Detailed Designed Justification

Why did you structure your classes this way?

At a high level, we organized the project into **layers** that roughly follow an MVC-style architecture:

- **UI Layer:** FXML views + controllers packages
- **Core / application layer:** the *app* and *core* packages that start the app and manage global state + navigation
- **Domain & business logic layer**: the *users*, *appointments*, and *database_management* packages that define our main entities and services that operate on them
- **Infrastructure / utilities:** *authentication, utils,* and *testing*

This separation keeps responsibilities clear such as: *controllers* are focused on interaction with the user, *services* handle data and business logic, and core classes deal with global concerns like navigation and logged-in state.

app/

The purpose of the app package is that its the entry point of the application. Its responsibility is to simply start JavaFX and handle control over to the rest of the system

Main

This class is the launcher for the application. It extends Application because this is required JavaFX structure, and its only responsibility is to start JavaFX and hand over control to the rest of the system.

appointment/

The appointment package is dedicated to the domain model representing appointments between doctors and patients. We place the Appointment class in its own package to emphasize that appointments are a core part of the system's business logic and should remain separate from other packages.

Appointment

The Appointment class represents a single scheduled interaction between doctor and a patient. It stores essential information for an appointment, such as the doctor, the patient (when applicable), the date and time, the status and any notes, so the rest of the application can work with a clear model.

authentication/

The authentication package contains logic responsible for handling user login and account creation. We separate authentication into its own package to keep credential handling, identity verification, and sign up flows independent from other packages. This keeps the security-related operations centralised and prevents controllers from directly managing user lookup or password validation

Authentication

The Authentication class provides the core operations for logging in existing users and signing up patients or doctors. It communicates with UserService to retrieve users and save newly created accounts while keeping the controller layer free from database or lookup logic.

controllers/

The controllers package contains all JavaFX controller classes that act as the bridge between FXML views and all the other underlying services and core logic. We grouped them in a single package to keep UI-related behavior in one place.

BaseController

BaseController is the common superclass for all the other controllers. It holds shared references to ScreenManager and the singleton AppState, so every controller cna navigate between screens and access the current application state without duplicating this setup code. This allows us to avoid repetition and ensure controllers follow the same pattern for navigation and state access.

DoctorPageController

DoctorPageController is the controller for the doctor landing page. It extends the base controller so it can use the shared ScreenManger and AppState as described above. It initialises the table columns, loads the doctor’s appointments from DoctorService and AppointmentService, and applies filters such as data range or patient name so that all filtering logic stays in one place. It also coordinates user actions on appointments (creating available slots, cancelling, deleting, and making them available again), always delegating persistence to the service layer rather than talking to the database.

LoginController

LoginController is responsible for the login screen and the first step of the user flow. It extends BaseController to reuse navigation and shared state, and it connects email/password fields in the FXML with the underlying authentication logic. When the user submits the form, the controller validates the inputs, calls the Authentication class to verify credentials, and then stores the authentication User in AppState so the rest of the app can access them in the current session. Depending on the user specialization, it routes them either to patient or doctor landing page using screenManager. It also uses MessageUtils to show success or error messages.

ModifyAccountController

This class manages the screen where a logged-in user can update their account details. It extends BaseController and it focuses on validating the form inputs and deciding where to send the user next. The controller reads the email and password fields (the fields where users update their email or password), performs basic checks such as non-empty email, and matching passwords, and then will delegate actual updates to the database. Once accepted, it alerts the user of success or failure.

PatientPageController

This class is responsible for showing patients their appointments and appointments available for them to book. It uses AppState to know which patient is logged in, configures table columns to show their own appointments including other information such as dates, time, doctor, information, notes, etc. The controller also wires up filtering controls and passes those to the PatientService under database_management. In addition, it allows patients to book available appointments then registering their id to the appointment, then SCHEDULING that appointment in the process.

SignUpController

The controller manages the registration screen for users. Instead of having two sign-up flows, the controller combines them into one screen and two forms and toggles between them. It validates all user inputs (required fields, email, password, and specialization if you're a doctor) before delegating account creation to the Authentication class. After successful sign up, it stores the newly created user in AppState and navigates to the correct landing page.

core/

The core package contains the foundational infrastructure that the rest of the application depends on. These classes provide global behavior that all controllers and screens use, such as navigation between views and keeping track who is logged in. By placing this logic in its own package, we avoid scattering global state and navigation code across multiple controllers.

AppState

AppState is a small singleton class that stores the user currently logged into the application. Instead of passing the user object or ID through every controller or screen, AppState provides a central place where that information can be retrieved during the session. This simplifies communication between different parts of the application and avoids creating unnecessary dependencies between controllers. The singleton design ensures that there is only one source of truth for session data at any time.

ScreenManager

Screen Manager is responsible for loading and displaying FXML screens. It centralizes all scene-switching logic so that controllers never manipulate JavaFX stage or worry about how

to load FXML files. When a screen is shown, ScreenManager also inputs itself into the controller so the controller can later request navigation.This design keeps navigation consistent throughout the application, reduces duplicated code, and maintains a clean separation between UI behavior and UI rendering.

database_management/

The database_management package constrains all service classes that talk to our backend database, Supabase. Its purpose is to isolate data-access and query logic from the controllers and domain models so that the rest of the application works with high-level operations like “fetch appointments for this doctor” for example. Like this, we keep controllers thin, make it easier to change how we store data in the future, and avoid scattering database details through the project.

AppointmentService

This is responsible for all generic appointment-related persistence. It builds queries for Supabase based on filters, converts the JSON response to Appointment objects, and exposes methods to fetch by ID, by user role, update an existing appointment, and many more.

DoctorService

DoctorService provides operations that are specific to doctors. It uses SupabaseClient and, where convenient, AppointmentService under the hood. For example, it allows doctors to create new available appointment slots, fetch their available slots, resolve patient IDs from a full name, and retrieve appointments filtered by date range and patient name.

PatientService

This focuses on appointment operations that are specific to patients. This accepts filter parameters such as date range, doctor id, and status, then translates those into a map of SupaBase query conditions and delegates actual fetch to AppointmentService.

SupabaseClient

SupabaseClient is the low-level HTTP client used by all services in the database_management package. It centralizes the base URL, API key, and HTTP request construction for GET, POST, PATCH, and DELETE operations, so we do not repeat connection and header logic in every service.

UserService

This class encapsulates all persistence logic related to users. It provides functions to fetch a user by email or ID, retrieve doctors by specialisation, or any other function related to users

in general (not specific to doctors or patients). This separation means controllers and authentication logic only deal with user objects and role concepts, while all the details of JSON parsing and Supabase endpoints remain in Userservice.

users/

The users package constrains the core domain model for people in the system. We grouped User, Doctor, and Patient here because they represent main actors that interact with the app. Keeping them together, and separated from controllers and services, makes it clear that these classes describe *who* is using the system.

User

User is an abstract base class that defines the common state behavior for doctors and patients like id, email, name, password, and specialization. Some setters, such as setEmail and setPassword, immediately call UserService.updateUser, which keeps the persisted user data in sync whenever key fields are changed.

Doctor

Doctor extends User and adds behavior specific to doctors, like creating available appointment slots, marking appointments as completed, and retrieving their appointments with optional filters. It allows persistence to DoctorService and AppointmentService, but the methods themselves are written in terms of domain operations that make sense from a doctor's perspective.

Patient

Patient, like doctor, is also extending User but focuses on patient user actions like finding and booking available appointments, and cancelling once reserved. These behaviors reflect real patient interactions and keep role-specific logic inside the patient model. This allows the rest of the application to work with simple, high-level operations directly on the patient object, keeping responsibilities clear and consistent with the overall domain design.

utils/

The utils package contains small helper classes that provide reusable functionality across the application. These utilities are not necessarily tied to any specific feature, so grouping them here keeps the rest of the codebase cleaner and avoids repeating common logic in multiple places.

JsonTransformer

JsonTransformer handles converting between JSON data returned by Supabase and the application's user objects. It centralizes both directions of this conversion, creating User

subclasses from JSON and generating JSON bodies for inserts and updates, so that controllers and services don’t need to handle any parsing details.

MessageUtils

MessageUtils provides simple helper methods for displaying error and success messages in JavaFX text elements. It keeps styling and visibility logic in one place so controllers can show or clear messages with a single clean call.

Why did you choose your specific data structures for a task?

Domain Objects (classes)

Appointment, User, Doctor, and Patient are implemented as Java classes. Reason: object-oriented domain modeling keeps related data and behavior together (fields like id, doctor, patient, date, status, and methods such as bookAppointment or cancelAppointment), which makes business logic (e.g., changing status, assigning patients) explicit, type-safe, and easy to maintain.

Lists / ArrayList (List<Appointment>)

Service methods return List<Appointment> and use new ArrayList<>() to collect results parsed from JSON arrays. Reason: ordered collection semantics are useful for result sets, iteration, and streaming; ArrayList gives simple, efficient random access and appending for the typical use case of accumulating query results.

Maps / HashMap (Map<String,Object> filters)

Filter building uses Map<String,Object> (backed by HashMap) to assemble dynamic query parameters. Reason: maps are a natural fit for named query parameters because they allow flexible, key-based lookup and easy iteration when building query strings for the Supabase REST endpoints.

JSON types (Gson: JsonObject/JsonArray/JsonNull)

The code uses Gson JsonObject/JsonArray/JsonNull to parse responses and compose request bodies. Reason: explicit JSON objects give precise control over fields, nullability, and serialization details when interacting with an external REST/JSON DB (Supabase), avoiding lossy string manipulation and making patch/post payloads clear.

Date/time (LocalDateTime + DateTimeFormatter)

Dates are represented with LocalDateTime and parsed/serialized with DateTimeFormatter.ISO_LOCAL_DATE_TIME. Reason: the java.time API provides strong typing and correct temporal semantics for comparisons, filtering (gte/lte), and formatting to/from the DB's ISO timestamp strings.

How does your application connect to and query the database?

Connection Layer

The app uses a simple HTTP client wrapper SupabaseClient to talk to the Supabase REST API: a single HttpClient instance issues synchronous client.send(...) requests to a base URL with the project API key sent in apikey and Authorization: Bearer headers. SupabaseClient exposes convenience methods get, post, patch, delete, and deleteByFilter that construct URIs (including Supabase filter query syntax like ?id=eq.5), attach JSON headers, and return the raw JSON response body.

Query Composition

Higher-level services build query endpoints as strings (using Map<String,Object> filters or StringBuilder) and call SupabaseClient.get(...) or other methods. For example, AppointmentService assembles queries from a Map<String,Object> of filters (doctor_id, date_time, status, etc.) and DoctorService builds filtered endpoints (including in.(...) lists) when searching by specialization or patient name.

Serialization / Parsing

Responses are parsed with Gson: services call JsonParser.parseString(...) to get JsonArray/JsonObject and extract fields. For writes/updates the code constructs JsonObject payloads (using JsonNull.INSTANCE for SQL NULL) and uses SupabaseClient.post/patch with Prefer: return=representation so the API returns the inserted/updated rows for reading back generated IDs.

Mapping to Domain Objects

After parsing JSON, services convert rows into typed domain objects (Appointment, Doctor, Patient, User) by reading primitive fields and invoking constructors (e.g., new Appointment(id, doctorId, patientId, dateTime, status, notes)), and by using JsonTransformer.jsonToUser(...) for user conversions. Service methods return typed collections (List<Appointment>, List<Doctor>, etc.) built with ArrayList for ordered, iterable result sets.

Date & Null Handling

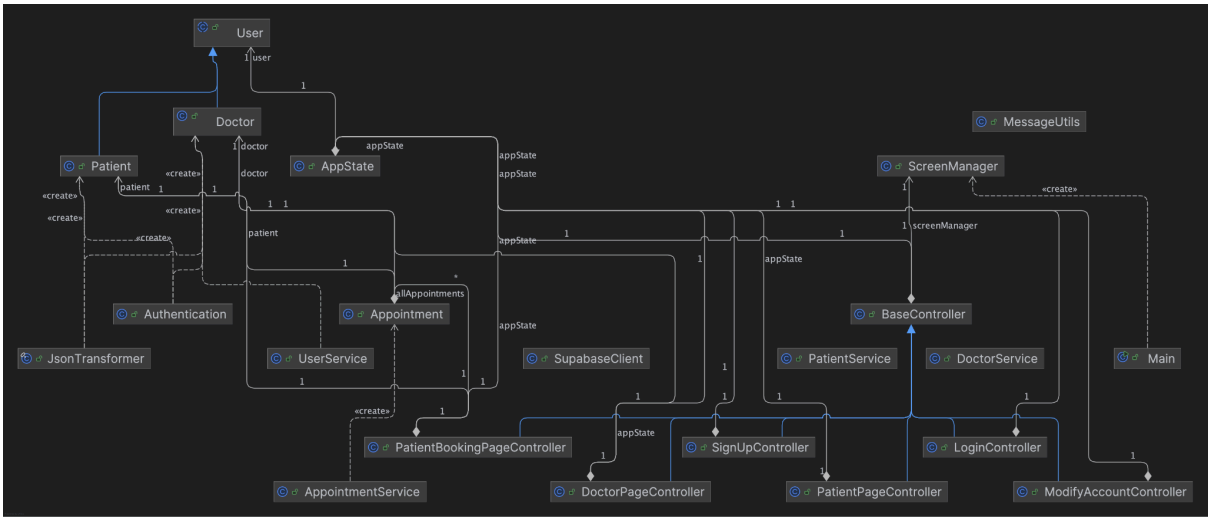
Timestamps use LocalDateTime with DateTimeFormatter.ISO_LOCAL_DATE_TIME for parsing/serializing so DB timestamps map cleanly to Java time types. Missing relations use patient_id as null (represented in JSON as JsonNull.INSTANCE) and the code sometimes uses -1 briefly when parsing before resolving to null/object references.

What was the most significant design challenge you faced and how did you solve it?

As the project grew, code and responsibilities started to mix: UI logic, data access, JSON parsing, and domain rules were scattered across files, which made bugs harder to find, tests harder to write, and parallel work riskier because changes in one area could unexpectedly break another.

We fixed this by clearly separating responsibilities into packages and small focused classes: controllers handle UI flow, domain classes (User, Doctor, Patient, Appointment) hold business state and behavior, service classes (AppointmentService, UserService, DoctorService, SupabaseClient) centralize all HTTP/DB interactions and JSON mapping, and utilities handle cross-cutting tasks like transformation and messaging. This structure reduced coupling, made dependencies obvious, simplified debugging and testing, and let different team members work on UI, services, or models independently without stepping on each other’s changes.

UML Class Diagram (Simplified)



For a more detailed version please refer to the java.png image in the repository.

Database Schema