integer	FK	References `medicines(id)`
quantity	integer		Quantity sold
unit_price	numeric(10,2)		Price per unit at the moment of the sale

4. Use Cases – CRUD

Below, a generic CRUD use case for the **Medicine** entity.
You can copy and adapt the same structure for **Customers**, **Prescriptions** and **Sales**.

4.1 Use Case: Create Medicine

Actor: Pharmacy employee (system user)

Description: The actor registers a new medicine in the catalog.

Preconditions

- The user has access to the system.
- Mandatory fields (name, price) are known.

Postconditions

- A new medicine record is stored in the database.
- The medicine appears in the catalog list.

Main Flow

1. The user opens the **Catalog** option.
2. Clicks on the **"+ New medicine"** button.
3. Fills in the required fields (name, description, sale price).

4. Clicks **Save**.
5. The frontend sends a `POST /api/medicines/` request with the new data.
6. The backend validates the information and stores the record.
7. The API responds with status **201 Created** and the created object.
8. The frontend displays a confirmation and updates the table.

4.2 Use Case: Read Medicine

Actor: Pharmacy employee
Description: The actor visualizes the list of registered medicines.
Main Flow (simplified)

1. User goes to **Catalog**.
2. Frontend calls `GET /api/medicines/`.
3. API returns the list in JSON.
4. Frontend renders the table with the medicines.

4.3 Use Case: Update Medicine

Actor: Pharmacy employee
Description: The actor updates the information of an existing medicine.
Main Flow (simplified)

1. In **Catalog**, the user clicks **Edit** on a specific medicine.
2. The form is loaded with current data.
3. User modifies fields and saves.
4. Frontend sends `PUT/PATCH /api/medicines/{id}/`.
5. Backend validates, updates and returns the modified record.

4.4 Use Case: Delete Medicine

Actor: Pharmacy employee
Description: The actor deletes a medicine that should no longer appear in the catalog.
Main Flow (simplified)

1. In **Catalog**, the user clicks **Delete** on a medicine.

2. A confirmation dialog appears.
3. If confirmed, the frontend sends `DELETE /api/medicines/{id}/`.
4. Backend removes the record and returns **204 No Content**.
5. The frontend removes the row from the table.

---

5. Backend Documentation

5.1 Backend Architecture

The backend is organized as a **Django project** with multiple apps, each one representing a domain area:

- `catalog` app – models and API for medicines
- `customers` app – models and API for customers
- `prescriptions` app – prescriptions and prescription items
- `sales` app – sales and sale items

Each app uses:

- `models.py` – database models
- `serializers.py` – DRF serializers
- `views.py` – API views / viewsets
- `urls.py` – routing for each resource

5.2 Folder Structure

```
Backend-Farma_rapid/
|-- manage.py
|-- farma_rapid/        # Project configuration
|   |-- settings.py
|   |-- urls.py
|   `-- wsgi.py
|-- catalog/
|   |-- models.py
|   |-- serializers.py
|   |-- views.py
|   `-- urls.py
|-- customers/
|   |-- models.py
```

```
|   |-- serializers.py
|   |-- views.py
|   `-- urls.py
|-- prescriptions/
|   |-- models.py
|   |-- serializers.py
|   |-- views.py
|   `-- urls.py
|-- sales/
|   |-- models.py
|   |-- serializers.py
|   |-- views.py
|   `-- urls.py
`-- requirements.txt
```

5.3 API Documentation (REST)

Only one entity is detailed here; repeat the pattern for the others.

Resource: `/api/medicines/` GET `/api/medicines/`
Purpose: List all medicines.

- **200 OK** – array of medicine objects.

POST `/api/medicines/`
Purpose: Create a new medicine.

Request body example:

“‘json { “name”: “Ibuprofen 400mg”, “description”: “Anti-inflammatory”, “sale_price”: 5500.00 }

5.3 API Documentation (REST) Resource: /api/medicines/ GET /api/medicines/

Purpose: Retrieve a list of all medicines.

Successful Response

Code Meaning 200 OK – Returns array of medicines POST /api/medicines/

Purpose: Create a new medicine.

Request Body Example

{ “name”: “Ibuprofen 400mg”, “description”: “Anti-inflammatory”, “sale_price”: 5500.00 }

Responses

Code Meaning 201 Created – Object successfully created 400 Bad Request – Validation errors GET /api/medicines/{id}/

Purpose: Retrieve a single medicine by ID.

Response

Code Meaning 200 OK 404 Not Found PUT/PATCH /api/medicines/{id}/

Purpose: Update an existing medicine.

Response

Code Meaning 200 OK – Updated successfully 400 Bad Request – Invalid data 404 Not Found DELETE /api/medicines/{id}/

Purpose: Delete a medicine.

Response

Code Meaning 204 No Content – Deleted 404 Not Found 5.4 REST Client (VS Code)

The API was tested using the REST Client extension in Visual Studio Code.

For each module (medicines, customers, sales), a corresponding .http file was created containing example requests.

Example of a .http test file ### Get all medicines GET http://localhost:8000/api/medicines/

Create new medicine

POST http://localhost:8000/api/medicines/ Content-Type: application/json

{ "name": "Paracetamol 500mg", "description": "Pain reliever", "sale_price": 4200 }

Get medicine by ID

GET http://localhost:8000/api/medicines/1/

Update medicine

PUT http://localhost:8000/api/medicines/1/ Content-Type: application/json

{ "name": "Paracetamol 650mg", "description": "Pain reliever", "sale_price": 5000 }

Delete medicine

DELETE http://localhost:8000/api/medicines/1/

6. Frontend Documentation

6.1 Technical Frontend Documentation

Framework used: **Angular 17**

Main concepts: **Components, Services and Routing**

Folder Structure (simplified)

```
src/
|-- app/
|   |-- app.routes.ts
|   |-- models/
|   |   |-- medicine.ts
|   |   |-- customer.ts
|   |   `-- sale.ts
```

```
|   |
|   |-- services/
|   |   |-- medicine.service.ts
|   |   |-- customer.service.ts
|   |   `-- sales.service.ts
|   |
|   |-- pages/
|   |   |-- catalog/
|   |   |   `-- medicines/
|   |   |       |-- medicines.component.ts
|   |   |       |-- medicines.component.html
|   |   |       `-- medicines.component.css
|   |   |
|   |   |-- customers/
|   |   |   |-- getall/
|   |   |   |-- create/
|   |   |   |-- update/
|   |   |   `-- delete/
|   |   |
|   |   |-- recipes/       # prescriptions list
|   |   |   `-- getall/
|   |   |
|   |   `-- sales/
|   |       `-- getall/
|   |
|   `-- components/
|       `-- layout/        # main layout and navigation
|
`-- assets/
```

Models: TypeScript interfaces describing the JSON structure returned by the API.

Services: Angular services using HttpClient to call the REST endpoints.

Components: Visual parts of the SPA (catalog table, customer list, sales list, etc.).

6.2 Visual Explanation of the System's Operation

Here you should insert screenshots of:

- The main layout (left navigation + header).
- The Catalog screen with the medicines table.
- The Customers screen with actions **Edit** and **Delete**.
- The Sales screen with registered sales.

7. Frontend–Backend Integration

All services use a base URL, for example:

```
private baseUrl = 'http://localhost:8000/api';
```

Each service method builds the final endpoint, for example:

```
getAll(): Observable<MedicineResponseI[]> {
  return this.http.get<MedicineResponseI[]>(
    `${this.baseUrl}/medicines/`
  );
}
```

CORS is enabled on the backend so that the Angular app (running on port 4200) can communicate with the API (port 8000).

Errors from the API (400, 404, 500) are captured in the `subscribe` error callback and can be displayed as messages in the interface.

8. Conclusions & Recommendations

The Farma Rapid project demonstrates how to integrate a Django REST API with a modern Angular frontend to build a complete CRUD system.
The modular organization of the system (catalog, customers, prescriptions, sales) simplifies maintenance, improves scalability and allows the system to grow in a controlled manner.
The structure of the backend using Django REST Framework and the use of routing and services in Angular provide a clean separation of responsibilities, ensuring that both performance and clarity are preserved.
In general, the system meets the requirements of a small pharmacy management solution by enabling efficient registration, editing and consultation of medicines, customers, prescriptions and sales.

Recommendations

- Add authentication and authorization mechanisms for different user roles (administrator, cashier, etc.).
- Implement unit tests for the main services and components.
- Add form validation on the frontend to prevent invalid data before sending requests.
- Deploy the solution to a cloud environment (e.g., Render, Railway, AWS) to make it accessible outside the local network.
- Add response messages and better error handling in the user interface.

9. Annexes (Optional)

- ERD and logical model exported as images.
- Screenshots of REST Client `.http` files used for testing the API.
- Deployment steps (environment variables, migrations, etc.).