University of Camerino

School of Science and Technologies

Course Master Degree In Computer Science (LM-18)
Complex Systems Design

Digital Management of Physiofitness clinic

Student
Alessio Re - alessio.re@studenti.unicam.it

Registration 135435

Supervisor
Flavio Corradini

Coadvisor
Edoardo Re

A.Y. 2025/2026

Abstract

As digitalization in healthcare and commercial is becoming more and more common, this project sets the goal of producing a prototype for an ecosystem of applications that lets small facilities have their own digitalized back and front office. Commercial activities are provided with these tools to help them with activities such as scheduling, CRM, billing, inventory, and predictive analytics. In addition, the application will include privacy controls and full and up-to-date security regarding role-based access. Being provided with such tools will guarantee to the facilities the possibility to put more focus on other key activities, and not on managing the data. The project is designed to set a benchmark for digital management solutions applicable to retail stores, veterinary clinics, insurance offices, and related sectors. Since the system is set to include rich data modules, AI can add value wherever there's prediction, automation, and decision support. The implementation of a virtual assistant (CRM module) will help with behavior profiling and answering FAQs. Moreover, AI can optimize and automate the scheduling process, or anticipate future events based on historical data, learning recurrent patterns, and eventually automate repetitive tasks and reminders, significantly reducing the risk of human operational errors. Most of the features will be included in a useful web application, implemented to even provide a mobile-friendly UI. The application is willing to introduce the user to an environment that covers several activities in the management of a facility, and with the help of AI, it will surely result in a helpful system.

Introduction

Building on trends in healthcare technology, my system will offer scheduling, CRM (Customer Relationship Management), billing, inventory, and basic analytics for a small physiotherapy clinic, all via a web/mobile-friendly interface. This mirrors modern clinic systems that automate patient management and administrative tasks. The goal is to streamline clinic operations (appointments, patient records, invoicing, supplies) and use AI for insights (i.e. predicting busy periods or no-shows). The project will follow a MVP approach: first define scope, compliance, and data needs; then build a lean MVP focusing on core features (users/auth, patient data, scheduling). Industry data suggest healthcare leaders are prioritizing digital tools and AI to improve efficiency. My prototype aligns with this. Another goal is to automate tasks (using Celery for reminders) and to provide simple forecasts to help staff plan.

Modules

I will implement four main modules corresponding to typical clinic needs:

Scheduling/Calendar: Online booking and appointment calendar for patients and staff. The system will allow creating and viewing appointments in a calendar. Automated reminders (email/SMS) before appointments will be handled by background tasks (through Celery implementation) to reduce no-shows. Automated alerts in existing physiotherapy software have been shown to cut no-show rates.

CRM/Patient Records: A secure patient database with contact info, medical history, and treatment notes. This acts as a simple EHR (Electronic Health Record) for the clinic. It will store each patient's profile, visit history, insurance info, etc. Physio practice software often includes full patient history and charting.

Billing/Invoicing: An invoicing system to generate patient bills and track payments. It can handle co-pays, insurance billing codes, and optionally integrate with Italy's e-invoicing standard later. Existing solutions support medical billing within the EHR. It will include basic invoices/receipts and mark paid or unpaid status.

Inventory/Supply Management: Track stocks of consumables (bands, linens, supplements, etc.). The system will record stock levels and alert admins when items fall below reorder thresholds. This helps clinics manage supplies without manual logs.

Each module will be implemented as a Django app and exposed via a clear data model and API. I will use Django's built-in user model, plus custom profiles for staff, and extend it with Role-Based Access Control (roles like Admin, Physiotherapist, Receptionist, each with permissions on modules). Django supports roles and permissions out of the box, allowing me to define groups for each staff type and restrict access accordingly. For auditing and privacy, Django's logging and middleware can record

changes to sensitive records, and I'll enforce encryption and GDPR/HIPAA-level data protection as a baseline.

Timeline

A realistic solo-developer schedule is 4–6 months. For illustration, here is a 15-week plan:

Figure 1: Project Gantt Chart

Weeks 1–2: Requirements & Design - Finalize scope with the clinic, sketch UI/UX, define data model and compliance needs (HIPAA/GDPR). Set up project repo and development environment (Django project with PostgreSQL).

Weeks 3–4: Core Backend Setup & Auth - Implement user roles (Django auth/groups). Create base models: Patient, Staff (with role), and Clinic resources. Set up Django admin and DRF basics.

Weeks 5–8: Develop Core Modules - Build out Calendar, Scheduling and CRM: appointment model, views/APIs, calendar UI. Add patient EHR fields. Simultaneously develop Billing: invoice model, bill generation, payment tracking. Overlap tasks where possible. By week 8, core CRUD for patients and appointments should work.

Weeks 7–10: Inventory & Notifications - Implement inventory models (items, stock levels). Integrate Celery: write tasks for sending emails or SMS reminders. Create a basic reminder (i.e. 24h before appointment). Test end-to-end: booking an appointment sends an email.

Weeks 9–12: AI Features & Analytics - Use collected data to prototype the predictive modules. For example, train a simple no-show prediction model on sample data, integrate it so staff can flag high-risk appointments (UI indicator). Build a demand forecast dashboard (i.e. a line chart of expected appointments per day).

Weeks 11–14: Frontend Polishing and Integration - Develop React components and pages for all features. Make UI mobile-responsive. Connect frontend to the DRF API endpoints built earlier. Conduct functional testing of workflows.

Weeks 13–15: Refinement & Pilot Deployment - Write unit tests for key functionality. Generate API documentation (Swagger-like). Refine based on self-review and any early feedback. Integrate the application with one real physiotherapy clinic (with test data).

This schedule follows agile MVP principles: focusing first on core operations before other features.

Project Architecture

The system can be organized into layered components (e.g. a web/mobile front end, an application server, and a database), often following an MVC (Model-View-Controller) pattern. In MVC, the Model holds data and business logic, the View is the user interface, and the Controller mediates input and updates. This decoupling makes it easier to maintain and extend each part independently.

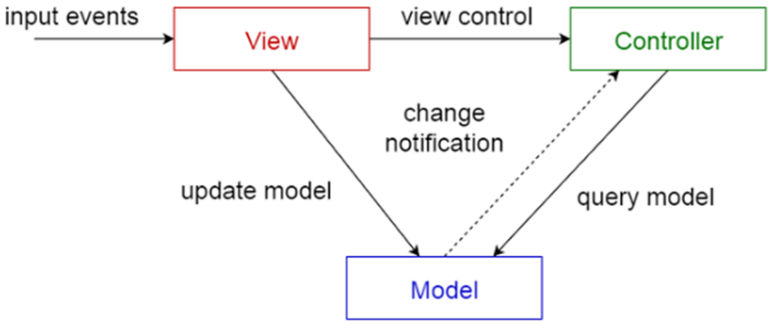


Figure 2: Model-View-Controller Architecture

In this model, user events trigger the View, which notifies the Controller; the Controller then updates or queries the Model, and any change in the Model updates the View. This structure underpins the static design: for instance, the *Patient* and *Appointment* classes (in the Model layer) encapsulate data and rules, while controllers handle requests (e.g. booking or billing).

In line with software engineering best practices, I've separated static elements (classes, databases, modules) from dynamic elements (workflows, user interactions)

The constraints of a 15 week timeline executed by a solo developer necessitate a unified codebase. However, this monolith must be structured with strict logical boundaries between domains (specifically Scheduling, CRM, Billing, and Inventory) to facilitate a potential future decomposition into microservices without requiring a total rewrite.

Components

Each module is a distinct subsystem in the architecture, working separately in a distributed and parallel application. The design also enforces role-based access: Admin users configure the system and oversee all data; Physiotherapists (and Trainers) access patient records, update treatments, and view their schedules; Receptionists handle scheduling, check-in/out, and billing tasks; Patients use a portal to book appointments and view their own data. These roles map to modules' interfaces, ensuring, for example, that only a receptionist can finalize payments.

Module 1 - Scheduling and Calendar

This is the main module of the clinic: it must be reliable, consistent, and fast.

Responsibilities

The Scheduling/Calendar module is responsible for all functionality related to availability and time-slot management in the clinic:

- Represent clinician (physiotherapist/trainer) working hours or days off.
- Publish clinician availability and allow receptionist/patient to search/book time slots for services.
- Create, update, cancel, reschedule appointments; enforce business rules (buffer times, service duration).
- Provide calendar views (for instance, filtered for clinician or for therapy) for staff and a patient view for their own bookings.
- Notifications (email/SMS).
- Provide endpoints for front-end calendar UI and for external calendar sync, like Google Calendar or iCal, if enabled (Optional).

Data Entities

- Clinician (StaffProfile or referenced user)
- Service
- AvailabilitySlot
- CalendarBlock
- Appointment
- Room/Treatment

Access Control and Roles

- Patients: can view and create their own appointments (subject to policies), cancel within allowed window.
- Receptionists: create, reschedule, or cancel any appointment for clinic, check clients in/out and payments.
- Physiotherapists/Trainers: view own calendar, block time (set exceptions), mark appointment as completed, add session notes.
- Admin: full control (manage clinicians, rooms, global rules).

Critical Points

Booking conflicts are the main hazard. Use a combination of database constraints and transactions, together with constant checks on booking blocks (if overlapping booking, abort with conflict error); at last, use client-sent idempotency keys for retries to avoid double bookings.

Integration with other Modules

CRM Appointment references patient ID; APIs fetch patient summary (name, contact, clinical notes) to show during booking/check-in.

Billing On booking or completion, module emits events to Billing, or Billing subscribes to appointment-creation events.

Inventory For services consuming supplies, the booking or completion can create inventory reservations or decrement counts.

Notifications sends messages via Notification component (Celery tasks).

AuditLog and Analytics all state changes logged for compliance and to feed ML pipelines (no-show predictor, demand forecasting).

Roadmap

1. Implement DB models (Appointment, CalendarBlock, AvailabilitySlot, AuditLog).
2. Add Postgres exclusion constraint for time ranges or implement overlap check + transaction.
3. Implement a scheduling service with find_available_slots and book_appointment, tested in isolation.
4. Expose DRF endpoints and basic React calendar UI.
5. Add Celery tasks for notifications and no-show detection.
6. Add external calendar adapter (optional).
7. AI readiness for predictive availability.
8. Add monitoring and load tests.

Module 2 - CRM and Patient Records

This module is central to clinical workflows. Accuracy, integrity, and privacy are critical.

Responsibilities

The CRM/Patient Records module is the store for patient-centered data. Its responsibilities:

- Persist and expose patient demographic data and contact info.
- Store medical history, clinical notes, treatment plans, progress measures and outcomes.
- Manage documents and binary attachments such as forms and images.
- Provide patient summary views for clinicians and patient-facing summaries for patients.
- Version and audit clinical notes.
- Provide search and filtering (by name, id, DOB, clinical flags).
- Enforce fine-grained access control (who may view or edit which fields).
- Provide de-identified data exports for analytics and eventual AI training.

Data Entities

- Patient
- PatientRecord
- TreatmentPlan
- Document/Attachment
- Flags (alerts: e.g. allergy, fall risk)

Access Control and Roles

- Admin: read or write all patient data.
- Physiotherapist/Trainer: read or write patient records for patients under their care; write notes; see health flags.
- Receptionist: read or write contact and registration fields; cannot edit clinical notes.
- Patient: read own basic data and patient-visible notes, upload documents, fill self-reported metrics.

Critical Points

Field-level restrictions are mandatory. Clinical notes and sensitive identifiers only visible to clinicians and admins. Documents that are flagged *clinician-only* must be hidden to patient portal. If a patient has denied sharing for analytics, exclude them from analytics exports.

Integration with other Modules

Scheduling Appointment endpoints fetch minimal patient summary. When a note is created after appointment completion, record links to appointment via id.

Billing Uses patient id for invoices; treatment codes in notes can map to billable items.

Inventory Notes may record consumables used so that they trigger inventory decrement events.

Roadmap

1. Implement DB models with migrations (Patient, PatientRecord, Document, Treatment, Flags).
2. Add AuditLog integration and versioning for notes.
3. Implement PatientRecord service.
4. Create DRF serializers and viewsets and object-level permissions.
5. Add document storage adapter (optional).
6. Implement consent manager and analytics export adapter(optional).
7. AI integration.

Module 3 - Billing

This module must be accurate, auditable, and have robust failure handling for external payments.

Responsibilities

The Billing module is responsible for all financial processes related to clinic services. Its responsibilities include:

- Representing billable services and price management.
- Creating, issuing and storing invoices linked to appointments, treatment plans or packages.
- Recording payments and refunds, handling multiple payment methods, and partial payments.
- Managing billing states and business rules such as taxes or discounts.
- Producing receipts and exportable financial reports.
- Integrating with third-party payment gateways.
- Enforcing financial audit trails and immutable payment logs for compliance.

Data Entities

- PriceItem (service pricing)
- Invoice
- Payment

Access Control and Roles

- Admin: create or edit invoices, override pricing, run reports.
- Receptionist: issue invoices, process cash or bank transfer payments, view reports.
- Patient: view own invoices, pay invoices.
- Physiotherapist/Trainer: view invoice summary for their services (read-only).

Critical Points

Payments are a critical external call. It is mandatory to ensure idempotency, as client and webhook requests represent keys to avoid double charges of a bill. Same significance is given to atomicity, maintained by wrapping DB updates (Payment and Invoice) in transactions. It's always important to verify signatures and authenticate sources.

Integration with other Modules

Notification Send invoices, payment confirmations.

Roadmap

1. Implement DB models and migrations (Invoice, PriceItem, Payment).
2. Implement Billing service and payment gateway adapter interface.
3. Implement webhooks handler (idempotent, authenticated).
4. Add DRF endpoints and PDF generator (for invoices).
5. Add monitoring and logging.

Module 4 - Inventory

The Inventory module is data-centric, feeding operational and financial modules (Billing, Scheduling) with real-time resource availability and consumption metrics.

Responsibilities

The Inventory module supports daily clinic operations by maintaining an accurate record of stock levels, purchase and usage history, and supplier information. Its main goals are:

- Tracking stock quantities, locations, and statuses (available, reserved, expired, out of stock).

- Managing reorder thresholds and generating automatic restock alerts.
- Registering incoming and outgoing transactions such as purchases.
- Enabling inventory audits, cost control, and waste minimization (AI).
- Providing usage reports that link consumables to specific appointments or staff (optional).
- Offering an integration point for AI-based demand forecasting and predictive restocking.

Data Entities

- InventoryItem
- Supplier
- PurchaseOrder
- PurchaseOrderItem
- InventoryTransaction (built in Audit Log, only for inventory)

Access Control and Roles

- Admin: full CRUD access.
- Receptionist: can update stock, process restocks.
- Physiotherapist/Trainer: can register used materials.
- Patient: no access.

Critical Points

Restrict edit operations to authorized personnel. All operations recorded immutably (Audit Log) and guarantee data integrity using database constraints to prevent negative stock values.

Integration with other Modules

Notification Connects with email or internal alerts for low-stock messages.

Billing Registering items codes and names on invoices.

Roadmap

1. Implement InventoryItem, InventoryTransaction, and basic CRUD.
2. Add reorder threshold logic and notifications.
3. Introduce role-based access and audit views.
4. Add PurchaseOrder and Supplier models with UI.

5. Add analytics dashboards and forecasting engine(optional).

6. Add AI-based reorder prediction and supplier optimization.

Data Management

A critical focus is on data. The system collects rich patient data (demographics, clinical notes, progress metrics) and operational data (appointment histories, billing, inventory usage). All of this can feed analytics to improve care and operations. Modern healthcare emphasizes data-driven methods. Concretely, a central EHR database stores all patient-related data, while a separate data warehouse or lake could aggregate anonymized data for analysis. Built-in analytics tools can track trends (e.g. appointment no-shows, common diagnoses). For instance, the system might analyze appointment trends, patient demographics, and treatment outcomes to derive insights and optimize resources. Dashboards would present KPIs (clinic utilization, revenue, patient recovery rates). Looking ahead, machine learning and AI can be layered on this data. For example, predictive analytics can forecast patient outcomes or no-show probabilities. AI modules could also automate tasks like scheduling, billing, and patient communication. In my architecture, this means designing extensible data pipelines and possibly using pattern-based modules so that when new data arrives, all interested components get updated in real time. In summary, the architecture must include robust data storage (relational DB for EHR and possibly a data lakehouse for analytics) and ensure data flows securely between modules.

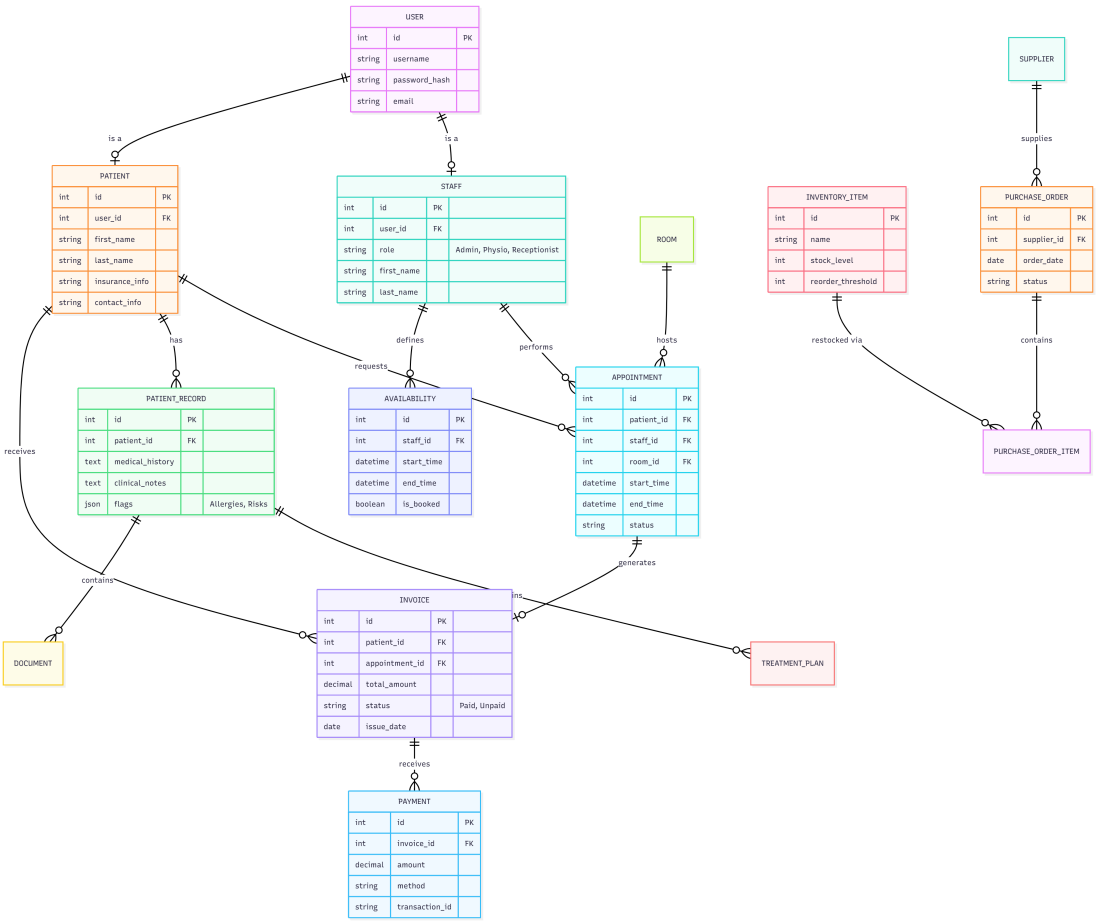
Security is built into the data layer and in integrations. For example, every API call checks the user's role. Using a layered architecture helps: the data-access layer handles all SQL/ORM calls, the business layer enforces rules, and the presentation layer (web UI/mobile app) is separated.

Rather than selling data, the project's value lies in data-driven services: Providing insight dashboards to clinic administrators. Offering AI-assisted scheduling and treatment outcome predictions. Creating an ecosystem where anonymized aggregate insights can support research or policy planning without compromising user privacy. If in the future the project integrates with partners (e.g., insurance systems or health research networks), data sharing agreements will be established to ensure full anonymization and compliance.

In a professional environment, especially healthcare, it's recommended to never actually delete data. If a doctor leaves the clinic, or a room is under maintenance, they cannot be deleted from the database because they are linked to past appointments. If a Room row is deleted, all history of appointments in that Room would break or vanish.

ER Diagram

This is the ER Diagram derived from the architecture described above:



Workflows & Use Cases

The system's dynamic behavior is captured by workflows and use-case scenarios. For example, the patient appointment process involves multiple actors: The Patient requests an appointment; the Receptionist checks the therapist's availability and confirms it; the appointment is scheduled. On the appointment day, the Physiotherapist sees the patient. Afterward, the Patient pays; the Receptionist records the payment, and the Accounting module confirms it. This example illustrates common steps: booking, confirmation, treatment, and billing. Other workflows include:

- Patient Registration: New patients provide personal and insurance data (via Receptionist or patient portal). The CRM module creates a Patient record;
- Appointment Booking: Patients or Receptionists schedule, reschedule, or cancel appointments. The system checks clinician availability and updates the calendar. Notifications may be sent.
- Patient Visit and Treatment: The physiotherapist accesses the patient record, conducts therapy, and enters treatment notes/outcomes. Completed sessions may update health metrics.
- Billing and Payment: After a treatment, the system generates an invoice linked to the appointment. The patient or insurer is billed (external payment integration), and the payment status updates the record. Receipts are issued.
- Inventory Use: When a treatment uses supplies, the clinician or receptionist records them. The inventory module deducts quantities and may trigger reordering

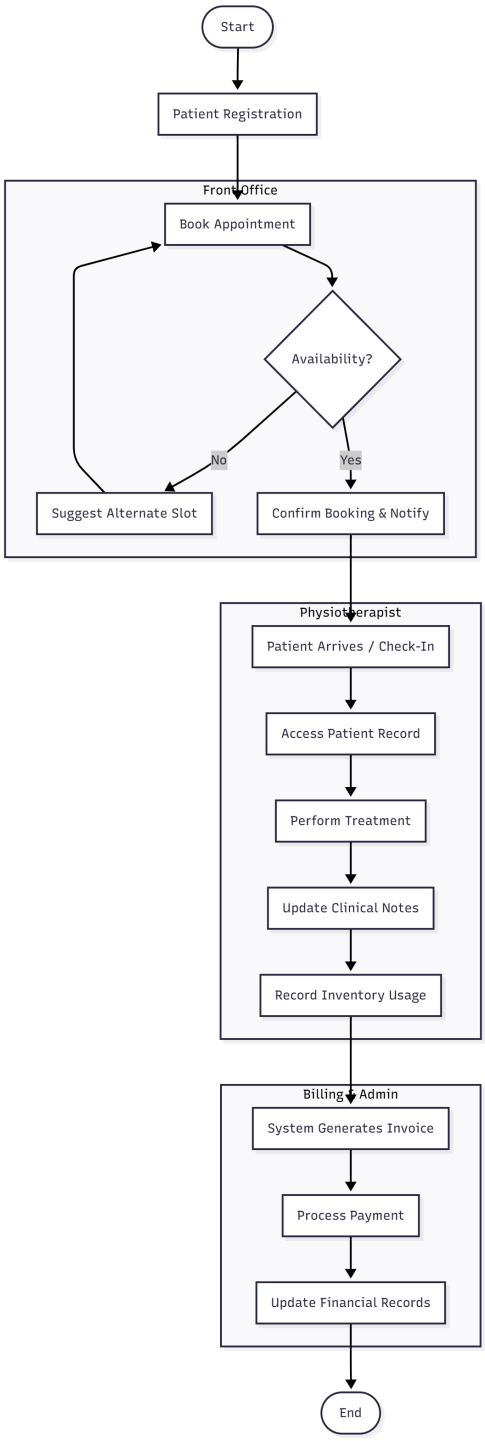


Figure 3: Workflow of a visit

Core Use Cases

- As a physiotherapist, I want to view my schedule so I can plan my day;
- As an admin, I want to manage clinic staff roles;
- As a receptionist, I want to create patient profiles so we can register new clients;
- As a user, I want to register my account or sign in it;
- As a patient, I want to view my appointments;
- As the system, I must send appointment reminders to reduce no-shows.

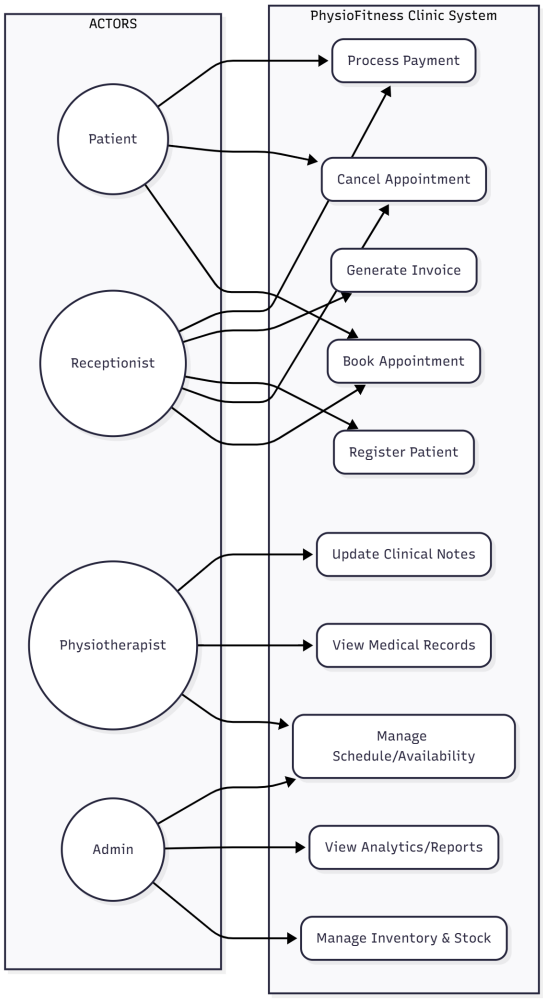


Figure 4: Use Case Diagram (Actors and Interactions)

I have designed a sequence diagram regarding the interactions between all the different layers in the system (or the application, in this case). This sequence diagram provides a crucial, time-ordered view of how the system's components collaborate to execute the "New Patient Appointment Booking" workflow. It validates the chosen Model-View-Controller (MVC) architecture by showing the precise order of messages:

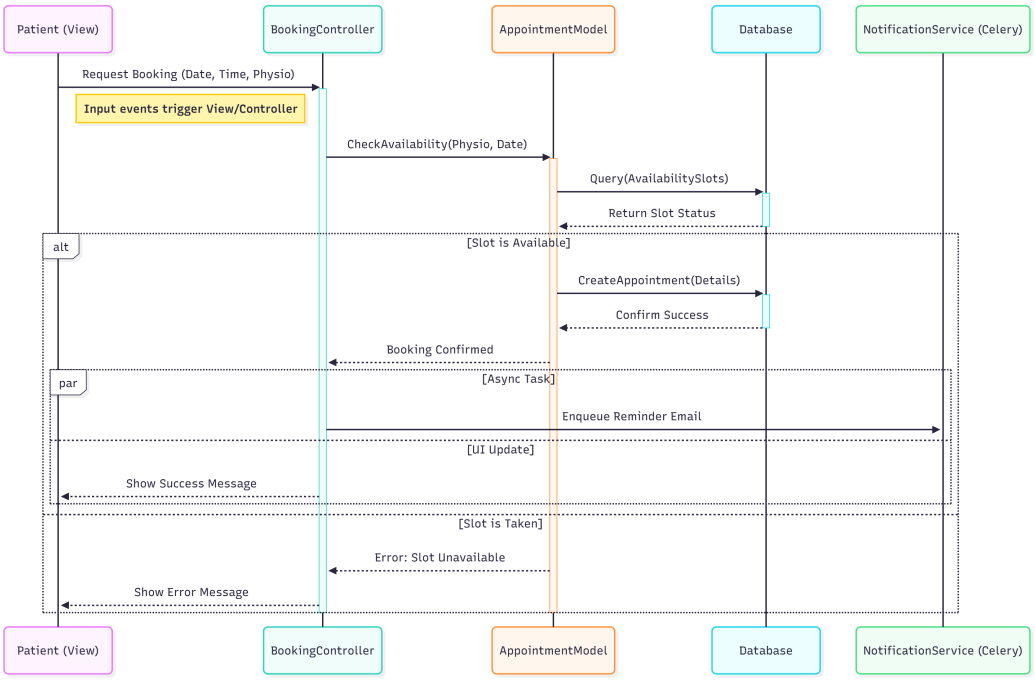


Figure 5: Sequence Diagram (Appointment Booking)

Components Workflows

In all the workflows, I have distinguished one for each of the 4 modules presented above. This will help better understand the functionalities of the finished application. This visualization shows exactly how different modules interact (e.g., how a completed appointment in the Scheduling module triggers an event in the Billing module) and clarifies the responsibilities of each actor. By mapping these interactions, I can easily identify dependencies, ensure data flows correctly, and confirm that all use cases are fully supported from start to finish. Next 4 images represent all the different components workflows, which provide solid architecture to start implementing the application code.

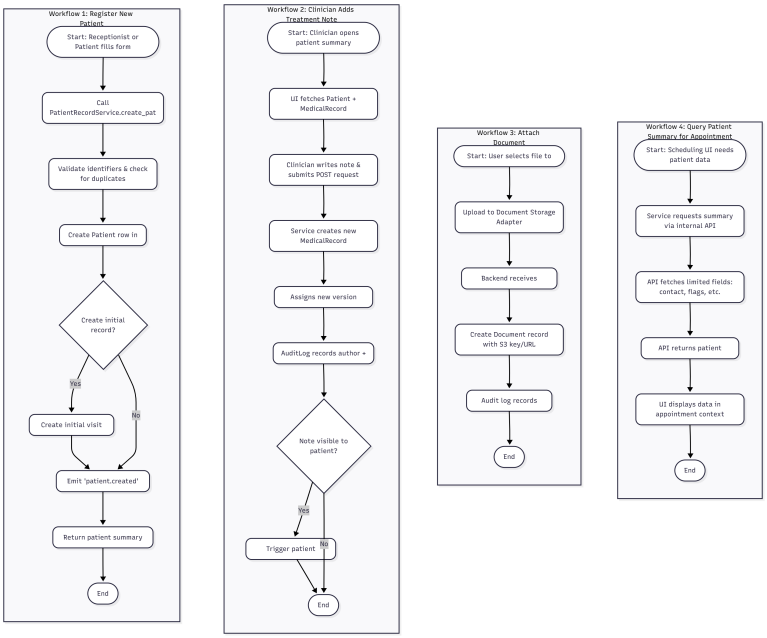


Figure 7: CRM Workflow Example

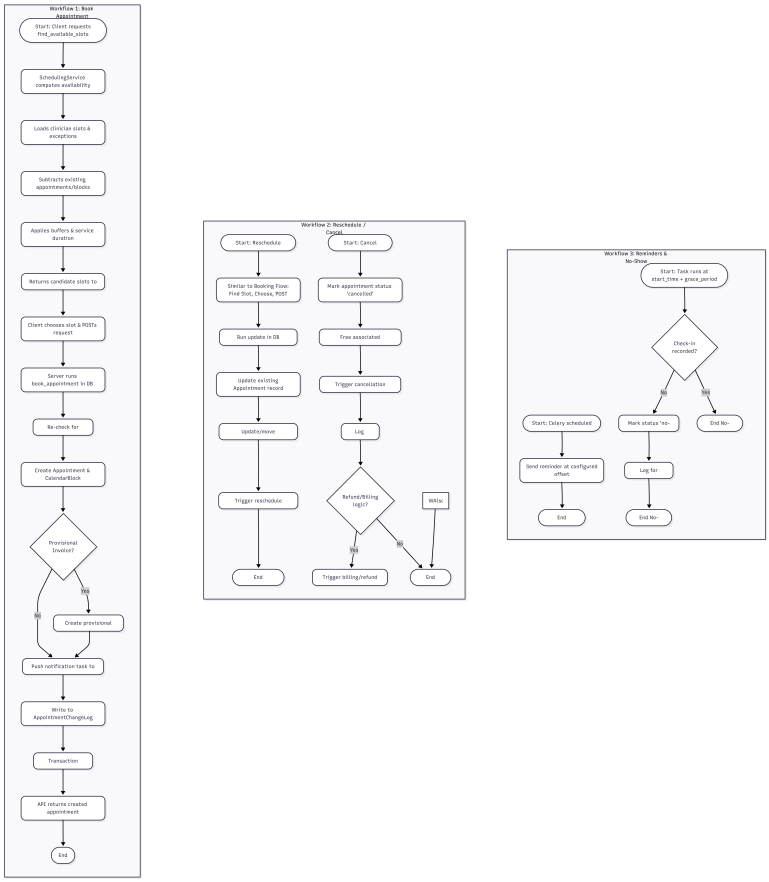


Figure 6: Schedule Workflow Example

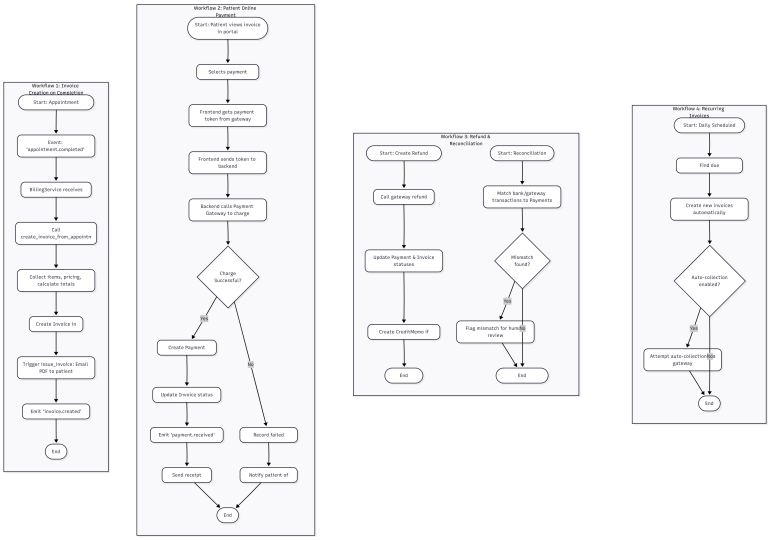


Figure 8: Billing Workflow Example

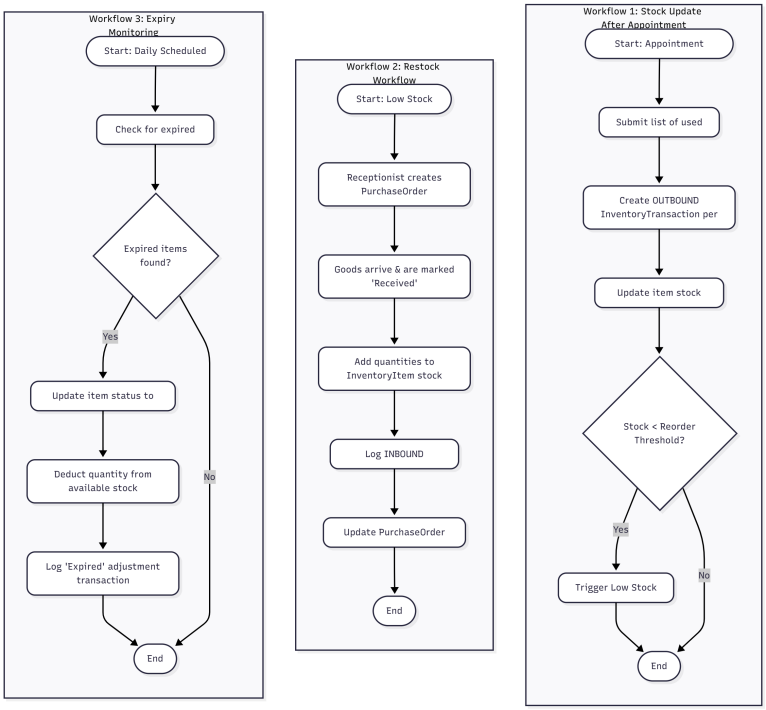


Figure 9: Inventory Workflow Example

Privacy & Compliance Notes

There are two different practices depending on whether the application is used in Europe or in the US. The first one requires private medical data management that satisfies the **GDPR**, and the second one states that Protected Health Information should satisfy **HIPAA** standards. So, these are the best considerations I have come across:

- Identify medical data as sensitive personal data
- Define lawful basis and explicit consent mechanism for data processing
- Data minimization and purpose limitation principles are guaranteed
- Secure storage via encryption at rest and HTTPS for transport
- Role-based access control and strict authentication implemented
- Audit logging for access to patient records and data changes, keeping private data safe.
- Document data retention policy and support user rights (access, correction, deletion)
- Incident response protocol and breach reporting (72-hour rule under GDPR)
- Ensure third-party providers meet compliance

Design Patterns

To implement the architecture cleanly, I will apply well known design patterns. For example:

Model-View-Controller As described above, MVC separates concerns.

Observer (Publish-Subscribe) Useful for event-driven updates. If patient data changes, the Observer pattern can notify all relevant modules (e.g. update analytics, notify clinician) without tight coupling;

Factory For creating standard objects (invoices, reports, or UI components), the Factory pattern ensures consistency. For instance, an InvoiceFactory could produce correctly formatted invoices for different billings;

Singleton For shared resources like database connections or configuration managers, the Singleton pattern ensures a single point of access.

Per this article, applying these patterns will support maintainability.

API Docs

I will build the APIs module by module, mirroring the work I did on the models. This keeps the project organized.

- **Phase 1**: Scheduling API (The Core) - Expose Rooms, Services, and Appointments.
  Critical Logic: Filtering appointments (Physios see their own; Receptionists see all);

GET /api/scheduling/rooms/

Description: Retrieves a list of all physical rooms available in the clinic.

Access Level: Staff / Admin
Response: JSON list of room objects (id, name, capacity).

GET /api/scheduling/services/

Description: Retrieves a list of available treatments and medical services offered by the clinic.

Access Level: Authenticated
Use Case: Populates the "Select Service" dropdown during booking.

GET /api/scheduling/appointments/

Description: Retrieves a list of appointments. The output is automatically filtered based on the requester's role.

Access Level: Authenticated
Behavior:

- **Patients:** See only their own booking history.
- **Staff:** See all appointments for the clinic.

POST /api/scheduling/appointments/

Description: Creates a new appointment booking.

Access Level: Authenticated
Input Body: `service_id`, `room_id`, `start_time`

PATCH /api/scheduling/appointments/{id}/

Description: Partially updates an existing appointment (e.g. to reschedule) or changes its status (e.g. to Cancelled).

Access Level: Owner / Staff
Input Body: `start_time` (for rescheduling) OR `status` (to cancel).

- **Phase 2**: Patient API - Expose Patient records.
  Critical Logic: Search functionality (search by name/fiscal code).

GET /api/patients/

Description: Retrieves a list of patients registered in the clinic. Supports filtering by name via query parameters.

Access Level: Staff (Full Access) / Patient (Restricted to Own Profile)
Query Params: `?search=Name` (Optional)

POST /api/patients/

Description: Creates a new patient record manually (e.g., performed by receptionists or doctors).

Access Level: Staff Only
Input Body: `first_name`, `last_name`, `dob`, `contact_details`

GET /api/patients/{id}/

Description: Retrieves the full profile and medical history of a specific patient.

Access Level: **Staff / Owner** (The patient themselves)
Response: Detailed JSON object including past appointments and clinical notes.

- **Phase 3**: Inventory API - Expose Items and Transactions.
  Critical Logic: Read-only operations.

GET /api/inventory/items/

Description: Retrieves the current stock levels for all inventory items registered in the system.

Access Level: Staff Only
Response: JSON list containing item details, current quantity, and low-stock threshold indicators.

POST /api/inventory/transactions/

Description: Logs a new stock transaction. This endpoint is used to record both incoming supplies (restocking) and outgoing usage (consumption during treatments).

Access Level: Staff Only
Input Body: `item_id`, `quantity`, `transaction_type` (`INBOUND` or `OUTBOUND`)

- **Phase 4**: Billing API - Expose Invoices and Payments.
  Critical Logic: Creating an invoice automatically from an appointment.

POST `/api/billing/invoices/`

Description: Generates a new invoice record. It automatically links to a specific appointment and calculates the total based on the service provided.

Access Level: Staff Only
Input Body: `appointment_id`, `due_date` (optional)
Behavior: Auto-populates line items from the appointment's service price.

POST `/api/billing/payments/`

Description: Records a payment against an existing invoice.

Access Level: Staff Only
Input Body: `invoice_id`, `amount`, `payment_method`
Side Effect: If the recorded payment covers the remaining balance, the Invoice status automatically updates from `DRAFT/UNPAID` to `PAID`.

- **Extra**: Authentication API - Expose all Authentication needs, such as logged User info and Access and Refresh tokens.

POST `/api/token/`

Description: Obtains a token couple (Access and Refresh) sending the credentials. Works as a Login.

Access Level: Public
Input Body: `username`, `password`

POST `/api/token/refresh/`

Description: Asks for a new valid Access token, using a still valid Refresh token.

Access Level: Public
Input Body: `refresh` (token)

POST `/api/users/register/`

Description: Register a new User/Patient account in the system.

Access Level: Public
Input Body: `email`, `password`, `first_name`, `last_name`

GET /api/users/me/

Description: Returns logged user profile (based on the token sent in the header).

Access Level: **Authenticated** (Bearer token needed)
Response: JSON with user data.

Swagger API

Before moving to frontend implementation, it would be helpful implementing a Swagger API to have a clear visual on how to obtain the data and what data is it possible to access (Figure 10).
It was set as a weeks 13-15 milestone, but adding this before moving on to the next iteration will speed up the process of designing the frontend (accessible from the `/api/docs/` url).

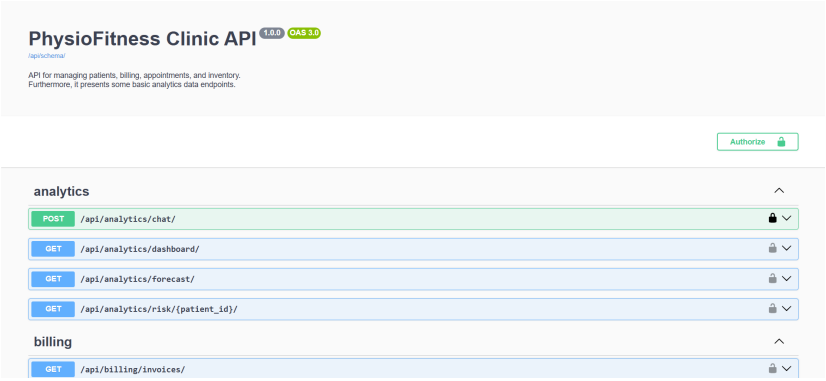


Figure 10: A preview of the Swagger API

System Testing Report: Backend Foundation (Weeks 1-8)

This testing phase validates the core backend architecture implemented during the first two development iterations. The focus is on Data Integrity, Role-Based Access Control, and Cross-Module Automation (e.g., Inventory updates triggering from transactions, Invoice generation from appointments).

Date: `25/11/2025`
Tester: `Alessio Re`
Environment: `Local Development (Django Admin & DRF Browsable API)`

Test Suite A - End-to-End Workflow

Objective: Simulate a complete patient journey to verify module integration.

Table 1: Test Suite A

Test ID	**Scenario**	**Steps to Execute**	**Expected Result**	**Status**
E2E-01	Patient Registration	1. Use API/Admin to create a new Patient. 2. Check Patient model.	A Patient profile is created	PASS
E2E-01.1	Claiming Patient Status	1. User signs in, creating new account. 2. Email automatically links to Patient email. 3. User is linked to Patient status, sharing data.	User email appears in Patient email field.	PASS
E2E-02	Appointment Booking	1. Login as Receptionist. 2. Book an Appointment for the Patient from E2E-01. 3. Link a Service (e.g., "Massage") and Room.	Appointment is saved. Status is "SCHEDULED". Audit Log records the creation.	PASS
E2E-03	Inventory Consumption	1. Login as Physiotherapist. 2. Create an OUTBOUND Transaction for "Massage Oil" (Qty: 1).	System accepts transaction. Item stock decreases by 1 automatically.	PASS
E2E-04	Invoice Generation	1. Create a new Invoice linked to the Appointment from E2E-02. 2. Do **not** manually add items. Click Save.	System automatically creates an InvoiceItem based on the Appointment's Service and Price. Total is calculated.	PASS

Test ID	Scenario	Steps to Execute	Expected Result	Status
E2E-05	Payment & Status	1. Create a Payment for the full amount of the Invoice. 2. Check Invoice Status.	Invoice Status updates automatically from DRAFT to PAID.	PASS

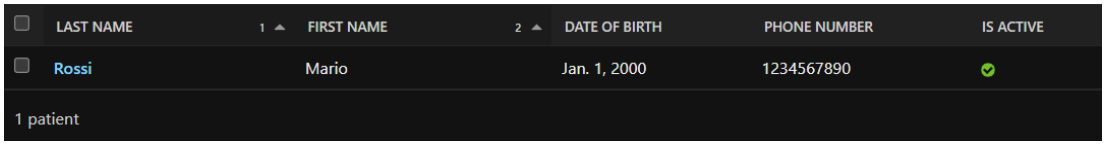


Figure 11: Result of E2E-01

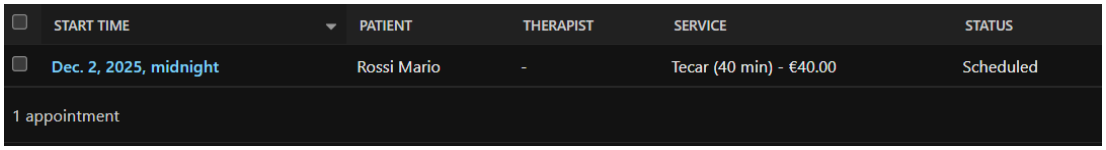


Figure 12: Result of E2E-02

Test Suite B - Module Constraint & Stress Tests

Objective: Verify that the system validation logic prevents invalid data and enforces business rules.

Table 2: Scheduling Module Constraint Tests. Captures time validation, therapist filtering, and access control privacy.

Test ID	Scenario	Steps to Execute	Expected Result	Status
SCH-01	Time Travel Check	Attempt to create an appointment where end_time is earlier than start_time.	Validation Error: API/Admin rejects the save operation.	PASS
SCH-02	Therapist Filtering	In the Appointment form, try to select a User with role='PATIENT' as the therapist.	The User is **not visible** in the dropdown (filtered by limit_choices_to).	PASS
SCH-03	Access Control	Login as a "Physiotherapist" and try to view appointments assigned to **another** physio.	403 Forbidden or Empty List (Data privacy enforced).	PASS

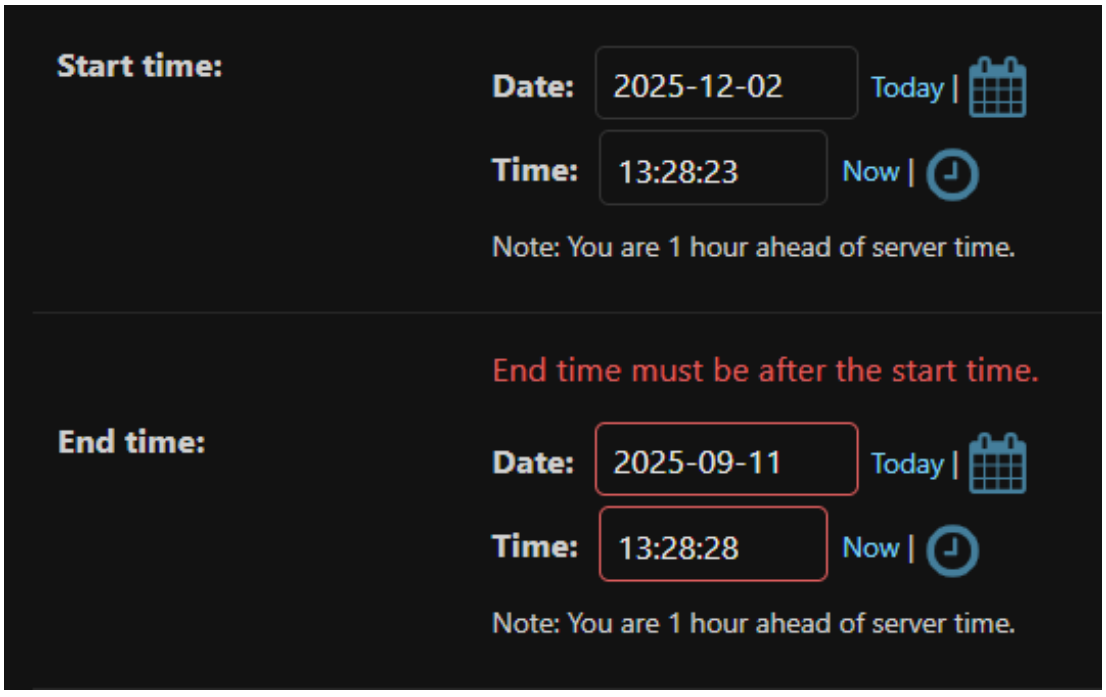


Figure 13: Result of SCH-01

Table 3: Inventory Module Constraint Tests. Captures negative stock protection, atomic updates, and transaction read-only status.

Test ID	Scenario	Steps to Execute	Expected Result	Status
INV-01	Negative Stock Protection	1. Select an Item with current_stock = 5. 2. Attempt to create an OUTBOUND transaction for quantity = 10.	Validation Error: "Insufficient stock. You have 5 but tried to remove 10."	PASS
INV-02	Atomic Updates	Create a valid transaction. Immediately check the Item's stock.	Stock is updated instantly. Transaction is recorded in history.	PASS
INV-03	Transaction Immutability	Attempt to edit the quantity of an existing transaction in Admin.	Fields are **Read-Only**. Historical data cannot be altered.	PASS

Table 4: Billing Module Constraint Tests. Captures auto-pricing, partial payment logic, and patient mismatch validation.

Test ID	Scenario	Steps to Execute	Expected Result	Status
BIL-01	Data Consistency	Create an Invoice Item without specifying a price.	System auto-fills the price from the associated Service.	PASS
BIL-02	Partial Payment	1. Create Invoice for €100. 2. Add Payment of €50.	Invoice Status updates to PARTIAL.	PASS
BIL-03	Patient Mismatch	Attempt to link an Invoice for Patient A to an Appointment belonging to Patient B.	Validation Error: "Mismatch! This appointment belongs to Patient B."	PASS

Conclusions

The backend system successfully passes all core functional and constraint tests. The architecture properly enforces data integrity and automates critical business workflows (stock management, billing status). The system is ready for Frontend or AI integration.

AI Integration

Since I don't have thousands of historical records to train a complex Neural Network yet, I implemented a Heuristic AI, defined as a Rule-Based Model. This is a standard starting point for MVPs: it calculates a *Risk Probability* based on the patient's actual past behavior. At this point I present the API endpoints reguarding this integrations I've done.

The following endpoints provide advanced insights, utilizing basic AI algorithms for risk assessment and demand forecasting, as well as a virtual assistant interface.

GET /api/analytics/dashboard/

Description: Admin Dashboard endpoint. Retrieves real-time Key Performance Indicators (KPIs) to populate the main administrative view.

Access Level: Admin Only
Response Data:

- `total_revenue`: Current financial aggregate.
- `active_patients`: Count of currently registered patients.
- `pending_invoices`: Count of unpaid invoices.

GET /api/analytics/risk/{id}/

Description: AI-driven No-Show Predictor. Calculates a risk score for a specific patient based on their appointment history (e.g., previous cancellations or missed visits).

Access Level: Staff / Physiotherapist
Response: `risk_score` (Integer 0-100%) indicating the likelihood of missing the next appointment.

GET /api/analytics/forecast/

Description: AI Demand Forecasting. Uses a moving-average algorithm on historical data to predict appointment volume for the upcoming 7 days.

Access Level: Staff / Physiotherapist
Filter: Supports filtering predictions by an admin (general load) or the currently logged-in physiotherapist to see personal load.

POST `/api/analytics/chat/`

Description: Virtual Assistant interface. A rule-based chatbot endpoint that parses natural language queries (e.g., "what are the opening hours?", "price list") and returns structured responses.

Access Level: Public
Input Body: `message` (string)
Response: `reply_text` (string), `action_suggestion` (optional).

However this may be intended as some advanced work, it just represents a starting point for more complex AI logics, for next iterations or future versions.

Notifications and Reminders

One of the work packages imposed by the client was the ability to send email notifications to patients once the appointment is booked. After asking for permission to send email, the system will guarantee these main two features: for a first version of the application, it guarantees emails for scheduling confirmations for what concerns appointments. Eventually, the system will send a reminder around 24 hours before the appointment to remember the patient related to it to attend it. For this purpose I will implement the Celery Worker and Celery Beat systems.

To run (in development environment), it's important to first use the Celery Worker (`celery -A clinic_backend worker -l info -P gevent`), that will take care of sending confirmation emails and reminders for the appointments.
Then, the system will need Celery Beat (`celery -A clinic_backend beat -l info`), which will silently trigger the tasks if they satifsy the requirements.

Frontend Integration

I am now entering Weeks 11-14, which are about developing the frontend for the system I've built until now.

This is where the project stops being code in a terminal and becomes a real product that people can click and use. This phase involves building a Single Page Application (SPA) using React.js that consumes the Django REST API.

Architecture and Data Flow

The frontend is built on a Component-Based architecture, styled with Tailwind CSS for rapid and consistent UI design. The application communicates with the backend via asynchronous HTTP requests.

A key performance feature implemented during this phase is Parallel Data Fetching. On the Staff Dashboard, the application leverages `Promise.all` (or concurrent hooks) to simultaneously retrieve:

- KPI Statistics (total revenue, appointment counts, active patients);
- Appointment Forecasts;

- Recent Activity Logs;

This ensures that the dashboard renders immediately with a complete data set, rather than loading components sequentially.

Key Implementation Details

Several specific functional requirements were addressed during the frontend integration:

Mobile Responsiveness Strategy

Recognizing the different usage patterns of our user base, a selective responsiveness strategy was adopted:

- **Patient Portal (Mobile-First):** The "Home" and "Profile Details" pages are fully optimized for mobile devices. This allows patients to book appointments, view their history, and access tele-consultation links via smartphones. The navigation uses a hamburger menu on small screens and adapts grid layouts to vertical stacks.
- **Staff Portal (Desktop-Optimized):** The administration dashboard and calendar views remain desktop-focused to accommodate complex data tables and scheduling grids that require larger screen real estate (Figure 14).

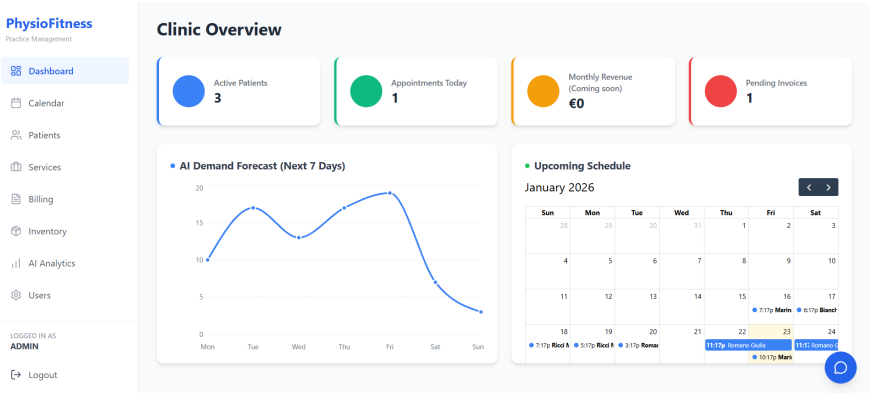


Figure 14: Staff Portal, which is optimized for desktop usage,

Soft Deletes and Data Integrity

To maintain historical data integrity, the system implements "Soft Deletes" via an `is_active` boolean flag (Figure 15).

- **Logic:** When a staff member "deletes" a patient, the record is not removed from the PostgreSQL database but simply marked as inactive.
- **UI Behavior:** Inactive patients are visually greyed out in the registry list. Crucially, the appointment booking form filters the patient dropdown to exclude inactive users, preventing new bookings for deactivated profiles while preserving past appointment records for financial reporting.

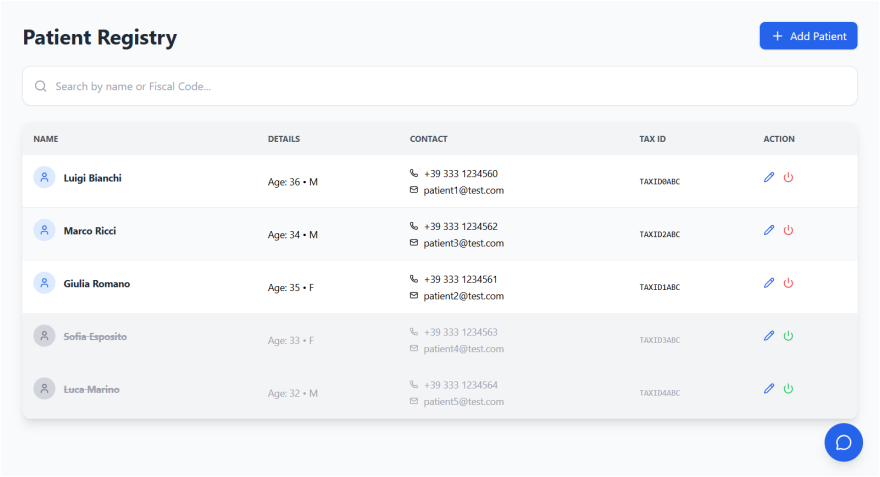


Figure 15: Patient List page, showing soft deletes in UI.

Integration Testing

Integration testing focused on the security and stability of the API communication. Key test scenarios included:

- **JWT Authentication Flow:** Verifying that the frontend correctly attaches the Bearer token to headers and handles 401 Unauthorized errors by redirecting to the login page.
- **Error Handling:** Ensuring the frontend correctly displays validation errors sent by the Django serializer (e.g., preventing the creation of a duplicate patient with an existing email).

System Architecture and Technical Specifications

The final system follows a decoupled Client-Server architecture, ensuring scalability and separation of concerns.

- **Frontend (Client):** Built with **React.js**. State management is handled via React Hooks (useState, useContext) to manage user sessions and real-time data updates. The UI uses **Recharts** for visualizing analytics data.
- **Backend (Server):** Developed using the Django REST Framework (DRF). It exposes a RESTful API that communicates with the frontend via JSON. Authentication is secured using SimpleJWT, ensuring stateless and secure user sessions.
- **Database:** PostgreSQL was chosen for its robustness in handling relational data. The schema is normalized to Third Normal Form (3NF) to reduce redundancy.
- **AI Module:** A chatbot assistant manages simple user requests. Plus, a dedicated service layer (`apps.analytics.ai`) handles predictive modeling. Currently, this uses a *Heuristic Rule-Based Engine* to calculate Patient Risk Scores in real-time based on demographics and historical behavior. *Note: In a production environment with sufficient historical data, this module is designed to be replaced*

by a Linear Regression or Time-Series model without altering the frontend interface.

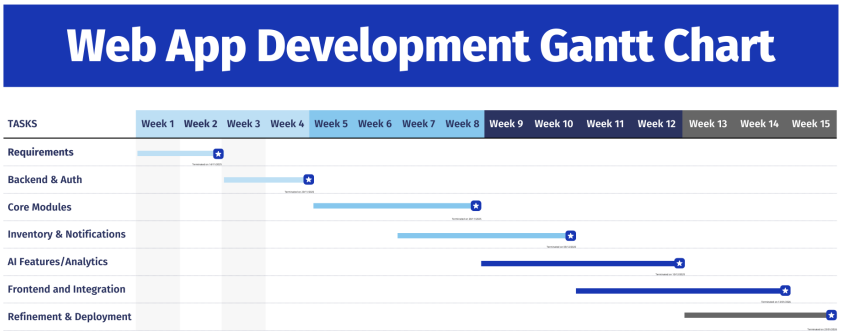


Figure 16: Gantt chart at the end of the project.

Maintenance and Administration

Audit Logging

For security and accountability, the system tracks critical data changes (User creation, Patient updates, Deletions).

- **Access:** These logs are read-only and accessible strictly via the Django Admin interface at `http://127.0.0.1:8000/admin`(Figure 17).
- **Security:** Logs cannot be edited or deleted through the interface, ensuring a tamper-proof audit trail.

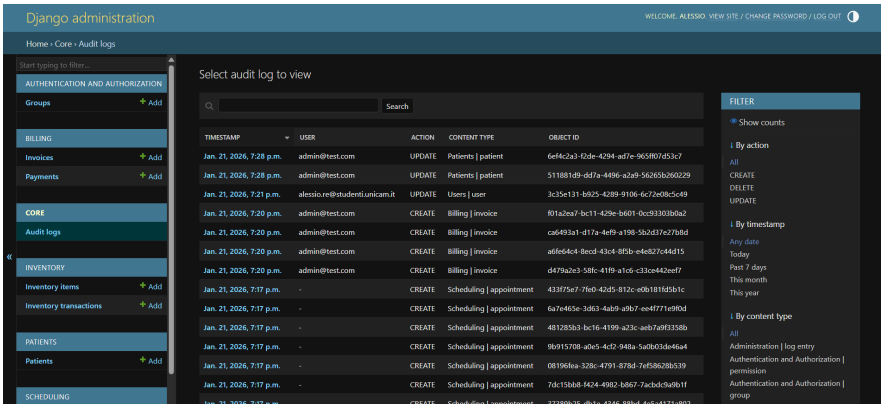


Figure 17: Audit logs table, read only.

Conclusion and Future Works

This project successfully demonstrates a full-stack physiotherapy management system that digitizes the core workflows of a clinic. By separating the frontend and backend, the system is ready for independent scaling.

Future Iterations

Several features were mocked in the final UI to demonstrate future extensibility (Figure 18):

- **Tele-Consultation:** A "Join Video Call" button is implemented for confirmed appointments. Future work would involve integrating **WebRTC** to enable real-time video communication.
- **E-Invoicing:** A "Download Invoice" action is present in the UI. A future iteration would integrate this with the Italian SDI (Sistema di Interscambio) for automated tax compliance and XML invoice generation.

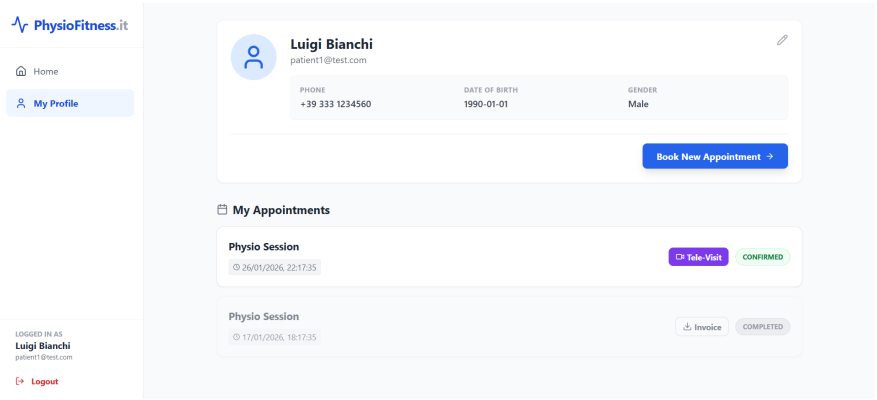


Figure 18: Patients profile page, implementing tele-consultation and e-invoicing.

It is important to notice that some features are still in development mode to make them more easily readable and understandable.

References

1. PhysioLB - Clinic Management Features. Available at: https://www.physiolb.com/
2. Physio-Clinic Italy - Services Structure. Available at: https://physio-clinic.it/#servizi
3. Paideia Hospital - Online Booking Flows. Available at: https://www.paideiahospital.it/en/book-online/