HealthHub: A Healthcare Data Management System

Paolo Palumbo, Francesco Panattoni, Nedal Hadam

June 18, 2025

Contents

Chapter 1

Introduction

Healthhub is a web-based platform designed to simplify and centralize the management of medical appointments. It offers dedicated features for both patients and healthcare professionals. Regular users can search for doctors, book visits, and leave reviews, while medical professionals can manage the services they offer and track their appointments.

- Allow users to search for medical professionals and retrieve information on their services
- Allow users to book appointments and review doctors after a visit
- Allow users to manage their appointment history and cancel or modify bookings
- Give users recommendations on doctors based on their location and past activity
- Allow doctors to specify their specializations, availability, and offered services
- Allow doctors to view and manage their upcoming appointments

Chapter 2

Dataset and Web Scraping

To populate the dataset for our application, we used a combination of web scraping, synthetic data generation, and logical inference based on real-world patterns of user behavior.

2.1 Web Scraping

The initial data was scraped from a platform containing medical reviews, MioDottore[1]. This platform provides a comprehensive database of medical professionals, including their specializations, locations, and user reviews. The scraping process involved extracting the following information:

- Doctor names and specializations
- Locations (cities and regions)
- User reviews and ratings
- Contact information (where available)
- Service offerings (e.g., types of consultations, treatments)

2.2 Synthetic Data Generation

To enhance the dataset, we generated synthetic data to simulate a more realistic user base and appointment history. This involved several steps:

- **Unique Users**: We extracted all unique usernames from the scraped reviews. For each user, we generated a detailed profile including demographic and geographical information.
- **Appointments**: For every review, we created a corresponding medical appointment. These were dated in the weeks preceding the review, to simulate a realistic flow where users leave feedback shortly after being treated.

[1] https://www.miodottore.it/

- **Likes**: To simulate engagement features such as likes on reviews, we analyzed the provinces of the doctors each user had interacted with. We then randomly selected other doctors in the same provinces and associated a number of likes comparable to the number of reviews written by each user.

2.3 Dataset Characteristics: Velocity and Variety

2.3.1 Velocity

The application context is inherently dynamic, and our dataset reflects this through the following characteristics:

- **High review frequency**: The dataset reflects an average of at least 100 new reviews per day, simulating the continuous flow of patient feedback that a live system would experience.
- **Growing doctor base**: Based on trends from medical boards, we estimate around 400 to 500 new doctors would register on the platform annually, contributing to the system's dynamism and evolving content.

2.3.2 Variety

The dataset incorporates diverse data types, which contribute to its heterogeneity:

- **Multi-format records**: Structured user and doctor profiles, unstructured textual reviews, and timestamped interaction logs (likes, appointments) are all represented.
- **Diverse entities**: The inclusion of users, doctors, reviews, likes, and appointments enables multi-relational analysis and feature richness.

2.4 Resulting Dataset

The original scraped dataset, stored in `scraped.json` (265 MB), contains the raw information collected from the web. This dataset served as the foundation for the data generation process, providing:

- 699,987 reviews
- 214,682 unique reviewers
- 87,632 doctors

Building upon this, the final dataset is stored in JSON format, totaling approximately 960 MB, and includes both real and synthetic data entries. Its main components are:

- **Doctors** (`doctors.json`, 289 MB) Contains 87,632 healthcare professionals, each with profile data and linked to 699,987 reviews. Reviews include timestamps and patient feedback.
- **Users** (`users.json`, 66 MB) Comprises 214,682 unique users extracted from the original dataset and extended with synthetic profiles.
- **Appointments** (`appointments.json`, 422 MB) Each review is connected to a synthetic appointment, scheduled in the weeks preceding the review date. Appointments simulate realistic scheduling and clinic visit patterns.
- **Templates** (`templates.json`, 162 MB) Stores template structures for appointment scheduling logic, such as available time slots, weekdays, and timing constraints.
- **User Likes** (`user_likes.json`, 22 MB) Represents user interactions with doctors, generated according to the geographic distribution of the doctors reviewed by each user.

Overall, this dataset offers a realistic simulation of user and doctor activity on a healthcare review platform, reflecting both volume and behavioral complexity.

Chapter 3

Design

3.1 Actors

The application involves three primary categories of actors, each with distinct roles and permissions:

- *Non-Authenticated User (Guest User):* refers to an anonymous individual who accesses the application without logging in. This actor is allowed to register, authenticate using existing credentials, or explore the platform by searching for doctors and viewing their public profiles.
- *Patient (User):* represents the end-user of the service. Once authenticated, the patient can book appointments with doctors, confirm or cancel them, and subsequently provide feedback through reviews.
- *Doctor:* a professional figure who offers medical appointments. The doctor can manage their availability schedule, making it accessible to patients, and consult an analytics dashboard displaying aggregated data such as earnings, reviews, and the number of visits performed.

3.2 Functional Requirements

The following section outlines the functional requirements.

3.2.1 Guest Users

Guest users are allowed to:

- Register for an account within the application;
- Log in using their credentials;
- Initiate the password recovery process in case of forgotten credentials;
- Search for doctors and access their public profiles.

3.2.2 Patients

Authenticated users identified as **patients** are granted the following capabilities:

- Manage and update their personal profiles;
- Search for doctors and access their public profiles;
- Book appointments and optionally include a note for the doctor;
- Endorse a doctor as a form of support or recommendation;
- Submit reviews regarding their medical experience;
- View their appointments, including past, current, and upcoming ones;
- Search for doctors through the recommendation feature provided by the system.

3.2.3 Doctors

Authenticated users identified as **doctors** are provided with the following functionalities:

- Manage and update their personal and professional profiles;
- Search for other doctors and access their public profiles;
- Define and configure the types of visits offered, along with the associated pricing;
- Oversee appointments by identifying scheduled patients and appointments for the current day;
- Configure and manage appointment availability through customizable scheduling templates;
- Monitor and respond to patient reviews;
- Access analytics, including visual representations of revenue, newly acquired patients, recent reviews, and a summary of completed visits.

3.3 Non-Functional Requirements

The following non-functional requirements address the quality attributes of the system, ensuring its performance, reliability, security, scalability, and maintainability.

3.3.1 Performance and Scalability

- The system shall maintain **acceptable response times** for common operations such as viewing medical records and scheduling appointments;
- The system shall efficiently **handle increased workload** during peak usage periods without degradation of service.

3.3.2 Availability and Reliability

- The application shall be **available** to all users **24/7**, minimizing downtime through redundancy and failover mechanisms;
- The system shall implement **backup and recovery procedures** to preserve data integrity and support rapid restoration after failures.
- The system shall tolerate occasional data staleness in non-critical views, ensuring high availability even under degraded conditions.

3.3.3 Security and Privacy

- The system shall enforce **secure, authenticated access for all users**, employing strong password policies and session management;
- All sensitive data shall be encrypted both in transit and at rest;
- The system provides defenses against injections.

3.3.4 Usability

- The user interface shall be intuitive and **user-friendly**, enabling users to perform tasks with minimal learning curve.
- The application shall exhibit **low latency** in user interactions to maintain a responsive experience.

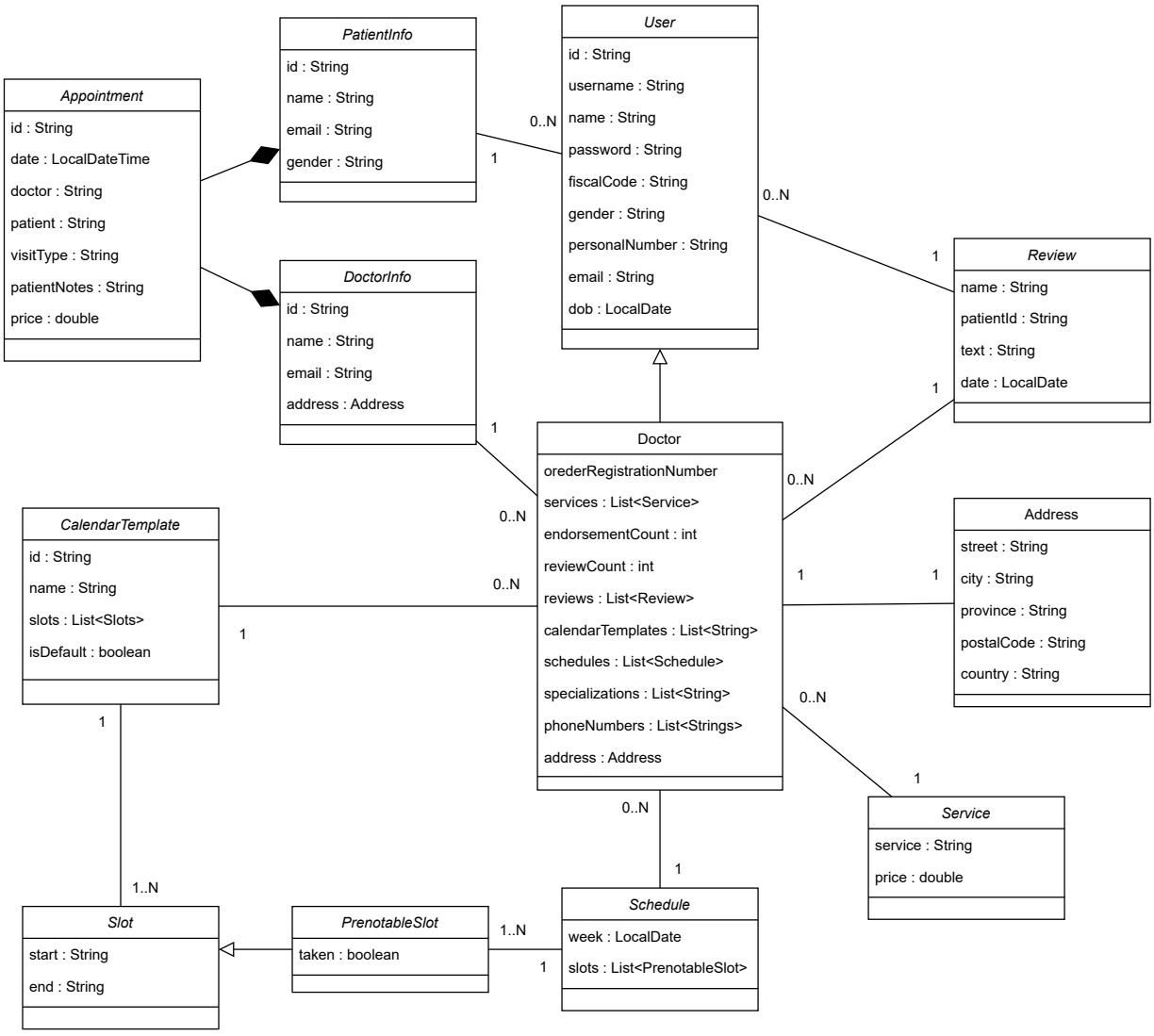
3.3.5 Portability and Flexibility

- The application shall be **deployable on multiple operating systems** (e.g., Windows, macOS, Linux) without requiring behavioral changes.
- The system's architecture shall support the addition of new attributes or modules (e.g., new appointment types) with minimal code modification.

3.3.6 Maintainability

- The codebase shall follow **Object-Oriented design** principles, promoting modularity and ease of comprehension.
- The system shall **minimize single points of failure** through component decoupling and redundancy.
- The application shall include comprehensive documentation and code comments to facilitate future enhancements and debugging.

3.4 UML Class Diagram



3.4.1 Relationships between classes

Here we briefly summarise the relation between each class.

- **Doctor** is a specialization of **User**;
- A **Doctor** may offer zero or more **Service** instances and each **Service** is provided by exactly one **Doctor**;
- A **Doctor** may have zero or more **Review** entries and each **Review** refers to exactly one **Doctor**;
- A **Doctor** may define zero or more **CalendarTemplates** and each **CalendarTemplate** belongs to exactly one **Doctor**;
- A **Doctor** may publish zero or more **Schedules** and each **Schedule** is associated with exactly one **Doctor**;
- Each **Schedule** comprises zero or more **PrenotableSlots** and each **PrenotableSlot** is part of exactly one **Schedule**;
- Each **CalendarTemplate** comprises zero or more **Slots** and each **Slot** belongs to exactly one **CalendarTemplate**;
- A **Doctor** has exactly one **Address** and each **Address** is linked to exactly one **Doctor** (and similarly to **DoctorInfo**);
- An **Appointment** is associated with exactly one **DoctorInfo** and one **PatientInfo** and each **DoctorInfo** or **PatientInfo** may have zero or more **Appointments**.

3.4.2 Design of the classes

Here we briefly summarise the primary attributes of each entity.

User

- `id: String` – unique identifier
- `username: String`
- `name: String`
- `password: String` – stored as hash
- `fiscalCode: String`
- `gender: String`
- `personalNumber: String`
- `email: String`
- `dob: LocalDate`

Doctor (specialization of **User**)

- `orderRegistrationNumber: String`
- `services: List<Service>`
- `endorsementCount: int`
- `reviewCount: int`
- `reviews: List<Review>`
- `calendarTemplates: List<CalendarTemplate>`
- `schedules: List<Schedule>`
- `specializations: List<String>`
- `phoneNumbers: List<String>`
- `address: Address`

Address

- `street: String`
- `city: String`
- `province: String`
- `postalCode: String`
- `country: String`

Appointment

- `id: String`
- `date: LocalDateTime`
- `visitType: String`
- `patientNotes: String`
- `price: double`

Schedule

- `week: LocalDate`
- `slots: List<PrenotableSlot>`

CalendarTemplate

- `id: String`
- `name: String`
- `slots: List<Slot>`
- `isDefault: boolean`

Slot

- `start: String`
- `end: String`

PrenotableSlot (subclass of **Slot**)

- `taken: boolean`

Review

- `name: String`
- `patientId: String`
- `text: String`
- `date: LocalDate`

Service

- `service: String`
- `price: double`

PatientInfo (belong to **Appointment**)

- `id: String`
- `name: String`
- `email: String`
- `gender: String`

DoctorInfo (belong to **Appointment**)

- `id: String`
- `name: String`
- `email: String`
- `address: Address`

Chapter 4

Implementation

4.1 Front-End

The front-end of our application has been developed as a traditional web-based interface, utilizing a combination of **JavaScript**, **HTML** and **CSS**. This technology stack enables the creation of a responsive and interactive user interface that can be accessed from any modern web browser without the need for additional installations. The application communicates asynchronously with the back-end server through **AJAX** (Asynchronous JavaScript and XML) requests, allowing data to be fetched and updated dynamically without requiring full page reloads. This results in a smoother user experience and improved performance, particularly when handling operations such as appointment booking, profile management, and real-time data visualization. The separation of concerns between presentation and logic also enhances the maintainability and scalability of the application.

4.2 Back-End

The back-end is implemented using **Java** due to its robustness, platform independence, and strong ecosystem. Java offers a well-established set of libraries and frameworks that support rapid development of secure, scalable, and maintainable web applications. Its object-oriented nature encourages clean architecture and modular design, which are essential for complex systems like healthcare platforms. Additionally, Java's widespread use in enterprise environments ensures long-term support and community-driven innovation, making it a reliable choice for production-grade applications.

4.3 Spring Boot

As the core of our application, we adopted the **Spring Boot** framework, which has significantly streamlined both the development and deployment processes. One of the primary advantages of Spring Boot lies in its built-in support for an embedded **Tomcat** server, allowing the creation of a fully functional web application without requiring external configuration. This design choice has enhanced our ability to construct responsive and maintainable

RESTful APIs.

In addition, to support our data management needs, we integrated **Spring Data Neo4j** and **Spring Data MongoDB**. The former facilitates interaction with the **Neo4j graph database** through a consistent repository-based abstraction, ideal for managing highly interconnected medical data such as relationships between doctors and patients. The latter offers seamless integration with **MongoDB**, **a document-oriented NoSQL database**, while preserving the flexibility and idiomatic programming style of the broader Spring ecosystem. Collectively, these technologies have provided a robust, scalable and modular architecture well-suited to the requirements of our application.

4.4 Back-End Structure

The application follows the layered architecture presented during lectures, organizing the codebase into clearly defined packages to promote modularity and separation of concerns.

4.4.1 `package it.unipi.healthhub.config`

The **Config** package usually contains configuration classes. In our case, we have only one class.

The `FilterConfig` class defines the registration of custom servlet filters used to manage access control within the application. Annotated with `@Configuration`, it registers filters via `FilterRegistrationBean`, associating each filter with specific URL patterns.

In detail:

- `DoctorDashboardAuthFilter`: restricts access to `/doctor/dashboard/*`;
- `PatientDashboardFilter`: applied to `/user/*`;
- `PatientApiFilter`: secures `/api/user/*`;
- `DoctorApiFilter`: secures `/api/doctor/*`;
- `LoginFilter`: manages access to the login page at `/login`.

4.4.2 `package it.unipi.healthhub.controller`

The `Controller` package manages API endpoint mappings and web page controllers for the core sections of the application, including the homepage, login, registration, and doctor dashboard. All classes in this package follow the Spring MVC (Model-View-Controller) framework, ensuring a clean separation of concerns between business logic, HTTP request handling, and view rendering.

More spefically:

- The `AuthController` handles authentication and registration processes for both patients (users) and doctors. Its responsibilities include rendering login, registration, and

password recovery views; managing user sessions; creating new user accounts; and processing password reset requests. Utility classes such as `ControllerUtil` and `HashUtil` are employed to promote modularity and code reuse.

- The `DoctorController` manages requests related to viewing a doctor's public profile. It retrieves the corresponding data via the `DoctorService` and injects it into the model for view rendering.

- The `DoctorDashboardController` supports navigation within the doctor's dashboard by mapping each HTTP GET request to a specific section such as appointments, profile, reviews, templates, and weekly schedule. It retrieves the authenticated doctor's data from the session and delegates business logic to the `DoctorService`.

- The `HomeController` handles requests to the root endpoints (`/`, `/index`) and to the `/search` page. Each method ensures that session-specific data is consistently added to the model using `ControllerUtil` before returning the appropriate view.

- The `UserController` manages authenticated user interactions through endpoints under `/user`, including profile viewing, appointments, and favorite doctors. It uses `ControllerUtil` to populate the model with session-based data. The class adopts a session-aware MVC design aimed at personalized user interaction.

4.4.3 package it.unipi.healthhub.controller.api

The `it.unipi.healthhub.controller.api` package defines a set of RESTful controllers that expose the application's core services via HTTP endpoints. These controllers are designed to return data (typically in JSON format) allowing asynchronous communication with frontend components and third-party systems.

Serving as the API layer of the system architecture, this package supports decoupled access to key resources such as users, doctors, appointments, and reviews. It complements the standard MVC controllers by enabling machine-oriented interactions, in alignment with modern web application practices.

DoctorAPI

The `DoctorAPI` class defines a RESTful web interface to manage and interact with doctor-related resources within the *HealthHub* system. It is annotated with `@RestController` and is mapped to the base path `/api/doctors`. Below is a summary of the main endpoints:

- `GET /api/doctors`
  Returns a list of all registered doctors.

- `GET /api/doctors/{id}`
  Retrieves a specific doctor by their unique identifier.

- `POST /api/doctors`
  Creates a new doctor resource.

- `PUT /api/doctors/{id}`
  Updates an existing doctor identified by `id`.
- `DELETE /api/doctors/{id}`
  Deletes the doctor with the specified `id`.
- `GET /api/doctors/{doctorId}/services`
  Retrieves the list of services offered by the specified doctor.
- `GET /api/doctors/{doctorId}/appointments`
  Returns the list of appointments associated with a doctor.
- `POST /api/doctors/{doctorId}/appointments`
  Books an appointment for the authenticated patient with the given doctor.
- `GET /api/doctors/{doctorId}/templates`
  Retrieves scheduling templates used by the doctor.
- `GET /api/doctors/{doctorId}/schedules/week?year=...&week=...`
  Returns the weekly schedule (including prenotable slots) for a doctor, given the ISO week and year.
- `GET /api/doctors/{doctorId}/endorsements`
  Retrieves the total number of endorsements for a doctor and whether the current user has endorsed them.
- `POST /api/doctors/{doctorId}/endorse`
  Allows a logged-in patient to endorse a doctor.
- `POST /api/doctors/{doctorId}/unendorse`
  Removes an existing endorsement made by the patient.
- `GET /api/doctors/{doctorId}/reviews`
  Returns all patient reviews for the specified doctor.
- `POST /api/doctors/{doctorId}/reviews`
  Submits a review for a doctor, provided the patient has had an appointment with them.
- `GET /api/doctors/{doctorId}/reviews/week/count`
  Returns the number of reviews a doctor received during the current ISO week.

PrivateDoctorAPI

The `PrivateDoctorAPI` class implements a secure REST interface for authenticated doctors to manage their personal data, appointments, and analytic insights. Annotated with `@RestController` and mapped to `/api/doctor`, it delegates domain logic to `AppointmentService` and `DoctorService`, enforcing session-based access control. Below is a summary of the main endpoints:

- `GET /api/doctor/appointments`
  retrieves today's appointments (or for a given date).
- `DELETE /api/doctor/appointments/{id}`
  deletes an appointment.
- `PUT /api/doctor/address`
  updates the doctor's address.
- `PUT /api/doctor/info`
  updates the doctor's profile info.
- `POST /api/doctor/phone`
  adds a phone number.
- `DELETE /api/doctor/phone/{index}`
  removes a phone number by index.
- `POST /api/doctor/specialization`
  adds a specialization.
- `DELETE /api/doctor/specialization/{index}`
  removes a specialization.
- `POST /api/doctor/service`
  adds a medical service.
- `GET /api/doctor/service`
  retrieves all offered services.
- `PUT /api/doctor/service`
  updates a medical service.
- `DELETE /api/doctor/service/{id}`
  deletes a medical service.
- `POST /api/doctor/template`
  creates a new availability template.
- `GET /api/doctor/template`
  retrieves all availability templates.
- `PUT /api/doctor/template`
  updates a template.
- `DELETE /api/doctor/template/{id}`
  deletes a template.
- `PUT /api/doctor/template/default`
  sets the default availability template.

- `POST /api/doctor/schedule`
  creates a weekly schedule.
- `GET /api/doctor/schedule`
  retrieves the weekly schedule.
- `PUT /api/doctor/schedule`
  updates the weekly schedule.
- `DELETE /api/doctor/schedule`
  deletes the weekly schedule.
- `GET /api/doctor/reviews`
  retrieves all received reviews.
- `DELETE /api/doctor/review/{id}`
  deletes a review.
- `PUT /api/doctor/password`
  changes the password.
- `GET /api/doctor/stats/visits`
  gets number of visits (optionally weekly).
- `GET /api/doctor/stats/earnings`
  gets annual earnings.
- `GET /api/doctor/stats/patients`
  gets number of new patients per month.

UserAPI

The `UserAPI` controller is a RESTful web service component. It provides endpoints for managing user entities within the HealthHub application, leveraging the `UserService` to perform CRUD (Create, Read, Update, Delete) operations on patients stored in a MongoDB database. The controller follows standard REST conventions, utilizing HTTP verbs to represent actions on resources.

- `GET /api/users`
  Retrieves a list of all users registered in the system.
- `GET /api/users/{id}`
  Fetches a specific user by their unique identifier. Returns HTTP 200 with the user data if found, otherwise HTTP 404.
- `POST /api/users`
  Creates a new user entity based on the data provided in the request body. Returns the created user.

- `PUT /api/users/{id}`
  Updates an existing user identified by the given ID with new data provided in the request body. Returns HTTP 200 with the updated user if the operation is successful; returns HTTP 404 if the user does not exist.

- `DELETE /api/users/{id}`
  Deletes the user identified by the specified ID. Returns HTTP 204 No Content regardless of whether the user existed, indicating that the request has been processed.

PrivateUserAPI

The `PrivateUserAPI` controller is a RESTful web service implemented using the Spring framework in Java, designed to provide secured endpoints for user-specific operations within the HealthHub system. It leverages HTTP sessions to identify the authenticated user (typically a patient) and delegates business logic to the `UserService`. The controller exposes various endpoints to manage user details, contacts, appointments, doctor recommendations, and password changes, emphasizing privacy and personalization.

- `GET /api/user/details`
  Retrieves detailed personal information of the currently authenticated user by extracting the patient ID from the HTTP session. Returns user details or an error message in case of failure.

- `PUT /api/user/details`
  Updates the personal details of the authenticated user. The new data is supplied in the request body as a `UserDetailsDTO` object.

- `GET /api/user/details/view?id={id}`
  Retrieves contact information and demographic data for a user specified by the `id` query parameter. Additionally, if a doctor is logged in, it returns the number of visits that doctor has conducted with the user.

- `GET /api/user/contacts`
  Fetches the contact information related to the authenticated user.

- `PUT /api/user/contacts`
  Updates the contact information of the authenticated user using data provided in the request body as a `PatientContactsDTO`.

- `GET /api/user/appointments/upcoming`
  Retrieves a list of upcoming appointments for the authenticated user.

- `GET /api/user/appointments/past`
  Retrieves a list of past appointments for the authenticated user.

- `DELETE /api/user/appointments/{appointmentId}`
  Cancels the appointment identified by `appointmentId`. Returns HTTP 200 on success or HTTP 400 if the appointment does not exist.

- `GET /api/user/recommendations?limit={limit}`
  Returns a list of recommended doctors for the authenticated user, with an optional parameter `limit` to specify the maximum number of results (default is 3).

- `GET /api/user/doctors/reviewed`
  Returns a list of doctors that the authenticated user has reviewed.

- `GET /api/user/doctors/endorsed`
  Returns a list of doctors endorsed by the authenticated user.

- `PUT /api/user/password`
  Allows the authenticated user to change their password. The current and new passwords are supplied in the request body as a `PasswordChangeDTO` and are hashed before validation and update.

SearchAPI

The `SearchAPI` controller is a RESTful component, responsible for executing search queries specifically targeted at medical doctors within the HealthHub system. This controller integrates data from two distinct data sources: a MongoDB database and a Neo4j graph database. The rationale behind this dual-source approach is to combine traditional document-oriented search results with personalized recommendations derived from graph-based relationships.

The primary endpoint exposed by this controller is: `GET /api/search/doctors`. This endpoint accepts a mandatory `query` parameter, representing the search string to be matched against doctor records. Additionally, it accesses the HTTP session to retrieve the identifier of the currently authenticated patient (if available), enabling personalized search refinement.

Upon receiving a valid query, the controller queries the MongoDB database via `DoctorService`, returning results as `DoctorMongoProjection` objects that represent doctor records optimized for search display. If the user is unauthenticated or the patient identifier is absent, the controller returns the top ten doctors from the MongoDB results, sorted by relevance.

For authenticated users, the controller enriches the search outcomes by querying the Neo4j graph database to retrieve personalized recommendations. These recommendations are returned as `DoctorNeo4jProjection` objects, each associated with a relevance score reflecting the strength of the doctor-patient relationship, such as prior interactions or endorsements. In the absence of Neo4j results, the system falls back to the top MongoDB results.

The integration of results from both sources is achieved by mapping doctor identifiers to Neo4j scores, then augmenting each MongoDB doctor's score accordingly. Doctors without Neo4j matches receive a default negative score to denote lack of personalized relevance. The aggregated list is sorted in descending order by combined score, and the top ten entries are converted into `DoctorDTO` objects for API response consistency.

Two private helper methods, `toDto` and `toTop10Dto`, support this process by converting raw entities to DTOs and by sorting and limiting the result set. This design ensures that search responses maintain a uniform format, whether or not personalization is applied.

Overall, the `SearchAPI` exemplifies a hybrid search strategy that effectively merges traditional document-based retrieval with graph-driven personalization, leveraging session data to optimize relevance and incorporating fallback mechanisms to ensure robust performance for both authenticated and anonymous users.

4.4.4 `package it.unipi.healthhub.dto`

The Data Transfer Objects (DTO) directory encapsulates a set of plain Java objects that serve as data carriers between different layers of the HealthHub application, primarily facilitating communication between the service and presentation tiers. These DTO classes are designed to aggregate and structure data relevant to specific business operations or user interactions, decoupling the internal domain model from external API contracts.

4.4.5 `package it.unipi.healthhub.dto`

...

Bibliography