Politecnico di Torino

Master of Science in Computer Engineering

Master's Thesis

Health Monitoring Application

Supervisor
Prof. Morisio Maurizio

Candidate
Mohamed Tourab
ID: s259371

1. Introduction

The World Health Organization (WHO), in its 2019 "Global Health Estimates," reported a worldwide increase in life expectancy over the last two decades [1]. Scientific advancements have significantly improved quality of life, resulting in a rise in average life expectancy (ALE) from 66.8 years to 73.4 years between 2000 and 2019.

Additionally, the study highlights another critical metric: Healthy Average Life Expectancy (HALE), which measures the average lifespan spent in good health. This metric increased from 58.3 years to 63.7 years during the same period. While ALE showed an increase of 6.6 years, HALE grew by only 5.4 years, indicating a slower rate of improvement in healthy life expectancy.

These statistics point to the challenge of achieving "quality longevity." Projections suggest that HALE will continue to diverge from overall longevity due to the higher likelihood of negative health events, such as cardiovascular and degenerative diseases, as people age. Therefore, it becomes increasingly important to focus on not just extending lifespan but also maintaining health throughout those years.

From a young age, it is crucial to adopt a lifestyle that promotes healthy aging, allowing individuals to enjoy their later years in good health. Modern medicine has enabled survival despite serious illnesses, but the quality of life remains a key consideration.

In recent years, the importance of a healthy lifestyle has been extensively discussed across various media platforms[2], both modern (like social media) and traditional (like television). This vast array of information can make it challenging to discern useful advice from misinformation. Additionally, encouraging positive long-term behaviors that require effort is a significant obstacle, as human psychology tends to prefer immediate rewards with minimal effort. However, consistent adherence to beneficial daily habits leads to a better quality of life and lowers the risk of severe diseases in old age. This concept can be difficult to grasp without external support.

The challenge is even greater when encouraging lifestyle changes in older adults, who may have deeply ingrained habits. Effectively promoting positive, lasting changes in this group often meets with resistance.

If well-being is imagined as an iceberg, the visible, immediate results represent only the tip. The true commitment to a healthy lifestyle lies in the submerged part of the iceberg, achieved through persistent practice of good habits. While the benefits may be less noticeable in the short term, this persistence helps prevent significant health issues later in life. Delving into the depths of this iceberg requires courage and a strong sense of self-responsibility. Thus, having guidance is essential.

The advent of smartphones has enabled the general public to monitor their physical activity and receive immediate feedback on their fitness levels. Having a miniature computer in one's pocket provides instant access to unlimited and personalized information, even at the gym or while running. Additionally, the numerous sensors (accelerometer, GPS, etc.) in modern phones have allowed for the development of a wide range of fitness applications. Today, anyone can, independently and without expert intervention:

- Access libraries and collections of exercises for weight loss, muscle gain, or other specific goals.

- Track running sessions, routes, distances covered, and times taken, providing objective and instant recording of personal improvements.
- Maintain a food diary to monitor the quantity, quality, and variety of nutrition.
- Receive reminders throughout the day to drink enough water.
- Be introduced to meditation with guided exercises to understand the importance of mental health.

Over the years, these apps have become increasingly intuitive and user-friendly. Furthermore, methods to incentivize their use, such as gamification techniques and the creation of dedicated communities, have been introduced. In the former case, users can earn badges and personalized recognitions, digital rewards, and, in rare instances, tangible prizes. Utilizing the social aspect provides additional motivation to reach certain goals, which can then be shared on social networks with groups participating in the same workouts or sports events.

These apps have proven to be excellent tools for increasing awareness about wellness and personal health for millions of people. However, they are not without limitations: most require active participation in data consultation and feature activation. Users must set aside time in their day for their use and always carry their phone, which cannot monitor certain sports.

To address this issue, passive wellness devices have been developed. Wearables, such as smartwatches, bracelets, rings, and glasses, offer many functionalities continuously and silently without user interaction. The market offers a wide range of these devices, from inexpensive models costing a few dozen euros to professional-grade tools with features suitable for medical use and significantly higher prices.

The main differentiators among devices are the variety and accuracy of the data generated, which depend on the onboard sensors and data analysis algorithms. Common sensors in wearables include:

- Accelerometers for measuring device acceleration, usually due to user movement.
- Gyroscopes for measuring angular velocity to determine device orientation.
- Altimeters for determining altitude.
- Thermometers for measuring the user's or ambient temperature.
- GPS for tracking user coordinates, especially useful without a smartphone.
- Electrocardiograms (ECG) for measuring heart activity and rate.
- Oximeters for detecting blood oxygen levels.
- Electrodermograph (EDG) for measuring skin conductivity to assess sweat levels.

The quality of sensors determines measurement accuracy, thus the reliability of the information produced. The variety of sensors increases the potential effectiveness of data analysis algorithms. Sleep information, for example, is derived from raw data on movement, temperature, and heart rate.

Common functionalities offered by wearables include:

- Physical activity tracking, such as steps taken, distance covered, and elevation change. These functions track daily minutes spent sedentary versus active, indicating activity intensity and estimating calories burned.

- Heart rate monitoring, providing constant measurement of resting heart rate, time spent in various heart rate zones indicating activity levels, detecting abnormal peaks, and recovery time post-exercise.
- Sleep monitoring, assessing both quantity and quality by combining parameters like lack of movement, heart rate, temperature, and respiration rate to estimate minutes spent in each sleep phase (light, deep, REM) and detect interruptions.

Wearables are highly useful for lifestyle monitoring, requiring minimal effort to use. As technology advances, data analysis becomes more accurate. Basic models are affordable, around a few dozen euros, yet offer sufficient functionalities for many users, making them popular, especially among younger generations.

However, adoption among older adults is significantly lower. Despite their ease of use and suitability even for those with no digital skills, the main challenge lies in data interpretation. The volume of data produced necessitates complex graphical interfaces, with numerous elements on the screen such as charts, numerical data, and units of measure. While functional for average users, these interfaces can be confusing and hard to read for those unaccustomed to digital displays.

2. Objectives

The application project, HealthApp, was born from a collaboration between DAUIN and doctors from Verona Hospital, led by Dr. Vincenzo Di Francesco, director of the Complex Geriatrics Unit III at Borgo Trento, in collaboration with the Pensioners Union (SPI) of the Italian General Confederation of Labour (CGIL). It aims to support a medical experimental study and use information technology to enhance patient well-being. Around a hundred elderly patients, recently hospitalized due to trauma or surgery, will be monitored for psychophysical health over six months after discharge. This monitoring includes check-ups, periodic questionnaires, and the collection of lifestyle-related data.

Check-ups will be conducted at the beginning and end of the study to determine the patient's initial clinical state and its evolution. Some questionnaires will be informational, providing indicators on data that are difficult to automate, like nutritional tests. Others will have validated medical significance, such as the Multidimensional Prognostic Index (MPI)[3], a prognostic index for adverse events like mortality, hospitalization, and institutionalization. The MPI is calculated using a mathematical algorithm that includes information from eight domains: basic and instrumental activities of daily living, cognitive status, nutritional status, risk of pressure sores or mobility issues, multimorbidity, polypharmacy, and living situation. The MPI output can be a continuous numerical index from 0 (no risk) to 1 (maximum risk) or in three mortality risk grades: low (MPI-1), moderate (MPI-2), or severe (MPI-3).

Various versions of the MPI have been validated in different clinical settings worldwide, showing high accuracy in predicting mortality and other adverse events in the elderly[4]. It is sensitive to changes in multidimensional frailty over time, making it one of the most commonly used tools for identifying and measuring frailty in clinical practice. Each patient will also use a smart fitness bracelet to collect data on daily physical activity, heart rate, and sleep quality, with the data accessible via a web application and a mobile app.

The study divides volunteers into an experimental group and a control group. Both groups will collect data, but the experimental group will receive frequent feedback on their data, with lifestyle indicators expressed simply and intuitively, from green (positive) to red (negative). The control group will view their data without feedback. The goal is to show that elderly individuals, when encouraged to adopt healthy behaviors, can recover more effectively, potentially reducing future hospitalization times and allowing clinical recovery in more comfortable environments.

This thesis project, involving five students under a professor's guidance, focuses on developing a software application with both back-end and front-end components. Following an extensive phase of study and design, we identified the main use cases, made architectural and implementation choices, and integrated the functionalities into a single application. Patients can monitor parameters related to nutrition, physical activity, health, and sleep, track progress, and receive improvement suggestions. Patients can answer custom questionnaires tailored by doctors, track food consumption, analyze physical activity data, monitor health parameters, and assess sleep quality, providing comprehensive insights for better patient care. Having surpassed this initial descriptive phase, the technical choices made will now be discussed, illustrating and explaining them.

3. Methodology & Architecture

3.1 Choosing Tracker

One of the major challenges faced in the project concerns the method of acquiring data from the tracker. The original idea aimed to develop an initial version of a mobile application with low-level interaction capabilities with a selected tracker. This application, installed on patients' smartphones, would feature a minimal graphical interface to guide the patient through authentication and pairing with the bracelet. Subsequently, a background service responsible for periodic synchronization would establish a connection with the bracelet, process and transform the data according to the specifications provided by the backend APIs, and finally store the information in the database.

The intention was to provide several adapters that would allow connection with different models, supporting trackers from various brands. Although the underlying communication technology is the same (Bluetooth), data encoding varies by manufacturer. Each adapter would handle different input data formats and produce output information modeled according to our database schema.

Through research and study of the type of connection between bracelets and smartphones, we identified the communication protocol more precisely[5]. It is called Bluetooth Low Energy (BLE) and is one of the main technologies of the Internet of Things (IoT).

As the name suggests, the main difference from classic Bluetooth is power consumption. BLE is used to interact with devices with limited battery capacity, exchanging small amounts of data periodically. Like classic Bluetooth, BLE operates in the 2.4 GHz frequency band, but the connections last only a few milliseconds before returning to sleep mode. This optimization allows for power consumption up to 100 times lower than common Bluetooth, with a limited bit rate of 125 Kb/s, 500 Kb/s, or 1 Mb/s depending on the version used.

The next step was to understand the formats and procedures used for data transfer. These are defined by the Generic Attribute Profile (GATT) associated with the wearable, which uses the Attribute Protocol (ATT) as the transport protocol [12].

This is a client-server connection where the tracker acts as the GATT server, ready to receive read/write requests for small amounts of data, called attributes. The client, which in our setup is the user's smartphone, is responsible for initiating the transaction.

Attributes are addressable pieces of information containing user data (or metadata) conceptually located on the server and accessible (or potentially modifiable) by the client.

Table 1. provides examples of attributes represented in tabular form.

Handle	Type	Permissions	Value	Length
0x0201	$UUID_1$ (16 bit)	R, no security	0x180A	2
0x0202	$UUID_2$ (16 bit)	R, no security	0x2A29	2
0x0215	$UUID_3$ (16 bit)	R/W, authorization	"string"	23
0x030C	$UUID_4$ (128 bit)	W, no security	{0xFF, 0xFF, 0x00, 0x00}	4
0x030D	$UUID_5$ (128 bit)	R/W, authentication	36.43	8
0x031A	$UUID_1$ (16 bit)	R, no security	0x1801	2

Each attribute is defined by:

- **Handle**: A unique 16-bit ID that allows the attribute to be addressed. Its value remains constant even across different connections.
- **Type**: A UUID (Universal Unique Identifier) that defines the type of attribute, i.e., the meaning to be assigned to the value. The ID can be 16, 32, or 128 bits. Shorter formats are exclusively used for standard attribute types, as defined by the Bluetooth SIG (Special Interest Group) in the 16-bit UUID Numbers Document. The 128-bit format is available for manufacturers to identify device-specific attributes.
- **Permissions**: Metadata that establishes which operations (read and/or write) are allowed on the attribute and what security requirements are necessary. Access to sensitive attributes may require user authorization. In some cases, an encrypted connection may be required to interact with the device.
- **Value**: The actual value of the attribute. There are no restrictions on the type of data it can contain.
- **Length**: The length of the value field, limited to a maximum of 512 bytes.

Attributes can be classified into services, characteristics, and descriptors, which allow for hierarchical structuring.

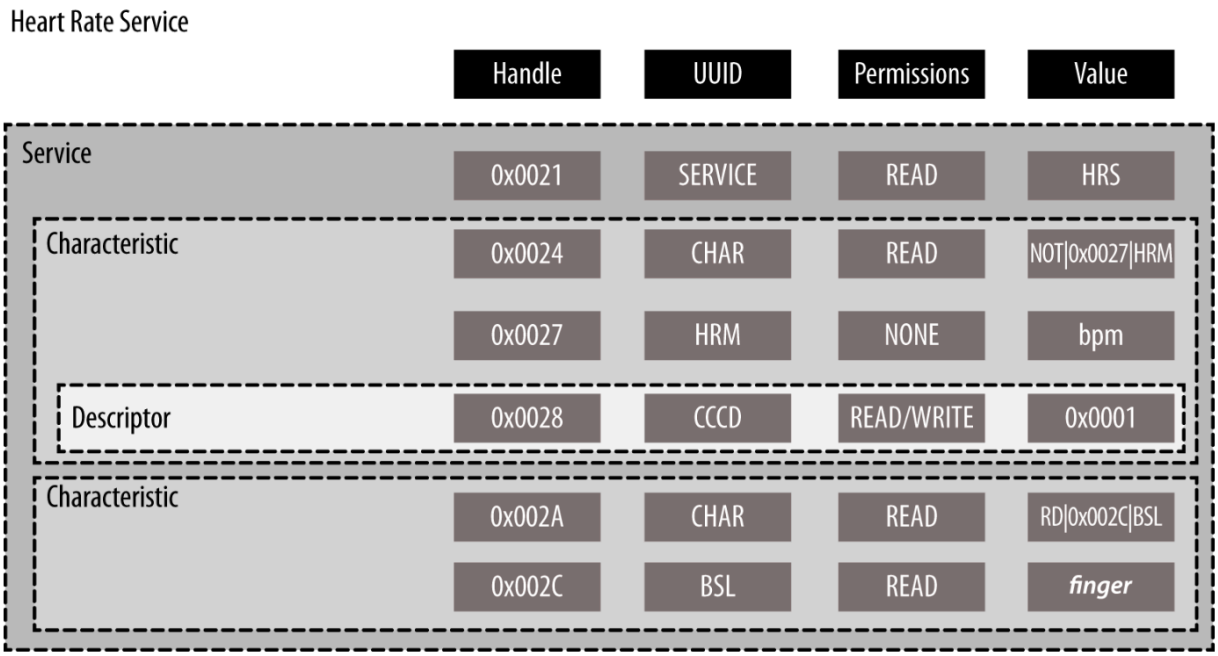


Figure 1

From figure 1. We see that all conceptually related attributes are grouped into services, each of which can contain zero or more characteristics. In turn, each characteristic can contain zero or more descriptors. The first attribute of a service, also known as a service declaration, always has a UUID defined by the Bluetooth SIG standards. Its value corresponds to the actual UUID of the exposed service, which can be either a standard or custom service.

Characteristics can be considered as containers of user data. They also have a first declaration attribute indicating the start of the characteristic and providing metadata related to subsequent values. Additional metadata is declared through descriptors. The declaration attributes for services and characteristics are used by the client during an initial discovery phase to identify the data exposed by the device. Only after retrieving the UUIDs and metadata is it possible to access the user data.

After studying the protocol, an attempt was made to establish a connection with a Xiaomi Mi Band 3 tracker [6], successfully identifying the exposed services and the addresses to retrieve the daily step count and heart rate. These attributes are identified by UUIDs 0x2B40 and 0x180D respectively, as specified by the Bluetooth SIG. This was a significant advantage, as device manufacturers can choose to use either custom or standard UUIDs; if they had opted for custom UUIDs, understanding the attribute data type would have been more complicated. For example, bracelets produced by Fitbit use vendor-specific identifiers for almost all user data, necessitating significant reverse engineering efforts to decipher the exposed attribute types.

In accordance with the specifications requested by the doctors, we initially identified three essential data points: the number of steps, resting heart rate, and daily sleep minutes. However, two major issues were encountered.

The first issue concerns derived data, such as sleep. According to our research, most mid-range devices suitable for the study do not perform local derivation calculations but instead rely on the manufacturers' proprietary servers. To fully utilize the trackers' capabilities, users must install the associated smartphone application. This way, the raw data directly measured/calculated by the wearable is sent to the servers, which store it and process it through proprietary algorithms to derive more complex data. For example, to determine if the user is sleeping, at least an accelerometer and heart rate readings are necessary.

The business success of wearable device companies largely depends on the effectiveness and accuracy of their algorithms and the management of vast amounts of data stored on their servers. Processing data directly on the devices would present significant disadvantages for companies:

- The need to equip wearables with more powerful and expensive chips.
- Reduced battery life due to increased processing.
- Greater exposure of intellectual property, as algorithms would need to be executed on each device's software.
- Users might choose not to install the associated application.

Therefore, low-level interaction with the wearable would not provide sleep-related information. This first problem could potentially be resolved by developing an algorithm to recognize sleep phases, but this would require a complex study. This option remains a possible solution for future developments.

The second issue is more challenging to resolve. The Xiaomi Mi Band 3, released in 2018, appears to be one of the last wearables that can interface without security requirements. Modern wearables are more advanced in protecting user data, requiring authorization processes and encrypting sensitive data. Only proprietary applications can adequately interface with the trackers.

Due to these issues, alternative solutions were sought. Leading fitness tracker manufacturers such as Fitbit and Zepp were contacted to obtain development tools for higher-level interaction with wearables. Zepp, a leading company in the smart wearable and fitness device market, develops the Zepp Life app, associated with both Amazfit devices (which Zepp also produces) and Xiaomi Mi Band products. Zepp offered the Zepp SDK, a mobile library providing APIs for interacting with most of the devices from both companies. The process to gain SDK access involved several steps to accredit as Zepp developers and demonstrate future uses of the library. Unfortunately, due to a lack of further communication from Zepp, this solution could not be pursued. In recent months, Zepp's website underwent rebranding, and SDK references were removed, likely indicating a transition in company strategies.

The only tool available for external developers from these companies is the web API. Both Fitbit and Zepp provide REST endpoints to retrieve data related to tracker-associated accounts, following authentication and authorization processes [7]. **Fitbit's web API was chosen as the solution to adopt.**

Since interfacing directly with third-party devices involves the previously discussed challenges, which are difficult to resolve within the expected study time frame, the workaround involved interacting with third-party servers (initially Fitbit's) that maintain user data. In this case, the data is already processed and ready to be retrieved and adapted to our database model.

The empirical nature of this work required revising the initial solution during the process, leading to several limitations and changes to the project:

- Patients were required to install the app associated with the chosen tracker, as the app is essential for synchronizing data from the tracker to Fitbit's servers. Additionally, each patient had to create an associated Fitbit account.
- The HealthApp mobile application was scaled down with dual functionalities in the web version and additional implementations for recommendations and incentives. This eliminated the need to implement the connection logic with the tracker.
- Control over user data was lost, becoming dependent on third parties.

While remaining functional relative to the requested specifications, there are many avenues for future improvements and developments. The project is ambitious and could extend to thesis work focused on hardware development to design custom trackers.

3.2 Three-Tier Architecture

Three Tier Architecture

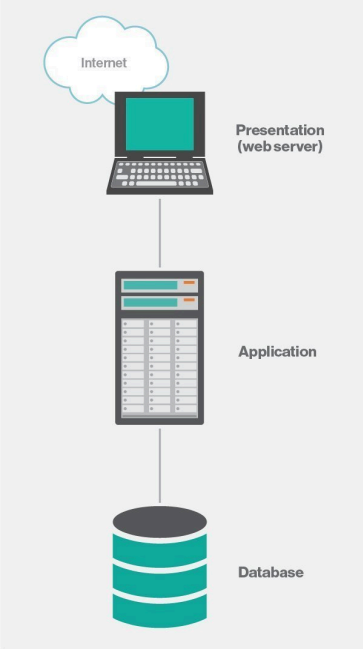


Figure 2 Three tier Architecture

The three-tier architecture is composed of three different logical and physical levels [10]:

- **Data Tier**: In this section, the data used by the application is stored and managed. In the case of HealthApp, a relational database based on SQL has been chosen.
- **Application Tier**: Commonly referred to as the "Server," it represents the core of the application where information is processed and prepared for presentation. This layer acts as an intermediary between the data and the client.
- **Presentation Tier**: Represents the user interface (which a client will access) where the received information is presented to the user, who can interact with it and make decisions based on it. Within the application, there are two types of clients: desktop (laptops, notebooks, etc.) and mobile (smartphones and tablets).

The following figure summarizes and clarifies the architecture of HealthApp:

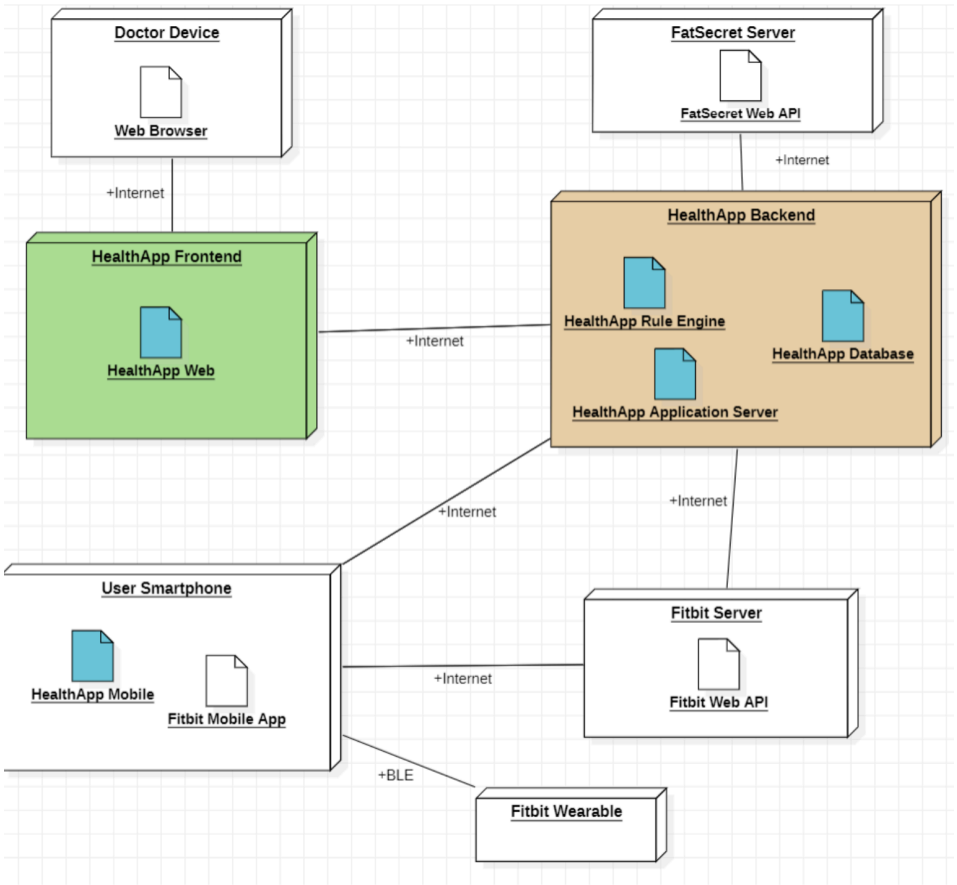


Figure 3 - Architecture Diagram (Deployment Diagram)

The diagram illustrates the relationships between the hardware (nodes) and the software (artifacts) within the application.

The two main nodes, namely the back-end and the front-end, are highlighted in orange and green, respectively. The components that are the direct subject of development and thus represent the core of the

project are shown in blue, while the various external agents (e.g., third-party services, hardware, etc.) that interact with the application to ensure its proper functioning are shown in white.
Delving into the details, regarding the hardware we have:

- **Fitbit Wearable**: A smart bracelet worn by patients that collects data on movement, health, and sleep.
- **User Smartphone**: Interfaces with the wearable through a BLE (Bluetooth Low Energy) connection and is the device through which patients use the functionalities of HealthApp. It interfaces with the backend via an internet connection.
- **HealthApp Backend**: Represents the machine where the backend of the application is executed. In addition to interfacing with the frontend, it also interfaces with third-party servers such as “Fitbit” to retrieve some of the necessary data in the different macro areas.
- **HealthApp Frontend**: Represents the machine where the web frontend of the application is executed. It interfaces with the backend via an internet connection.
- **Doctor Device**: Interfaces with the HealthApp Frontend through an internet connection and is the device (desktop, notebook, tablet, etc.) through which doctors can monitor their patients.
- **Fitbit Server**: Stores the data related to movement, health, and sleep produced by the wearables of various patients and makes it available to the HealthApp Backend via API.

Regarding the software modules, we have:

- **HealthApp Mobile**: Runs on the mobile device and is used by patients.
- **Fitbit Mobile App**: Runs on the mobile device and is used by patients as a tool for connection and data transmission through the wearable. This proprietary app will subsequently upload the data to the Fitbit Server.
- **HealthApp Web**: Runs on the HealthApp Frontend and is used by doctors to monitor patients.
- **Web Browser**: Necessary artifact to access the HealthApp Frontend.
- **Fitbit Web API**: Used by the HealthApp Backend to interface with the Fitbit server and retrieve the movement, health, and sleep data of patients from the proprietary app.
- **HealthApp Database**: Structure representing the application's database and containing all the information necessary for its proper functioning.
- **HealthApp Application Server**: The heart of the application, capable of managing access control, manipulating data in the HealthApp Database, and communicating with third-party servers as well as the mobile and web frontend. This communication occurs via APIs that allow for querying the information stored by the server.

It is observed that in the backend of HealthApp, data and server coexist on a single machine, with the aim of implementing this separation in a subsequent distribution phase. Communication between the Server and Client occurs through REST APIs.

A consequence of this architectural choice is the emergence of two clearly distinct roles: the figure responsible for the presentation layer (frontend developer) and the one responsible for the application and data layer (backend developer). Within the context of HealthApp, there is a further subdivision within the presentation layer, partitioned into mobile and web sublayers.

This introductory chapter will conclude with a brief illustration of the frameworks adopted in the development of the various layers of the application.

3.3 Backend

The application and data layer, commonly known as the backend, must be able to guarantee some minimum requirements, including:

- Continuous availability
- Security in the storage and manipulation of sensitive information
- Robustness against potential external intrusions
- Consistency of data accessed by different clients

To adopt a relational schema for interfacing, a possible solution is to use the Express framework for Node.js for the application layer, combined with SQL for the data layer. The strengths of this solution lie in its simplicity and immediacy of use, as well as its flexibility in creating a custom configuration. However, this simplicity conflicts with the need for a working version within a reasonable timeframe as it reduces productivity and requires the manual implementation of every security control, a method widely discouraged in IT as it likely exposes the system to vulnerabilities exploitable by attackers.

For this reason, we chose to use Spring Boot, a Java framework, combined with the more recent Kotlin language for the application layer, along with SQL for the data layer. Spring Boot offers the possibility to create a solution that is scalable, modular (there is a dependency system to import only the necessary modules), simple to develop, suitable for the required needs, secure (since various controls are automatically performed by the framework), and reduces development time, thus increasing productivity [8]. We utilized Kotlin to leverage its features such as high-order functions, coroutines, and extension functions [14].

To ensure our backend remains flexible and easy to maintain as requirements change frequently, we adopted a combination of Hexagonal Architecture and Layered Architecture. This approach offers the flexibility needed to stay agile while maintaining a robust code base. Hexagonal Architecture, in particular, helps to isolate our domain logic from external influences such as databases, infrastructure, or REST API controllers. By using interfaces around our services (domain logic), we ensure that changes in the surrounding environment do not impact the core business logic.

For integration tests, we used the H2 database, and database migration was managed using Liquibase to handle database versions effectively [9].

The role of the backend, in its entirety, is to respond to client requests made through REST APIs and to communicate with the external service providers integrated into the application (Fitbit for movement), which will be discussed in more detail later.

3.4 Frontend

The presentation layer, as previously mentioned, is divided into two sublayers: mobile and web.

The web sub-layer is implemented using the React framework for JavaScript [15]. It offers ease of development through a modular (component-based) approach and is a versatile solution that adapts to both classes of users: Doctors and Patients. The former can easily manage the latter, viewing their most relevant data, manipulating it, and integrating it with additional information of interest. Patients, on the other hand, can only view and manipulate their own data without the possibility of accessing confidential information, ensuring the security and protection of each user.

The mobile sub-layer, on the other hand, is implemented using the React-Native framework for JavaScript [16]. This choice is motivated not only by the modularity offered in development but also by the ability to support the two most widespread mobile operating systems: iOS and Android. Productivity is thus maximized at the expense of the implementable functionalities, which are restricted to those offered by both mobile systems, but these functionalities are more than sufficient for the application's purposes. This sub-layer is intended exclusively for patients, as it is considered essential for data collection and quick visualization of suggestions and trends, whereas for more in-depth consultation and analysis (reasonably performed by a doctor), a web application is preferred.

Before concluding the chapter, one last important topic to address concerns user interactions. As previously reiterated, two classes of users have been identified: doctors and patients. While the doctor will interact exclusively with the application through their personal computer, the patient can do so from both mobile and web platforms. Additionally, the data collection procedure requires the patient to use a wearable device, specifically an activity tracker, to collect data on movement, sleep, and partially on health. This data will then be managed by the application and organized into the four macro-sections previously mentioned (Nutrition, Movement, Health, Sleep).

4. Architecture

In this section, we will delve into our architectural decisions and the frameworks and libraries we have utilized. These choices were crucial because, as we mentioned earlier, they impact our development time and our ability to adapt to new requirements. Throughout the development process, we continually discovered new possibilities, making flexibility a key consideration in our decisions.

4.1 Important Concepts

4.1.1 Spring Framework

The Spring Framework is a comprehensive and versatile platform for developing Java applications. Its robust architecture simplifies the development process, promotes good coding practices, and provides a variety of tools to create enterprise-level applications. In this article, we will explore the core components of the Spring architecture as depicted in the provided image, which highlights the key aspects: Dependency Injection (DI), Inversion of Control (IoC), Aspect-Oriented Programming (AOP), and the Architecture Module.

- **Dependency Injection (DI):** Dependency Injection is a design pattern used to implement IoC, allowing the creation of dependent objects outside of a class and providing those objects to a class in different ways. Spring implements DI to promote loose coupling between different components. In the image, DI is mentioned twice, underscoring its importance in the Spring Framework. By managing the dependencies of various objects, DI makes the system more modular and easier to test.
- **Inversion of Control (IOC):** Inversion of Control is a principle in which the control of objects or portions of a program is transferred to a container or framework. In Spring, IOC is achieved through Dependency Injection. The image shows IOC as a foundational component, indicating its crucial role in managing the lifecycle of Spring beans. By decoupling the execution of a certain task from the implementation, Spring's IOC container allows for more flexible and maintainable code.
- **Aspect-Oriented Programming (AOP):** Aspect-Oriented Programming allows developers to separate cross-cutting concerns from the business logic of the application. These concerns can include logging, transaction management, security, etc. AOP helps in modularizing these concerns, leading to cleaner and more manageable code. In the image, AOP is positioned as a critical aspect of the Spring Framework, enabling developers to define such aspects in a declarative manner, without cluttering the core business logic.
- **Architecture Module:** The Architecture Module in Spring is a higher-level component that integrates various other modules and tools provided by Spring. This module ensures that different aspects of the Spring Framework work seamlessly together, providing a cohesive development experience. The image places the Architecture Module as the final component, signifying its role in harmonizing the various features of the framework.

Spring not only offers these core features but also provides a variety of additional libraries and tools that enhance the development experience and add powerful capabilities to applications for instance it offer the following libraries that we made our lives much easier while building the HealthApp:

- **Spring REST Template:** The RestTemplate is a synchronous client to perform HTTP requests. It simplifies communication with HTTP servers, and enforces RESTful principles. RestTemplate provides higher-level methods for interacting with HTTP servers, making it easy to consume RESTful services.
- **Spring Security:** Spring Security is a powerful and highly customizable authentication and access-control framework. It provides comprehensive security services for Java applications, including authentication, authorization, and protection against common vulnerabilities.
- **Spring Beans:** Spring Beans are the objects that form the backbone of a Spring application. They are managed by the Spring IoC container. The configuration metadata that you supply to the container tells Spring how to create and configure the beans.
- **Spring JDBC:** Spring JDBC simplifies the use of JDBC and helps to avoid common errors. It provides a consistent way to handle errors, manage transactions, and obtain connections, offering templates that reduce the boilerplate code needed to interact with a database.
- **Spring Web and Spring Web MVC:** Spring Web provides the basic web-oriented integration features for Spring applications, while Spring Web MVC is a comprehensive and flexible framework for building web applications. It follows the Model-View-Controller (MVC) design pattern, separating application logic, user interface, and input control, thus promoting a clean separation of concerns

All these concepts have been used extensively in our project. The Spring Framework, used within the HealthApp, suggests the use of several components, each with its own responsibility. A component within the framework is declared using annotations such as "@" followed by the type of component. The base annotation is @Component, which can be specialized further into @Controller, @Service, and @Repository.

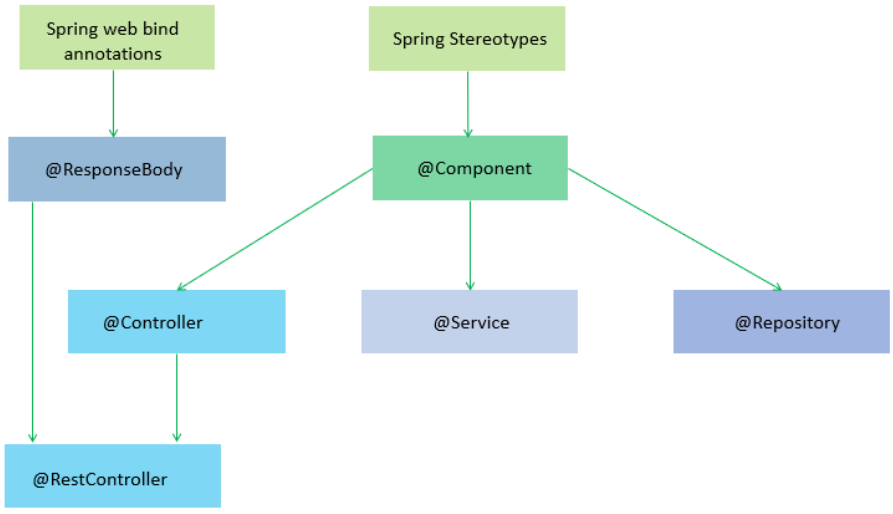


Figure 4 – Component Specializations

Each of these annotations has been utilized in different layers, adhering to the layered architecture we implemented. More information on the layered architecture and hexagonal architecture will be provided in the following sections.

In addition to the core components, we leveraged Spring's rich ecosystem of libraries to enhance the functionality and security of HealthApp. We used RestTemplate for seamless communication with external services, Spring Security to implement robust authentication and authorization mechanisms, and Spring JDBC for efficient and error-free database interactions. The Spring Web MVC framework helped us maintain a clean separation between the business logic and presentation layer, making our application easier to manage and scale.

By utilizing these powerful tools and adhering to best practices provided by the Spring Framework, our HealthApp project is robust, secure, and maintainable, demonstrating the strength and flexibility of the Spring ecosystem

Before delving into these topics, it is important to mention that our application server is built not only on the Spring Framework but also on the Spring Boot framework, which extends Spring. Spring Boot significantly simplifies the development process by providing a faster and more efficient way to build production-ready applications with minimal configuration. One of the key features of Spring Boot is auto-configuration, which automatically configures your Spring application based on the dependencies you have added to the project. This eliminates the need for extensive boilerplate configuration, allowing developers to start working on the application logic almost immediately.

Additionally, Spring Boot includes embedded servers such as Tomcat i.e our current used server. These embedded servers allow developers to package their applications as executable JAR files, simplifying the deployment process by eliminating the need to deploy WAR files to an external server. This feature is particularly beneficial for continuous integration and continuous deployment (CI/CD) pipelines, ensuring consistent and repeatable deployments.

Spring Boot also offers a variety of starter templates known as Spring Boot Starters, which provide convenient dependency descriptors for different functionalities like web, data access, security, and messaging. These starters allow developers to easily include necessary dependencies and configurations, speeding up the setup process for new projects. (ref)

An integral part of Spring Boot is the Actuator module, which provides a set of production-ready features for monitoring and managing applications. Actuator includes endpoints for health checks, metrics, and application information, which are essential for maintaining the health and performance of production systems. These monitoring and management capabilities help in identifying and resolving issues quickly, ensuring the high availability and reliability of the application.The combination of Spring and Spring Boot in our HealthApp project has allowed us to streamline development processes, making our application robust, scalable, and maintainable.(ref)

4.2 Layered Architecture

Layered architecture is a widely used architectural pattern in software development, and it is especially effective in the context of Spring Boot applications. This architecture divides an application into separate layers, each with specific responsibilities, facilitating better organization, maintainability, and scalability. In this article, we will explore the key concepts, benefits, and implementation strategies of layered architecture in Spring Boot applications.

Understanding Layered Architecture

Layered architecture is an approach to software design that organizes the system into different layers, each layer with a specific role and responsibility. This separation of concerns allows for a more modular and manageable system. The typical layers in a Spring Boot application include:

1. **Presentation Layer**: This layer is responsible for handling user interactions and presenting information. It is the topmost layer that interacts directly with the end-users.
2. **Service Layer**: This layer contains the business logic of the application. It acts as an intermediary between the presentation and data access layers.
3. **Data Access Layer**: This layer is responsible for data persistence and retrieval. It interacts with the database or any other data storage mechanism.
4. **Database Layer**: This layer manages the actual data storage and retrieval from the underlying database system.

Each of these layers serves a distinct purpose and communicates with adjacent layers, creating a structured and modular application.

The Presentation Layer

The Presentation Layer is the user-facing part of the application. In a Spring Boot application, this layer typically involves components like controllers, views, and front-end interfaces. The main responsibility of this layer is to handle user inputs, validate them, and send them to the service layer for processing. It then receives the processed data and presents it back to the user.

In a web application, the presentation layer often uses Spring MVC (Model-View-Controller) framework to manage HTTP requests and responses. The controller components handle the incoming requests, process them with the help of services, and return the appropriate views or data responses.

The Service Layer

The Service Layer, also known as the business logic layer, is where the core functionality of the application resides. This layer processes the data received from the presentation layer, applies business rules, and makes decisions based on those rules. It ensures that the application behaves according to the business requirements.

Services in this layer are typically designed to be reusable and stateless. This means they do not hold any state between calls, making them easier to test and maintain. The service layer interacts with the data access layer to fetch and persist data as needed.

The Data Access Layer

The Data Access Layer is responsible for interacting with the data storage mechanism, which is usually a relational database. This layer abstracts the complexity of database interactions, providing a simple interface for the service layer to perform CRUD (Create, Read, Update, Delete) operations.

In Spring Boot, this layer is often implemented using Spring Data JPA (Java Persistence API), which provides a repository abstraction over the database. This allows developers to focus on writing business logic without worrying about the underlying database interactions.

The Database Layer

The Database Layer is the foundational layer that manages the actual data storage and retrieval operations. This layer typically involves the database management system (DBMS) itself, where the physical data resides. It is responsible for ensuring data integrity, performance, and storage management.

The database layer interacts with the data access layer through database queries and transactions. In a Spring Boot application, this interaction is facilitated by ORM (Object-Relational Mapping) tools like Hibernate, which translate high-level data access commands into SQL queries understood by the database.

Benefits of Layered Architecture

Layered architecture offers several benefits that make it a popular choice for developing Spring Boot applications:

1. **Separation of Concerns**: By dividing the application into distinct layers, each layer focuses on a specific aspect of the application, leading to a cleaner and more organized codebase.
2. **Modularity**: Each layer can be developed, tested, and maintained independently. This modularity enhances the ability to manage and scale the application.
3. **Reusability**: Components in the service layer can be reused across different parts of the application, reducing redundancy and promoting code reuse.
4. **Maintainability**: The clear separation of responsibilities makes it easier to locate and fix issues, improving the maintainability of the application.
5. **Testability**: With distinct layers, unit testing and integration testing become more straightforward, as each layer can be tested in isolation.

Implementation Strategies

To implement layered architecture in a Spring Boot application, consider the following strategies:

1. **Define Clear Interfaces**: Each layer should expose well-defined interfaces to the layers above it. This promotes loose coupling and makes it easier to swap out implementations if needed.

2. **Use Dependency Injection**: Spring Boot's dependency injection framework allows for the easy management of dependencies between layers. This reduces tight coupling and enhances testability.
3. **Keep Layers Independent**: Ensure that layers do not depend on concrete implementations of other layers. Instead, depend on abstractions to maintain independence.
4. **Follow Best Practices**: Adhere to best practices like DRY (Don't Repeat Yourself), SOLID principles, and proper exception handling to maintain a clean and robust codebase.

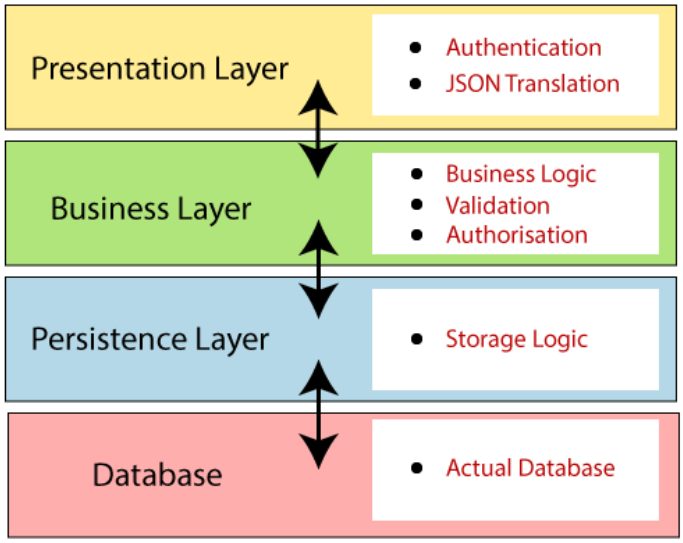


Figure 5: Layered Architecture

Our HealthApp employs a layered architecture to ensure a clean separation of concerns, improve maintainability, and enhance testability. This architectural approach divides the application into distinct layers, each with specific responsibilities. The primary layers in a Spring Boot application are the Presentation Layer (referred to as api package in our project structure), Business Layer (referred to as application package in our project structure), Persistence Layer (referred to as repository package in our project structure), and Database Layer. Each layer communicates with the adjacent layers, following a hierarchical structure that promotes organized and modular development. We have also separated some of the layers to their own packages like: remote, security, scheduledTasks, configuration, drools.

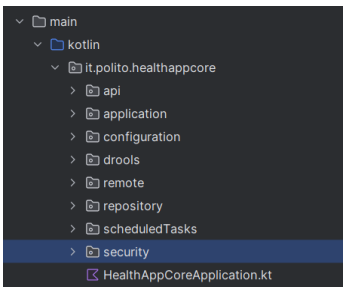


Figure 6 HealthApp layers

4.2.1. API Layer

The API Layer is the topmost layer of the application, responsible for handling HTTP requests and responses. It serves as the interface between the user and the application, converting user actions into appropriate service requests. This layer includes components such as controllers, RESTful endpoints, Data Transfer Objects (DTOs) representing requests and responses, and validators for data validation. In implementing this layer, I adhered to REST API standards [19], representing resources such as questionnaires, questionnaire templates, doctors, and patients, and performing CRUD operations on these resources. For instance, the endpoint used to manage patients has the base URL "/api/patients." A GET request to this URL returns a list of all patients, a POST request creates a new patient, a PUT request updates an existing patient, and a DELETE request removes a patient. A GET request with an ID (e.g., "/api/patients/{id}") returns the specified patient. This logic is consistently applied to other resources in our HealthApp.

REST APIs (Representational State Transfer Application Programming Interfaces) facilitate seamless communication between diverse systems. Introduced by Roy Fielding in 2000, REST is an architectural style based on HTTP, emphasizing scalability, performance, and ease of modification. Key principles of REST include statelessness, where each request contains all necessary information; a client-server architecture, separating UI and data storage; cacheability to improve performance; a layered system for modularity; and a uniform interface for consistent interactions. These principles ensure RESTful systems are flexible, scalable, and robust, making them ideal for the evolving nature of web applications.

The API layer is divided into 3 main pieces:

- **Controllers**: Manage incoming HTTP requests, map them to appropriate service calls, and return responses. These following controllers use @RestController annotation to define request mappings. The methods of these classes are annotated with different kinds of annotations like the crud annotations: @GetMapping, @PostMapping, @PutMapping, @PatchMapping and security annotations like @PreAuthorize, we will see more about that in later sections.

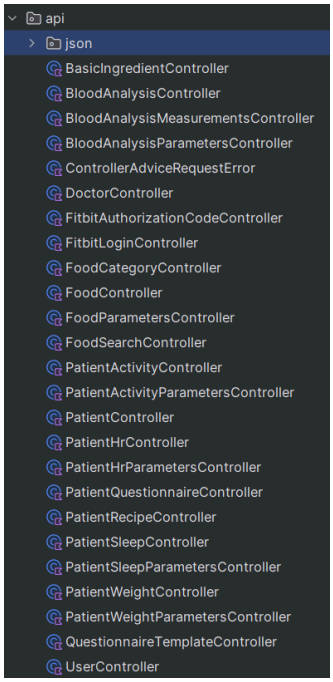


Figure 7: Controllers

- **Json**: Contain all the contract (Data classes) that represent the requests and responses

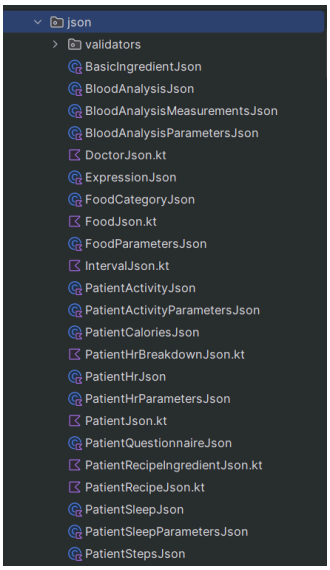


Figure 8: Json Data classes

- **Validators**: (Maybe I could also explain spring boot filters that are used to achieve these)This Package contain the validators that validate all the inputs we receive in the request bodies before calling our services

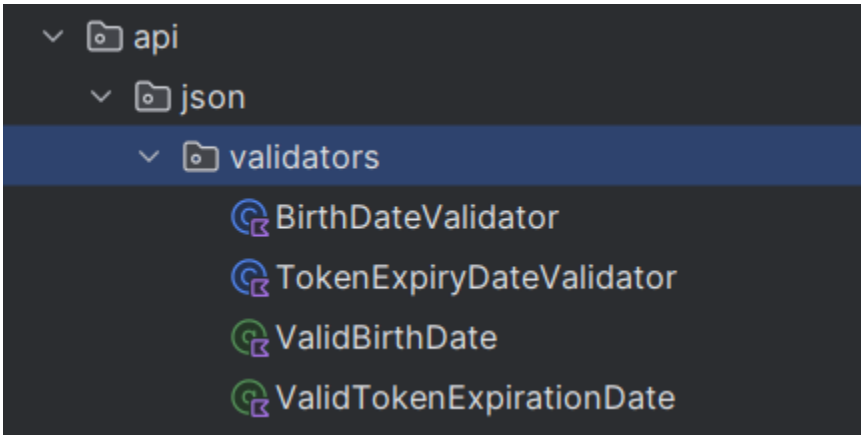


Figure 9: Controller Validators

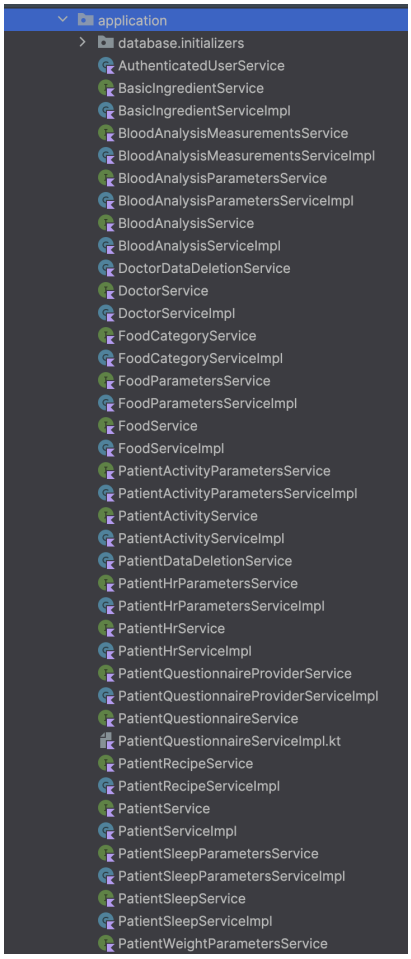
4.2.2. Application Layer

The Application Layer, also known as the Service Layer, contains the core business logic of the application. This layer processes data received from the Presentation Layer, applies business rules, and coordinates between the other layers. It ensures that the application adheres to its business requirements and provides the necessary functionality. Classes within the application layer are usually called services.

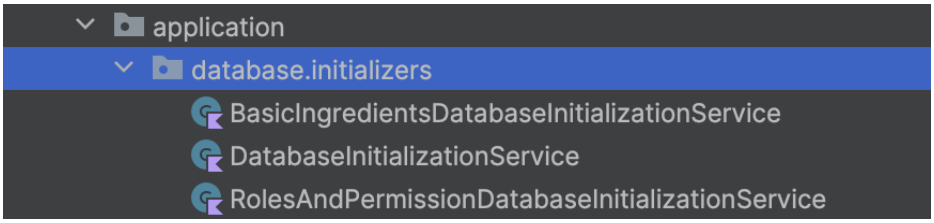
Services: Service classes, annotated with @Service, encapsulate the business logic. They interact with repositories to fetch and persist data, apply business rules, and handle transactional processes. By centralizing business logic in the Service Layer, the application maintains a clear separation between presentation concerns and business rules, making the code easier to manage and test.

The application layer is divided into two main packages:

- **Base package**: This package contains all the core business logic for all of our features (Authentication, CRUD operations for our entities, Patient data manipulation, Questionnares)



- Database.initializer package: That was used initially to initialize our database with predefined data. These spring beans run every time our application is restarted. This was a bit problematic so we opted for a better solution i.e database versioning tools in our case we use liquibase.



4.2.3. Persistence Layer

The Persistence Layer handles data access logic, abstracting the underlying data storage and retrieval mechanisms. It acts as a bridge between the Business Layer and the Database Layer, translating business objects to and from database entities.

Repositories: In Spring Boot, repositories are interfaces or classes annotated with @Repository. They provide CRUD (Create, Read, Update, Delete) operations and custom query methods using Spring Data JPA or other data access technologies. By using repositories, the application isolates database interactions, making it easier to switch data sources or modify data access logic without affecting the business logic.

Java Persistence API (JPA) is a powerful framework that simplifies the development of data access layers in Java applications. In the context of Spring Boot, JPA is often used in conjunction with Spring Data JPA, which provides a higher level of abstraction for data access. One of the most notable features of Spring Data JPA is its support for query methods, which allow developers to define data retrieval queries directly in repository interfaces without writing any implementation code. This article explores JPA query methods, their types, benefits, and best practices.

JPA query methods are part of the repository interfaces in Spring Data JPA. These methods are automatically implemented by Spring Data based on the method names and their parameters. This approach leverages the power of convention over configuration, allowing developers to define complex queries with minimal boilerplate code.

There are three main types of query methods in Spring Data JPA:

1. **Derived Query Methods**: These methods derive queries from the method name. Spring Data JPA parses the method name and constructs the query accordingly. For example, a method named findByUsername will generate a query that retrieves entities based on the username field.
2. **@Query Annotation**: For more complex queries that cannot be expressed through method names, the @Query annotation can be used. This annotation allows developers to write JPQL (Java Persistence Query Language) or native SQL queries directly in the repository interface.
3. **Query by Example (QBE)**: This approach allows for dynamic query generation based on the properties of a given example entity. It provides a flexible way to search for entities that match the example entity's properties.

Derived Query Methods

Derived query methods are the simplest form of query methods. They follow a specific naming convention that Spring Data JPA interprets to create the query. The naming convention typically includes the entity's field names and query keywords like findBy, readBy, countBy, and deleteBy.

For instance, a method named findByLastName will generate a query to find entities with a matching lastName field. Developers can also combine multiple fields and use keywords like And, Or, Between, LessThan, GreaterThan, etc., to create more specific queries.

@Query Annotation

When derived query methods are insufficient for complex queries, the @Query annotation provides a powerful alternative. This annotation allows developers to write custom JPQL or native SQL queries directly within the repository interface.

For example, a method annotated with @Query("SELECT u FROM User u WHERE u.email = ?1") will generate a query to find users by their email address. The @Query annotation can also be used with named parameters, allowing for more readable and maintainable queries.

The @Query annotation is particularly useful for performing joins, aggregations, and other complex operations that are not easily expressed through method names. It offers a high degree of flexibility and control over the generated queries.

Query by Example (QBE)

Query by Example is a flexible and type-safe way to perform queries based on the properties of a given example entity. This approach is useful for dynamic queries where the criteria are not known at compile time.

To use QBE, developers create an example entity with the desired properties set and pass it to the repository's findAll(Example<S> example) method. Spring Data JPA will generate a query that matches entities based on the example entity's non-null properties.

QBE is particularly advantageous for applications that require dynamic search functionalities, such as search filters in web applications. It provides a clean and intuitive way to construct queries without writing explicit JPQL or SQL.

Benefits of JPA Query Methods

JPA query methods offer several benefits that enhance the development experience and efficiency:

1. **Reduced Boilerplate Code**: By deriving queries from method names and using annotations, developers can avoid writing repetitive and boilerplate code, leading to a cleaner codebase.
2. **Improved Readability**: Query methods are self-explanatory and easy to read, making the code more understandable and maintainable.
3. **Rapid Development**: With automatic query generation, developers can quickly implement data access layers, accelerating the overall development process.
4. **Flexibility**: The combination of derived methods, @Query annotations, and QBE provides a versatile toolkit for handling various querying needs.
5. **Type Safety**: JPA query methods leverage Java's type system, ensuring compile-time type checking and reducing runtime errors.

Best Practices

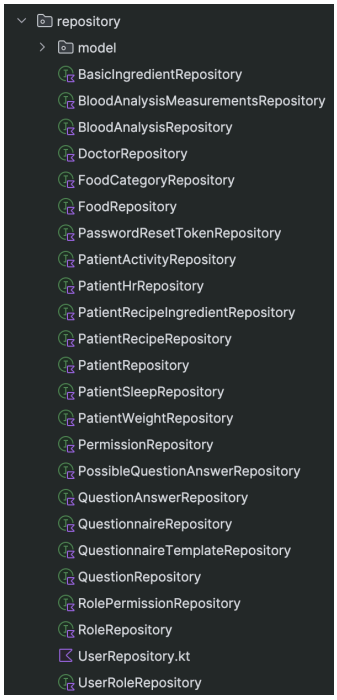
To maximize the effectiveness of JPA query methods, consider the following best practices:

1. **Follow Naming Conventions**: Adhere to the naming conventions for derived query methods to ensure they are correctly parsed and generated by Spring Data JPA.
2. **Use @Query for Complex Queries**: For queries that involve multiple joins, aggregations, or other complex operations, use the @Query annotation to write custom queries.

3. **Optimize Performance**: Be mindful of the performance implications of the generated queries. Ensure that queries are optimized and use appropriate indexes to avoid performance bottlenecks.
4. **Document Queries**: Provide documentation and comments for complex queries to enhance maintainability and knowledge sharing among team members.

In our Health App we extensively used Derived Query Methods which helped us query our database in an easy and yet optimized way.

- Repository package: contains all the CRUD JPA repositories to interface with our database.



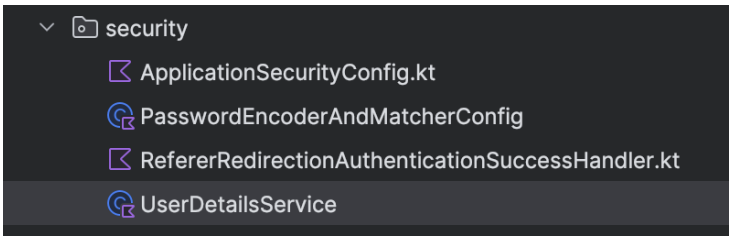
4.2.4. Database Layer

The Database Layer is the lowest layer, consisting of the actual database and the data access code that directly interacts with it. This layer is responsible for managing data persistence, executing SQL queries, and maintaining the integrity and consistency of stored data.

- Database Configuration: In a Spring Boot application, database configuration is typically handled through properties files (application.properties or application.yml). This configuration includes setting up data sources, defining connection pools, and specifying ORM (Object-Relational Mapping) settings.

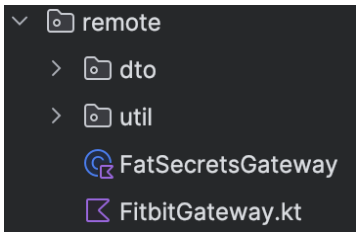
4.2.5 Security Layer

The security layer is one of the most important and critical layers. It is responsible for both authentication and authorization of our user.



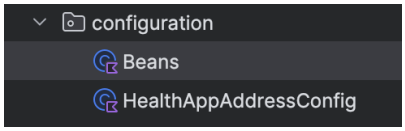
4.2.6 Remote Layer

The remote layer (often called the service or client layer) typically includes components that interface with external entities. These components can be REST clients, also referred to as gateways. This layer is responsible for making HTTP calls to external APIs, handling responses, and transforming data as necessary



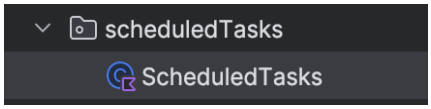
4.2.7 Configuration Layer

The configuration package is typically used to manage application-specific configurations and settings that are necessary for the application to function correctly. This package often contains classes annotated with @Configuration, which can define beans, external service configurations, security settings, data sources, and more.



4.2.8 Scheduled Tasks Layer

The scheduled tasks package is used to manage the configuration and implementation of scheduled tasks. These tasks are often used for running periodic jobs such as data cleanup, sending notifications, or performing other routine maintenance tasks. The package typically contains classes annotated with @Scheduled and may also include configuration for task scheduling.



4.3. Hexagonal Architecture

Hexagonal architecture, also known as the Ports and Adapters pattern, is a design paradigm that emphasizes the decoupling of an application's core logic from external concerns such as databases, user interfaces, and other external systems. This approach allows developers to create applications that are more maintainable, testable, and adaptable to change [11].

In the context of a Spring Boot application, hexagonal architecture can be particularly beneficial due to Spring Boot's extensive support for modular development and dependency injection, which naturally aligns with the principles of hexagonal architecture.

In our HealthApp project, we aimed to follow these principles meticulously by designing our application in such a way that each service has a corresponding interface, ensuring that our services do not rely on any external dependencies, thus maintaining a clear separation between the business logic and the infrastructure.

For each service, we defined an interface in the application layer, such as PatientService, FoodService, and BloodAnalysisService, which declare the necessary methods for the business operations without including implementation details. The actual implementations of these services, like PatientServiceImpl, FoodServiceImpl, and BloodAnalysisServiceImpl, reside in separate classes, ensuring that the core business logic remains isolated and can be easily tested independently of external systems.

Whenever any part of the application logic needs to use an external system, such as a REST client or a gateway from the remote package, we create an interface for that gateway within the application package. This structure allows us to inject the implementations using Spring Boot's dependency injection, keeping the application layer clean and focused solely on the business logic.

This flexibility is particularly beneficial for testing, as we can easily replace real implementations with mocks or stubs during unit tests. By leveraging Spring Boot's support for dependency injection and modular development, annotations like @Service, @Repository, and @Component help define the roles of various classes, while @Autowired or constructor injection can be used to inject dependencies.

This setup ensures that the core logic interacts with abstractions rather than concrete implementations, aligning perfectly with hexagonal architecture's emphasis on decoupling.

This approach enhances maintainability and adaptability, allowing new requirements to be integrated more smoothly. If we need to switch to a different external health data provider, we only need to implement the existing interface for the new provider without altering the core business logic, making the application more adaptable to change and easier to extend.

By adhering to hexagonal architecture, we ensure that our application is more robust and resilient to changes in external systems, as the business logic is shielded from direct dependencies on external APIs, database schemas, or other infrastructure details. Implementing hexagonal architecture in our HealthApp project has provided numerous benefits, including improved testability, maintainability, and flexibility, creating a robust and adaptable application structure.

5. Database

5.1. Database Selection

For the HealthApp project, we initially utilized H2 as our database management system (DBMS) [17]. H2 is an open-source, in-memory database that is commonly used for development and testing purposes due to its lightweight nature and ease of setup. However, as the project evolved and the need for a more robust and scalable solution became apparent, we transitioned to PostgreSQL [18]. PostgreSQL was chosen for its advanced features, scalability, and strong support for ACID transactions, ensuring data integrity and consistency, which are critical for healthcare-related applications.

5.1.1. Transition from H2 to PostgreSQL

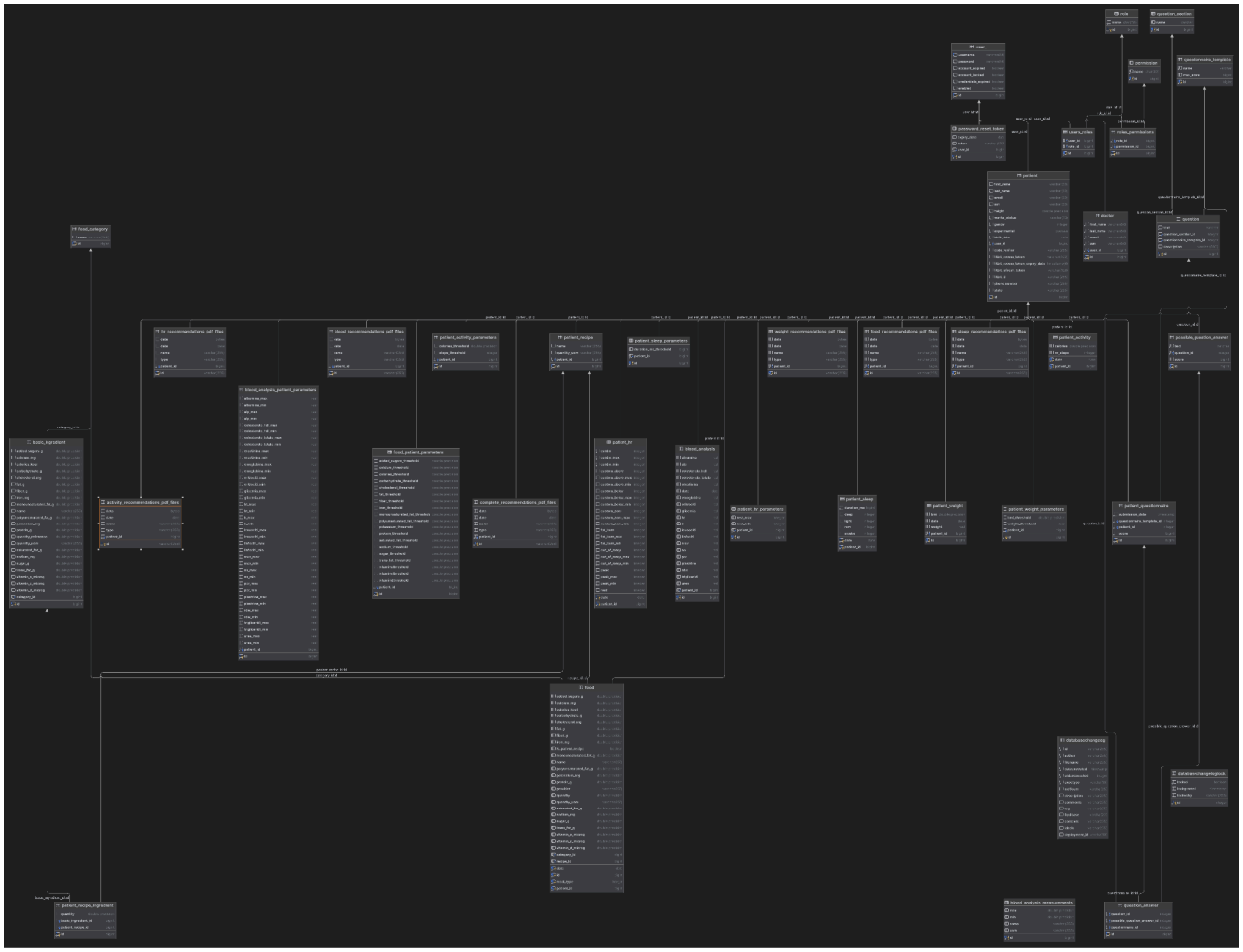
The decision to start with H2 was driven by the need for a quick and efficient setup during the initial development and testing phases. H2's in-memory capabilities allowed for rapid prototyping and immediate feedback on changes, significantly speeding up the development process. However, as the application's complexity grew, the limitations of H2 became more apparent. Specifically, H2's in-memory nature posed challenges for persistent data storage and handling large datasets, which are essential for a production-level healthcare application.

Transitioning to PostgreSQL involved several key steps:

1. **Schema Migration**: The database schema was exported from H2 and adapted to PostgreSQL's syntax and capabilities. This included adjusting data types and ensuring compatibility with PostgreSQL's advanced features.
2. **Data Migration**: Existing data within the H2 database was extracted and imported into PostgreSQL. This process was automated to minimize downtime and ensure data consistency.
3. **Configuration Changes**: The application's configuration was updated to connect to the PostgreSQL database instead of H2. This involved changes to the Spring Boot configuration files and ensuring that the necessary PostgreSQL drivers were included.

5.1.2. Database Schema

The database schema for HealthApp was meticulously designed to encapsulate the core functionalities of the system, focusing on the questionnaire and patient response aspects. The schema includes several entities. Each of these entities plays a crucial role in the application's overall structure and functionality.



The schema is quite extensive, so I will break it down into smaller sections for better clarity. For the parts I have worked on, we can divide the schema into three main segments: Patients and Doctors, User Authentication and Authorization, Questionnaires, and Patient Medical Records. I will explain each of these segments and the various relationships between the entities within them to illustrate how they are interconnected. I will start by discussing the primary entities, such as patients, doctors, and users. Following this, I will move on to the entities related to questionnaires and finally, address those associated with medical records.

5.1.2.1. Patient and Doctors related Entities Descriptions and Relationships

In our system, we have implemented a relational database to store and manage data for doctors and patients, along with their associated activities, sleep records, heart rates, recipes, and questionnaire responses. The database schema is designed using JPA (Java Persistence API) annotations to define the entity classes and their relationships. Below is a detailed description of the key entities and their relationships.

1. Doctor Entity

The Doctor entity represents medical professionals in the system. The entity is mapped to the doctor table in the database. Each doctor has a unique identifier (id), first name (firstName), last name (lastName), email address (email), and social security number (ssn). Additionally, each doctor is associated with a User entity, representing their login credentials.

Table: doctor

Column	Type	Description
id	Long	Primary key, auto-generated
first_name	String	Doctor's first name
last_name	String	Doctor's last name
email	String	Doctor's email address
ssn	String	Doctor's social security number
user_id	Long	Foreign key to the user table

Relationships:

- OneToOne relationship with the User entity.

2. Patient Entity

The Patient entity represents individuals receiving medical care. This entity is mapped to the patient table and contains various personal details such as the patient's first name (firstName), last name (lastName), phone number (phoneNumber), email address (email), social security number (ssn), height (height), marital status (maritalStatus), gender (gender), and birth date (birthDate). The entity also includes experimental status (experimental) and several fields related to Fitbit integration (e.g., fitbitUserId, fitbitAccessToken, etc.).

Table: patient

Column	Type	Description
id	Long	Primary key, auto-generated
first_name	String	Patient's first name

last_name	String	Patient's last name
phone_number	String	Patient's phone number
email	String	Patient's email address
ssn	String	Patient's social security number
height	Double	Patient's height
marital_status	String (Enum)	Patient's marital status
gender	String (Enum)	Patient's gender
experimental	Boolean	Indicates if the patient is part of an experimental group
birth_date	LocalDate	Patient's birth date
user_id	Long	Foreign key to the user table
fitbit_id	String	Fitbit user ID
code_verifier	String	Fitbit code verifier
state	String	Fitbit state
fitbit_access_token	String	Fitbit access token
fitbit_refresh_token	String	Fitbit refresh token
fitbit_access_token_expiry_date	ZonedDateTime	Fitbit access token expiry date

Relationships:

- OneToMany relationships with several entities:
    - PatientQuestionnaire
    - PatientActivity
    - PatientSleep
    - PatientHr
    - PatientRecipe
- OneToOne relationship with the User entity.

5.1.2.2. Authentication and Authorization Related Entities

1. User Entity

The User entity represents the authentication credentials for doctors and patients in the system. The entity is mapped to the user table.

Table: user

Column	Type	Description
id	Long	Primary key, auto-generated
username	String	Unique username
password	String	Encrypted Password
account_expired	Boolean	Indicates if the account is expired
account_locked	Boolean	Indicates if the account is locked
credentials_expired	Boolean	Indicates if the credentials are expired
enabled	Boolean	Indicates if the account is enabled

2. Role Entity

The Role entity defines various roles available in the system. The entity is mapped to the Role table and includes a list of permissions associated with each role.

Table: Role

Column	Type	Description
id	Long	Primary key, auto-generated
name	String (Enum)	Role name, e.g., PATIENT, DOCTOR, ADMIN

Relationships:

- OneToMany relationship with the RolePermission entity.

3. Permission Entity

The Permission entity defines various permissions that can be assigned to roles. The entity is mapped to the Permission table.

Table: Permission

Column	Type	Description
id	Long	Primary key, auto-generated
name	String (Enum)	Permission name, e.g., patient_questionnaire:read

4. UserRole Entity

The UserRole entity maps users to their roles. The entity is mapped to the users_roles table.

Table: users_roles

Column	Type	Description
id	Long	Primary key, auto-generated
user_id	Long	Foreign key to the user table
role_id	Long	Foreign key to the role table

5. RolePermission Entity

The RolePermission entity maps roles to their permissions. The entity is mapped to the roles_permissions table.

Table: roles_permissions

Column	Type	Description
id	Long	Primary key, auto-generated
role_id	Long	Foreign key to the role table
permission_id	Long	Foreign key to the permission table

5.1.2.3 Enumerations

MaritalStatus Enum: Represents the marital status of a patient with possible values:

- SINGLE
- MARRIED
- DIVORCED
- WIDOWED
- UNSPECIFIED

Gender Enum: Represents the gender of a patient with possible values:

- MALE
- FEMALE

ApplicationUserRole Enum: Defines the roles that can be assigned to users:

- PATIENT
- DOCTOR
- ADMIN

ApplicationUserPermission Enum: Defines the permissions that can be assigned to roles. Each permission has a string value that describes the action and the entity it applies to. Example permissions include:

- patient_questionnaire:read
- patient_questionnaire:write
- questionnaire_template:read
- questionnaire_template:write

5.1.2.4 Entity Relationships

- Each Doctor and Patient is associated with a User entity using a OneToOne relationship, where the User entity holds authentication credentials for the respective doctor or patient.
- The UserRole entity maps users to their roles, facilitating a ManyToMany relationship between the User and Role entities.
- The RolePermission entity maps roles to their permissions, facilitating a ManyToMany relationship between the Role and Permission entities.
- The Patient entity has OneToMany relationships with several other entities representing different aspects of the patient's data, such as activities, sleep records, heart rate measurements, recipes, and questionnaire responses.

5.1.2.5. Questionnaire Related Entity Descriptions and Relationships

1. Question Entity

The Question entity represents individual questions within a questionnaire. The entity is mapped to the question table in the database. Each question has a unique identifier (id), text (text), an optional description (description), and is associated with a QuestionSection and QuestionnaireTemplate. It can also have multiple possible answers (PossibleQuestionAnswer).

Table: question

Column	Type	Description
id	Long	Primary key, auto-generated
text	String	The text of the question
description	String	Optional description of the question (max 500)
question_section_id	Long	Foreign key to QuestionSection
questionnaire_template_id	Long	Foreign key to QuestionnaireTemplate

Relationships:

- ManyToOne relationship with QuestionSection
- OneToMany relationship with PossibleQuestionAnswer
- ManyToOne relationship with QuestionnaireTemplate

2. PossibleQuestionAnswer Entity

The PossibleQuestionAnswer entity represents the possible answers to a question. The entity is mapped to the possible_question_answer table in the database. Each possible answer has a unique identifier (id), text (text), and an optional score (score).

Table: possible_question_answer

Column	Type	Description
id	Long	Primary key, auto-generated
text	String	The text of the possible answer
score	Long	Optional score for the answer
question_id	Long	Foreign key to Question

Relationships:

- ManyToOne relationship with Question

3. QuestionAnswer Entity

The QuestionAnswer entity represents the answers given by patients to questions in a questionnaire. The entity is mapped to the question_answer table in the database. Each answer has a unique identifier (id), references to the question (question) and the chosen possible answer (possibleQuestionAnswer).

Table: question_answer

Column	Type	Description
id	Long	Primary key, auto-generated
question_id	Long	Foreign key to Question
possible_question_answer_id	Long	Foreign key to PossibleQuestionAnswer
questionnaire_id	Long	Foreign key to PatientQuestionnaire

Relationships:

- OneToOne relationship with Question
- OneToOne relationship with PossibleQuestionAnswer
- ManyToOne relationship with PatientQuestionnaire

4. QuestionnaireTemplate Entity

The QuestionnaireTemplate entity represents a template for a questionnaire. The entity is mapped to the questionnaire_template table in the database. Each template has a unique identifier (id), a name (name), and a maximum score (maxScore). It also has a list of questions (questions).

Table: questionnaire_template

Column	Type	Description
id	Long	Primary key, auto-generated
name	String	Name of the questionnaire
max_score	Long	Maximum score for the questionnaire

questions	List<Questions>	List of associated questions

Relationships:

- OneToMany relationship with Question

5. QuestionSection Entity

The QuestionSection entity represents sections within a questionnaire, grouping related questions together. The entity is mapped to the question_section table in the database. Each section has a unique identifier (id) and a name (name). It also has a list of questions (questions).

Table: question_section

Column	Type	Description
id	Long	Primary key, auto-generated
name	String	Name of the section
questions	List<Questions>	List of associated questions

Relationships:

- OneToMany relationship with Question

6. PatientQuestionnaire Entity

The PatientQuestionnaire entity represents a completed questionnaire by a patient. The entity is mapped to the patient_questionnaire table in the database. Each patient questionnaire has a unique identifier (id), a score (score), a submission date (submissionDate), and is associated with a QuestionnaireTemplate and a Patient. It also has a list of question answers (questionAnswers).

Table: patient_questionnaire

Column	Type	Description
id	Long	Primary key, auto-generated
score	Long	Score obtained by the patient
submission_date	Timestamp	Date when the questionnaire was submitted

questionnaire_template_id	Long	Foreign key to QuestionnaireTemplate
patient_id	Long	Foreign key to Patient
questionAnswers	List<QuestionAnswer>	List<QuestionAnswer>

Relationships:

- ManyToOne relationship with QuestionnaireTemplate
- ManyToOne relationship with Patient
- OneToMany relationship with QuestionAnswer

5.1.2.6. Medical Records Related Entities Description and Relationships

1. BloodAnalysis Entity

The BloodAnalysis entity represents blood analysis records for patients. The entity is mapped to the blood_analysis table in the database. Each blood analysis record has a unique identifier (id), various blood analysis measurements, a date (date), and is associated with a Patient.

Table: blood_analysis

Column	Type	Description
id	Long	Description
erythrocytes	Float	Measurement of erythrocytes
hemoglobin	Float	Measurement of hemoglobin
mean corpuscular volume	Float	Measurement of mean corpuscular volume
hematocrit	Float	Measurement of hematocrit
leukocytes	Float	Measurement of leukocytes
platelets	Float	Measurement of platelets
blood glucose	Float	Measurement of blood glucose
urea	Float	Measurement of urea
sodium	Float	Measurement of sodium

potassium	Float	Measurement of potassium
creatinine	Float	Measurement of creatinine
total cholesterol	Float	Measurement of total cholesterol
HDL cholesterol	Float	Measurement of HDL cholesterol
triglycerides	Float	Measurement of triglycerides
C-reactive protein	Float	Measurement of C-reactive protein
albumin	Float	Measurement of albumin
lymphocytes	Float	Measurement of lymphocytes
red cell distribution width	Float	Measurement of red cell distribution width
red cell distribution width	Float	red cell distribution width
date	LocalDate	Date of the blood analysis
patient_id	Long	Foreign key to Patient

Relationships:

- ManyToOne relationship with Patient

2. BloodAnalysisMeasurements Entity

The BloodAnalysisMeasurements entity represents the measurement details for various blood analysis parameters. The entity is mapped to the blood_analysis_measurements table in the database.

Table: blood_analysis_measurements

Column	Type	Description
id	Long	Primary key, auto-generated
name	String	Name of the measurement
uom	String	Unit of measurement (optional)
min	Double	Minimum value (optional)
max	Double	Maximum value (optional)

3. PatientActivity Entity

The PatientActivity entity represents the physical activities of patients. The entity is mapped to the patient_activity table in the database. It uses a composite key (PatientIDDatePK) consisting of patient ID and date.

Table: patient_activity

Column	Type	Description
id	Composite Key	Composite key (patient_id and date)
nr_steps	Int	Number of steps
calories	Double	Calories burned
patient_id	Long	Foreign key to `Patient`

Relationships:

- ManyToOne relationship with Patient

4. PatientHr Entity

The PatientHr entity represents heart rate data for patients. The entity is mapped to the patient_hr table in the database. It uses a composite key (PatientIDDatePK) consisting of patient ID and date.

Table: patient_hr

Column	Type	Description
id	Composite Key	Composite key (patient_id and date)
rest	Int	Resting heart rate
custom_below	Int	Time below custom zone (in minutes)
custom_below_min	Int	Lower limit for custom below zone
custom_below_max	Int	Upper limit for custom below zone
custom_zone	Int	Time in custom zone (in minutes)
custom_zone_min	Int	Lower limit for custom zone

custom_zone_max	Int	Upper limit for custom zone
custom_above	Int	Time above custom zone (in minutes)
custom_above_min	Int	Lower limit for custom above zone
custom_above_max	Int	Upper limit for custom above zone
out_of_range	Int	Time out of range (in minutes)
out_of_range_min	Int	Lower limit for out of range zone
out_of_range_max	Int	Upper limit for out of range zone
fat_burn	Int	Time in fat burn zone (in minutes)
fat_burn_min	Int	Lower limit for fat burn zone
fat_burn_max	Int	Upper limit for fat burn zone
cardio	Int	Time in cardio zone (in minutes)
cardio_min	Int	Lower limit for cardio zone
cardio_max	Int	Upper limit for cardio zone
peak	Int	Time in peak zone (in minutes)
peak_min	Int	Lower limit for peak zone
peak_max	Int	Upper limit for peak zone

Relationships:

- ManyToOne relationship with Patient

5. PatientSleep Entity

The PatientSleep entity represents sleep data for patients. The entity is mapped to the patient_sleep table in the database. It uses a composite key (PatientIDDatePK) consisting of patient ID and date.

Table: patient_sleep

Column	Type	Description
id	Composite Key	Composite key (patient_id and date)
duration_ms	Long	Duration of sleep in milliseconds
rem	Int	Minutes of REM sleep
deep	Int	Minutes of deep sleep
light	Int	Minutes of light sleep

awake	Int	Minutes awake
patient_id	Long	Foreign key to Patient

Relationships:

- ManyToOne relationship with Patient

6. PatientWeight Entity

The PatientWeight entity represents weight data for patients. The entity is mapped to the patient_weight table in the database.

Table: patient_weight

Column	Type	Description
id	Long	Primary key, auto-generated
weight	Float	Weight of the patient
bmi	Double	Body Mass Index of the patient
date	LocalDate	Date of the weight record
patient_id	Long	Foreign key to Patient

Relationships:

- ManyToOne relationship with Patient

5.1.3. Liquibase

In modern software development, managing database schemas and migrations efficiently is critical for maintaining consistency and ensuring the integrity of the data across different environments. One of the popular tools for database versioning and migration is Liquibase. Liquibase is an open-source library that helps track, manage, and apply database changes across various environments. In our project, Liquibase was used extensively to track database versions and insert initial data, which significantly streamlined our development process and mitigated numerous issues related to manual database management.

5.1.3.1 Why Liquibase?

1. Version Control for Databases

Liquibase enables version control for databases, similar to how Git provides version control for code. This feature allows developers to track changes, collaborate more effectively, and ensure that all environments (development, testing, production) are in sync with the latest schema changes.

2. Flexible Change Types

Liquibase supports a variety of change types, including SQL, XML, YAML, and JSON. This flexibility allows teams to define database changes in the format that best suits their workflow. In our project, we primarily used XML for its readability and ease of integration with existing tools.

3. Rollback Support

One of the critical features of Liquibase is its robust support for rollbacks. It allows developers to define how to undo changes explicitly, ensuring that any applied changes can be reverted if needed. This capability is crucial for maintaining database integrity and quickly addressing issues that may arise after a deployment.

5.1.3.2. Tracking Database Versions

In our project, we used Liquibase to manage and apply schema changes through a series of changelog files. Each changelog file contains a set of changesets, which are individual units of change (e.g., adding a new table, modifying an existing column). Below is an example of a Liquibase changelog file used in our project:

```
databaseChangeLog:
- include:
    file: schema.yaml
    relativeToChangelogFile: true
```

5.1.3.3. Inserting Initial Data

One significant advantage of using Liquibase is its ability to handle initial data insertion efficiently. Instead of manually inserting data through application code, we defined data insertion operations in the Liquibase changelog files. This approach ensures that the initial data is consistent and can be easily managed across different environments.

In our project, we initially used a DatabaseInitializationService class to insert data programmatically. The order of method invocations in the initializeUsers() method was critical, as changing the order would break the initialization process due to the manual setting of foreign keys and dependencies:

```
@Service
class DatabaseInitializationService(
  private val userRoleRepository: UserRoleRepository,
  private val userRepository: UserRepository,
  // Other repositories...
) {
  @EventListener(ApplicationReadyEvent::class)
  fun initializeUsers() {
    // Commented out due to data persistence issues
    // insertAdminUser()
    // insertDoctors()
    // insertPatients()
    // insertUserRole()
  }
}
```

By moving to Liquibase, we eliminated the need for this complex and error-prone initialization logic. Instead, we defined the initial data directly in the changelog files, which Liquibase managed automatically. This change not only simplified our codebase but also resolved issues related to duplicate entries and initialization order dependencies.

5.1.3.4. Benefits Realized

1. Reduced Complexity and Improved Maintainability

Using Liquibase for database versioning and initial data insertion significantly reduced the complexity of our database initialization code. This simplification made the codebase more maintainable and less prone to errors related to manual data management.

2. Consistent Environments

Liquibase ensured that all environments (development, testing, production) were consistently updated with the latest schema changes and initial data. This consistency is critical for detecting and fixing issues early in the development cycle, thereby improving the overall stability of the application.

3. Enhanced Collaboration

Liquibase's version control capabilities improved collaboration among team members. Changes to the database schema were tracked, reviewed, and merged just like code changes, facilitating better communication and coordination within the team.

Liquibase proved to be an invaluable tool in our project for managing database versions and inserting initial data. By replacing manual database initialization code with Liquibase changelogs, we achieved a more maintainable, consistent, and collaborative approach to database management. This transition not only streamlined our development process but also resolved several issues related to data consistency and initialization order. As a result, Liquibase has become an essential part of our development toolkit, ensuring that our database evolves smoothly alongside our application.

6. API Implementation

This section provides a comprehensive overview of the various API endpoints implemented in the Health App Core. These endpoints are categorized into three main areas: Doctor, Patient, and User Management; Questionnaire Templates and Patient Questionnaires; and Medical Records including Patient Weight, Sleep, Heart Rate, Activity, and Blood Analysis. Each endpoint is described with its functionality, expected arguments, and return values to ensure a clear understanding of how to interact with the API.

The implementation leverages Swagger UI for documentation, providing an interactive interface to explore and test the API endpoints. This documentation can be accessed at Swagger UI with the following url: http(s)://base_url/swagger-ui/index.html

6.1. Doctor, Patient, and User Management Endpoints

The Doctor, Patient, and User Management endpoints are designed to handle the core CRUD (Create, Read, Update, Delete) operations for managing doctor, patient, and user data within the Health App Core. Following REST API standards, these endpoints treat each entity as a resource, allowing clients to perform standard operations without maintaining any session state between requests, thus ensuring the application remains stateless. This approach simplifies interactions and enhances scalability by relying on HTTP methods to manage resources.

For the Doctor endpoints, operations include retrieving all doctors, fetching details of a specific doctor by ID, creating new doctor records, and deleting existing ones. Patient endpoints similarly allow for the creation, retrieval, update, and deletion of patient records, as well as managing Fitbit data updates. The User endpoints provide functionality for user retrieval by username, password resets, and password changes.

- **Get All Doctors**
  - **Endpoint:** GET /api/doctors
  - **Description:** Retrieves a list of all doctors.
  - **Arguments:** None.
  - **Returns:** A list of doctors.
- **Get a Doctor**
  - **Endpoint:** GET /api/doctors/{doctorId}
  - **Description:** Retrieves the details of a specific doctor by their ID.
  - **Arguments:**
    - doctorId (Path Variable): The ID of the doctor to retrieve.
  - **Returns:** A doctor object.
- **Create a Doctor**
  - **Endpoint:** POST /api/doctors
  - **Description:** Creates a new doctor record.
  - **Arguments:**
    - doctorJson (Request Body): JSON object containing details of the new doctor.
  - **Returns:** The created doctor object.

- **Delete a Doctor**
  - **Endpoint:** DELETE /api/doctors/{doctorId}
  - **Description:** Deletes a doctor by their ID.
  - **Arguments:**
    - doctorId (Path Variable): The ID of the doctor to delete.
  - **Returns:** No content.

Patient Endpoints

- **Create a Patient**
  - **Endpoint:** POST /api/patients
  - **Description:** Creates a new patient record.
  - **Arguments:**
    - patientJson (Request Body): JSON object containing details of the new patient.
  - **Returns:** The created patient object.
- **Update an Existing Patient**
  - **Endpoint:** PUT /api/patients/{patientId}
  - **Description:** Updates the details of an existing patient.
  - **Arguments:**
    - patientJson (Request Body): JSON object containing updated details of the patient.
    - patientId (Path Variable): The ID of the patient to update.
  - **Returns:** The updated patient object.
- **Get All Patients**
  - **Endpoint:** GET /api/patients
  - **Description:** Retrieves a list of all patients.
  - **Arguments:** None.
  - **Returns:** A list of patients.
- **Update Patient Fitbit Data**
  - **Endpoint:** PATCH /api/patients/{patientId}
  - **Description:** Updates the Fitbit data for a specific patient.
  - **Arguments:**
    - fitbitInfo (Request Body): JSON object containing the updated Fitbit data.
    - patientId (Path Variable): The ID of the patient to update.
  - **Returns:** The updated patient object.
- **Get a Patient by ID**
  - **Endpoint:** GET /api/patients/{patientId}
  - **Description:** Retrieves the details of a specific patient by their ID.
  - **Arguments:4**
    - patientId (Path Variable): The ID of the patient to retrieve.
  - **Returns:** A patient object.
- **Delete a Patient by ID**
  - **Endpoint:** DELETE /api/patients/{patientId}
  - **Description:** Deletes a patient by their ID.

- **Arguments:**
  - patientId (Path Variable): The ID of the patient to delete.
- **Returns:** No content.

User Endpoints

- **Get a User by Username**
  - **Endpoint:** GET /api/users/{username}
  - **Description:** Retrieves user details based on the username.
  - **Arguments:**
    - username (Path Variable): The username of the user to retrieve.
    - userType (Request Param): The role of the user (e.g., patient or doctor).
  - **Returns:** User details based on the role specified.
- **Reset User Password**
  - **Endpoint:** POST /api/users/resetPassword
  - **Description:** Sends an email to reset the user's password.
  - **Arguments:**
    - request (HttpServletRequest): The HTTP request object.
    - username (Request Param): The username of the user to reset the password for.
    - userType (Request Param): The role of the user.
  - **Returns:** A string indicating that an email has been sent.
- **Check Password Change Token**
  - **Endpoint:** GET /api/users/changePassword
  - **Description:** Validates the password change token and redirects to the appropriate page.
  - **Arguments:**
    - token (Request Param): The token to validate.
  - **Returns:** A redirect to the password change form or an error page.
- **Save User Password**
  - **Endpoint:** POST /api/users/savePassword
  - **Description:** Saves the new password for the user.
  - **Arguments:**
    - passwordChangeJson (Request Body): JSON object containing the new password details.
  - **Returns:** A JSON object indicating the result of the password change.
- **Authenticated Save User Password**
  - **Endpoint:** POST /api/users/authSavePassword
  - **Description:** Authenticated endpoint for saving the new password for the user.
  - **Arguments:**
    - passwordChangeLoggedJson (Request Body): JSON object containing the new password details and username.
  - **Returns:** A JSON object indicating the result of the password change.

6.2 Questionnaire Endpoints

The Questionnaire endpoints are designed to facilitate the management of questionnaire templates and patient-specific questionnaires within the Health App Core. These endpoints enable the creation, retrieval, updating, and deletion of questionnaire templates, which serve as blueprints for the questionnaires assigned to patients. Doctors can create these templates and assign them to patients, who then fill out the questionnaires. This allows doctors to easily track their patients' responses, making it easier to monitor their health and progress.

By adhering to REST API standards, these endpoints operate in a stateless manner, ensuring that each HTTP request from the client to the server contains all the information needed to understand and process the request. This stateless approach enhances the scalability and reliability of the application.

Utilizing these questionnaires, doctors can efficiently track their patients' answers, significantly simplifying the process of following up with each patient. This reduces the difficulty of remembering every patient's background and current state, thereby improving the quality of care and allowing doctors to focus on providing better medical advice and treatment.

Questionnaire Template Endpoints

- **Get All Questionnaire Templates**
  - **Endpoint:** GET /api/questionnaires/templates
  - **Description:** Retrieves a list of all questionnaire templates.
  - **Arguments:** None.
  - **Returns:** A list of questionnaire templates.
- **Get Questionnaire Template by ID**
  - **Endpoint:** GET /api/questionnaires/templates/{questionnaireTemplateId}
  - **Description:** Retrieves a specific questionnaire template by its ID.
  - **Arguments:**
    - questionnaireTemplateId (Path Variable): The ID of the questionnaire template to retrieve.
  - **Returns:** A questionnaire template object.
- **Create a Questionnaire Template**
  - **Endpoint:** POST /api/questionnaires/templates
  - **Description:** Creates a new questionnaire template.
  - **Arguments:**
    - questionnaireTemplateJson (Request Body): JSON object containing the details of the new questionnaire template.
  - **Returns:** The created questionnaire template object.

Patient Questionnaire Endpoints

- **Get a Single Patient Questionnaire**
  - **Endpoint:** GET /api/patients/{patientId}/questionnaires/{questionnaireId}
  - **Description:** Retrieves a specific patient questionnaire by its ID.
  - **Arguments:**

        - patientId (Path Variable): The ID of the patient.
        - questionnaireId (Path Variable): The ID of the questionnaire to retrieve.
    - **Returns:** A patient questionnaire object.
- **Get All Patient Questionnaires**
    - **Endpoint:** GET /api/patients/{patientId}/questionnaires
    - **Description:** Retrieves all questionnaires for a specific patient.
    - **Arguments:**
        - patientId (Path Variable): The ID of the patient.
    - **Returns:** A list of patient questionnaires.
- **Create a New Patient Questionnaire**
    - **Endpoint:** POST /api/patients/{patientId}/questionnaires
    - **Description:** Creates a new questionnaire for a specific patient.
    - **Arguments:**
        - patientId (Path Variable): The ID of the patient.
        - patientQuestionnaireJson (Request Body): JSON object containing the details of the new patient questionnaire.
    - **Returns:** The created patient questionnaire object.
- **Update a Single Patient Questionnaire**
    - **Endpoint:** PATCH /api/patients/{patientId}/questionnaires/{questionnaireId}
    - **Description:** Updates a specific patient questionnaire.
    - **Arguments:**
        - patientId (Path Variable): The ID of the patient.
        - questionnaireId (Path Variable): The ID of the questionnaire to update.
        - patientQuestionnaireJson (Request Body): JSON object containing the updated details of the patient questionnaire.
    - **Returns:** The updated patient questionnaire object.
- **Delete a Single Patient Questionnaire**
    - **Endpoint:** DELETE /api/patients/{patientId}/questionnaires/{questionnaireId}
    - **Description:** Deletes a specific patient questionnaire.
    - **Arguments:**
        - patientId (Path Variable): The ID of the patient.
        - questionnaireId (Path Variable): The ID of the questionnaire to delete.
    - **Returns:** No content.

6.3 Medical Records Endpoints

Regarding the operations chosen to be supported, medical analyses follow the classic CRUD paradigm with the four APIs: GET, POST, PUT, and DELETE. Concerning units of measurement and standard values, it is important to note that these are derived from universally established conventions. These conventions are not subject to significant changes, which is why, as mentioned earlier, this information can be considered immutable and stored directly in the database from the backend code. Instead of hardcoding these values, it is preferable to have a routine that initializes the database. This approach allows for modifications directly in the database without altering the server's source code, thereby avoiding temporary unavailability.

Patient Weight Endpoints

- **Create a New Patient Weight**
  - **Endpoint:** POST /api/patients/{patientId}/weights
  - **Description:** Creates a new weight record for a specific patient.
  - **Arguments:**
    - patientWeightJson (Request Body): JSON object containing the details of the new patient weight.
    - patientId (Path Variable): The ID of the patient.
  - **Returns:** The created patient weight object.
- **Get All Patient Weights**
  - **Endpoint:** GET /api/patients/{patientId}/weights
  - **Description:** Retrieves all weight records for a specific patient.
  - **Arguments:**
    - patientId (Path Variable): The ID of the patient.
  - **Returns:** A list of patient weight objects.
- **Update a Single Patient Weight**
  - **Endpoint:** PUT /api/patients/{patientId}/weights/{idW}
  - **Description:** Updates a specific weight record for a patient.
  - **Arguments:**
    - patientWeightJson (Request Body): JSON object containing the updated details of the patient weight.
    - patientId (Path Variable): The ID of the patient.
    - idW (Path Variable): The ID of the weight record to update.
  - **Returns:** The updated patient weight object.
- **Delete a Single Patient Weight**
  - **Endpoint:** DELETE /api/patients/{patientId}/weights/{idW}
  - **Description:** Deletes a specific weight record for a patient.
  - **Arguments:**
    - patientId (Path Variable): The ID of the patient.
    - idW (Path Variable): The ID of the weight record to delete.
  - **Returns:** No content.

Patient Sleep Endpoints

- **Get All Sleep Durations Between Two Dates**
  - **Endpoint:** GET /api/patients/{patientId}/sleep/duration
  - **Description:** Retrieves all sleep duration records for a specific patient between two dates.
  - **Arguments:**
    - patientId (Path Variable): The ID of the patient.
    - startDate (Request Param): The start date for the interval.
    - endDate (Request Param): The end date for the interval.
  - **Returns:** A list of patient sleep records.

Patient Heart Rate (HR) Endpoints

- **Get All HRs Between Two Dates**
  - **Endpoint:** GET /api/patients/{patientId}/hrs
  - **Description:** Retrieves all heart rate records for a specific patient between two dates.
  - **Arguments:**
    - patientId (Path Variable): The ID of the patient.
    - startDate (Request Param): The start date for the interval.
    - endDate (Request Param): The end date for the interval.
  - **Returns:** A JSON object containing heart rate breakdown information.

Patient Activity Endpoints

- **Get All Steps Between Two Dates**
  - **Endpoint:** GET /api/patients/{patientId}/activities/steps
  - **Description:** Retrieves all step records for a specific patient between two dates.
  - **Arguments:**
    - patientId (Path Variable): The ID of the patient.
    - startDate (Request Param): The start date for the interval.
    - endDate (Request Param): The end date for the interval.
  - **Returns:** A list of patient step records.
- **Get All Calories Between Two Dates**
  - **Endpoint:** GET /api/patients/{patientId}/activities/calories
  - **Description:** Retrieves all calorie records for a specific patient between two dates.
  - **Arguments:**
    - patientId (Path Variable): The ID of the patient.
    - startDate (Request Param): The start date for the interval.
    - endDate (Request Param): The end date for the interval.
  - **Returns:** A list of patient calorie records.

Blood Analysis Measurements Endpoints

- **Get All Medical Values, Units of Measurement, and Thresholds**
  - **Endpoint:** GET /api/bloodAnalysis/measurements
  - **Description:** Retrieves all medical values, units of measurement, and thresholds.
  - **Arguments:** None.

    - **Returns:** A list of blood analysis measurements.
- **Update Single Medical Value, Units of Measurement, and Threshold**
    - **Endpoint:** PUT /api/bloodAnalysis/measurements/{idMVUomThrs}
    - **Description:** Updates a specific medical value, unit of measurement, and threshold.
    - **Arguments:**
        - bloodAnalysisMeasurementsJson (Request Body): JSON object containing the updated details.
        - idMVUomThrs (Path Variable): The ID of the measurement to update.
    - **Returns:** The updated blood analysis measurement object.

Blood Analysis Endpoints

- **Get All Medical Values for a Patient**
    - **Endpoint:** GET /api/patients/{patientId}/bloodAnalysis
    - **Description:** Retrieves all blood analysis records for a specific patient.
    - **Arguments:**
        - patientId (Path Variable): The ID of the patient.
    - **Returns:** A list of blood analysis records.
- **Create Medical Values for a Patient**
    - **Endpoint:** POST /api/patients/{patientId}/bloodAnalysis
    - **Description:** Creates new blood analysis records for a specific patient.
    - **Arguments:**
        - bloodAnalysisJson (Request Body): JSON object containing the details of the new blood analysis.
        - patientId (Path Variable): The ID of the patient.
    - **Returns:** The created blood analysis record.
- **Update Single Medical Value for a Patient**
    - **Endpoint:** PUT /api/patients/{patientId}/bloodAnalysis/{idMV}
    - **Description:** Updates a specific blood analysis record for a patient.
    - **Arguments:**
        - bloodAnalysisJson (Request Body): JSON object containing the updated details.
        - patientId (Path Variable): The ID of the patient.
        - idMV (Path Variable): The ID of the blood analysis record to update.
    - **Returns:** The updated blood analysis record.
- **Delete Single Medical Value for a Patient**
    - **Endpoint:** DELETE /api/patients/{patientId}/bloodAnalysis/{idMV}
    - **Description:** Deletes a specific blood analysis record for a patient.
    - **Arguments:**
        - patientId (Path Variable): The ID of the patient.
        - idMV (Path Variable): The ID of the blood analysis record to delete.
    - **Returns:** No content.

Through the illustrated APIs, patients can record their medical analyses in the application, digitizing them if they are not already, and thus avoiding the risk of losing them as might happen with paper documents. Additionally, doctors can read and evaluate these analyses simply by opening the HealthApp web client, eliminating the need for patient appointments. Generally, results may already be digitized by the proprietary software of the laboratories where the samples are analyzed and made available to the doctor. However, if the laboratory belongs to another hospital complex or is private, it is challenging for the doctor to easily access them. This solution is therefore quite advantageous.

6.4 Fitbit Integration

In contrast to the previously illustrated endpoints, this paragraph focuses on the group of APIs related to data generated through Fitbit, covering three main areas: activity, health, and sleep. The essential element for collecting this information is the Fitbit tracker/wearable, which patients need to wear daily to ensure effective data production. The use case involves patients generating data on their heart rate, steps taken, calories burned, or sleep patterns. This data is stored on Fitbit's proprietary servers and retrieved by the HealthApp backend. The frontend then makes this information available to patients and doctors, presenting it in various styles and granularities.

Moreover, the integrated recommendations system which will be developed later will provide preliminary analysis feedback on a daily, weekly, or monthly basis. Given that the information is transmitted from the wearable to the Fitbit servers using the proprietary Fitbit app, it is necessary to retrieve this data and store it in the HealthApp database. This interaction mechanism exclusively involves the backend of the application, making the process completely transparent to the clients.

To achieve this, it is first necessary to establish a connection mechanism between the servers. Fitbit provides developers with a set of Web APIs for reading information produced by the wearable devices. To access these resources, it is required to register your application on the Fitbit Developer Console.

The registration process involves filling out a form where you provide key details about the application you intend to develop, a brief description of its main features, and request either read-only or read-and-write access to the data. A crucial decision at this stage is selecting the type of application you want to register. There are three options:

1. **Server**: This application type features a multi-tier architecture, including a server. Authentication and user data management are handled on the server side, making data accessible to each user.
2. **Client**: This type of application has a single-tier architecture. Authentication and user data management are handled on the client side, allowing each user to access their data.
3. **Personal**: A client or server application where the only accessible data is that of the developer who registers the app. This type does not support other users.

For HealthApp, a server-type application with read access was chosen. After completing the form, the registration must be approved by the Fitbit team. Once approved, you will have all the necessary parameters to use the APIs provided to developers. These parameters include:

- **OAuth 2.0 Client ID**: A unique identifier for the application used for authentication and API calls to Fitbit servers. Although the term "client" might be confusing, it refers to HealthApp acting as a client when reading data from Fitbit.
- **Client Secret**: A confidential key that HealthApp's backend must store securely to authenticate with Fitbit servers.
- **Redirect URL**: This URL is used when a HealthApp user connects with their Fitbit account. Users are redirected from HealthApp to Fitbit to log in and grant permissions. After permissions are granted, users are redirected back to the specified Redirect URL (an URL of HealthApp Web).
- **OAuth 2.0 Authorization URI**: The API endpoint that the client will call to start the process of linking a Fitbit account to HealthApp. This endpoint is: https://www.fitbit.com/oauth2/authorize
- **OAuth 2.0 Access/Refresh Token Request URI**: The API endpoint the server will call to request or refresh tokens to access the user's Fitbit data. This endpoint is: https://api.fitbit.com/oauth2/token

Data Access and Security

Before retrieving data produced by the tracker, the patient must authorize HealthApp to access their data. This authorization cannot be granted by storing the patient's Fitbit account email and password in the HealthApp database, as this is insecure and poses a risk to personal data. Instead, users can share only the necessary information for the application's operation without giving full access to their Fitbit account.

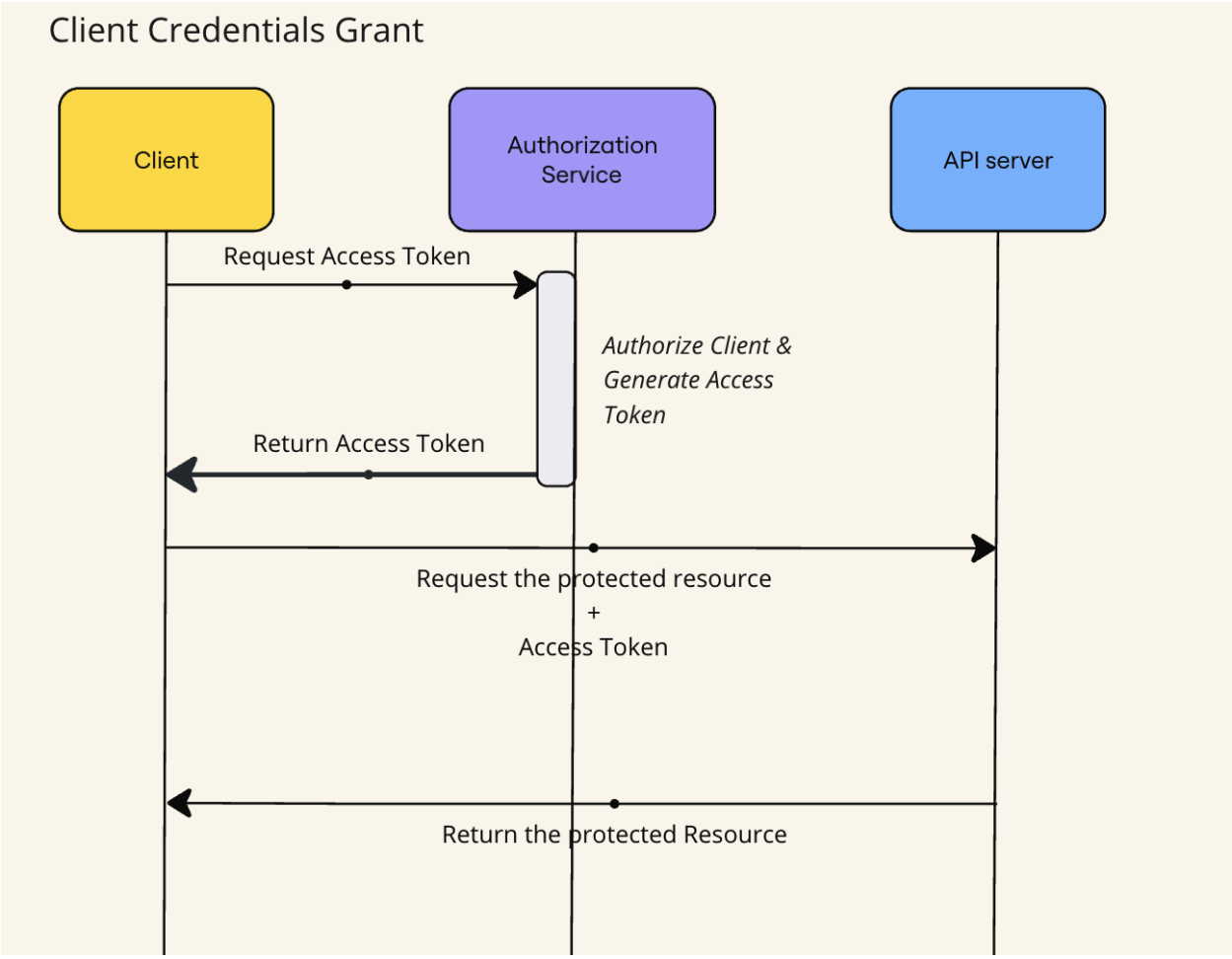
To address security concerns, an authentication procedure involving users, the HealthApp frontend and backend, and Fitbit servers is based on the OAuth 2.0 protocol.

OAuth 2.0 Authentication Protocol

The OAuth 2.0 authentication protocol can be described at a high level as follows:

1. The application sends an authorization request to the Authorization Server to access a protected resource.
2. The resource owner (the user) grants access.
3. The Authorization Server returns an Access Token used in all subsequent requests as an "identification badge." This token is time-limited and must be periodically refreshed using a Refresh Token.

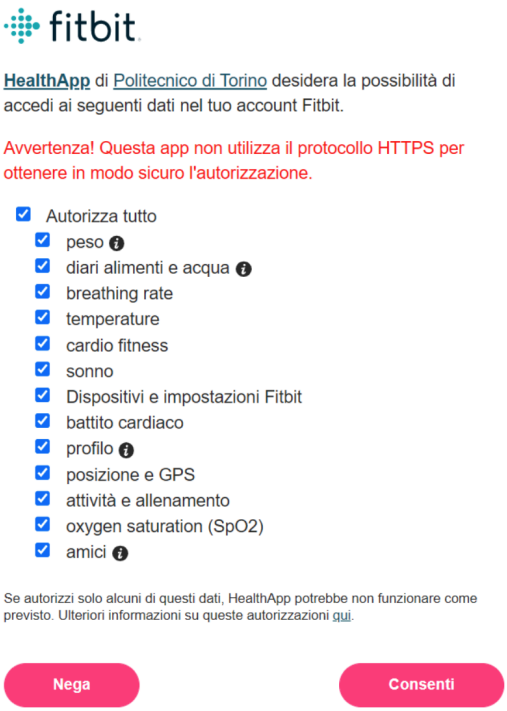
Fitbit's authentication process uses Client Credentials Grant Authentication, which involves using the client ID and client Secret obtained during registration. The diagram below summarizes the exchanges between the HealthApp server and the Fitbit server that lead to generating an access token for each user:



This process ensures that HealthApp can securely and efficiently access Fitbit-generated data while maintaining user privacy and data integrity.

Detailed Process:

1. The patient, newly registered on HealthApp, needs to log in for the first time.
2. During the first login, the HealthApp web client calls the OAuth 2.0 Authorization URI, passing the client ID in plaintext and the client Secret encrypted via a cryptographic challenge.
3. The Fitbit Authorization Server recognizes the client originating the request (in this case, HealthApp) and offers the patient the option to log in with their Fitbit account.
4. Once logged in, the user is prompted to grant the necessary permissions to access their data generated by the wearable, as shown in the following figure.



5. After obtaining consent, the Fitbit Authorization Server generates a one-time code and calls the API of HealthApp contained in the Redirect URL, passing this code as a parameter.
6. The received code is used by the HealthApp backend to call the OAuth 2.0: Access/Refresh Token Request URI and request the token generation.
7. The Fitbit Authorization Server generates an access token and a refresh token. The access token is valid for eight hours, allowing access to the patient's data. After eight hours, it needs to be renewed using the refresh token, which can be used once to obtain a new access token. The access and refresh tokens are sent as a response to the API call related to the OAuth 2.0: Access/Refresh Token Request URI.
8. The HealthApp server receives and saves the access and refresh tokens, associating them with the patient's account, thereby gaining access to the data.

This procedure is carried out only during the patient's first login to HealthApp. For all subsequent logins, the server checks if the token is still valid. If the token has expired, another procedure is initiated:

1. The HealthApp server retrieves the saved refresh token and sends it to the Fitbit Authorization Server via the OAuth 2.0: Access/Refresh Token Request URI.
2. The Fitbit Authorization Server generates a new pair of access and refresh tokens and sends them back to the HealthApp server.
3. The HealthApp server saves the new access and refresh tokens.

The refresh procedure is much quicker and more efficient than obtaining the initial token. It is repeated not only during patient logins if the token has expired but also through a daily routine that verifies and

updates expired tokens for all patients. If there are issues refreshing the token during login, the process of obtaining a new access token from scratch will be repeated, similar to the first login procedure.

Since, under the current architecture, a patient account can only be created by a doctor or administrator account (self-sign-up is not available), the entire initial account setup procedure will likely be performed in the presence of the doctor, who should be instructed on the necessary steps.

7. Security

In our project, Spring Security plays a crucial role in safeguarding our application by providing a robust security configuration [20]. The ApplicationSecurityConfig class is central to this setup, utilizing annotations such as @EnableWebSecurity and @EnableMethodSecurity(prePostEnabled = true) to enable web security and method-level security. The configuration defines a comprehensive SecurityFilterChain bean, which handles various aspects of security including CORS, CSRF protection, and authentication. It meticulously specifies access control for different endpoints using authorizeHttpRequests, ensuring that sensitive operations are only accessible to users with appropriate permissions or roles, like DOCTOR_READ or PATIENT_WRITE. Authentication is managed through a custom DaoAuthenticationProvider that integrates a UserDetailsService and a PasswordEncoder for encoding and matching passwords. Additional security measures include customized login and logout handlers, CSRF disabling for specific endpoints, and detailed role-based access controls for endpoints related to patients, doctors, and various health-related data. Moreover, method-level security is enforced in controllers using @PreAuthorize, ensuring that only authenticated and authorized users can access or manipulate resources. For instance, in the PatientHrController, access to heart rate data is restricted to the patient or authorized roles through the @PreAuthorize("@authenticatedUserService.hasIdAndIsPatient(#patientId)") annotation, thus ensuring compliance with security policies and protecting sensitive health information.

The first step involves validating the inputs sent via the existing APIs from web and mobile clients. This validation goes beyond the automated checks performed by the framework on the type and presence of values. The most significant validation mechanisms include:

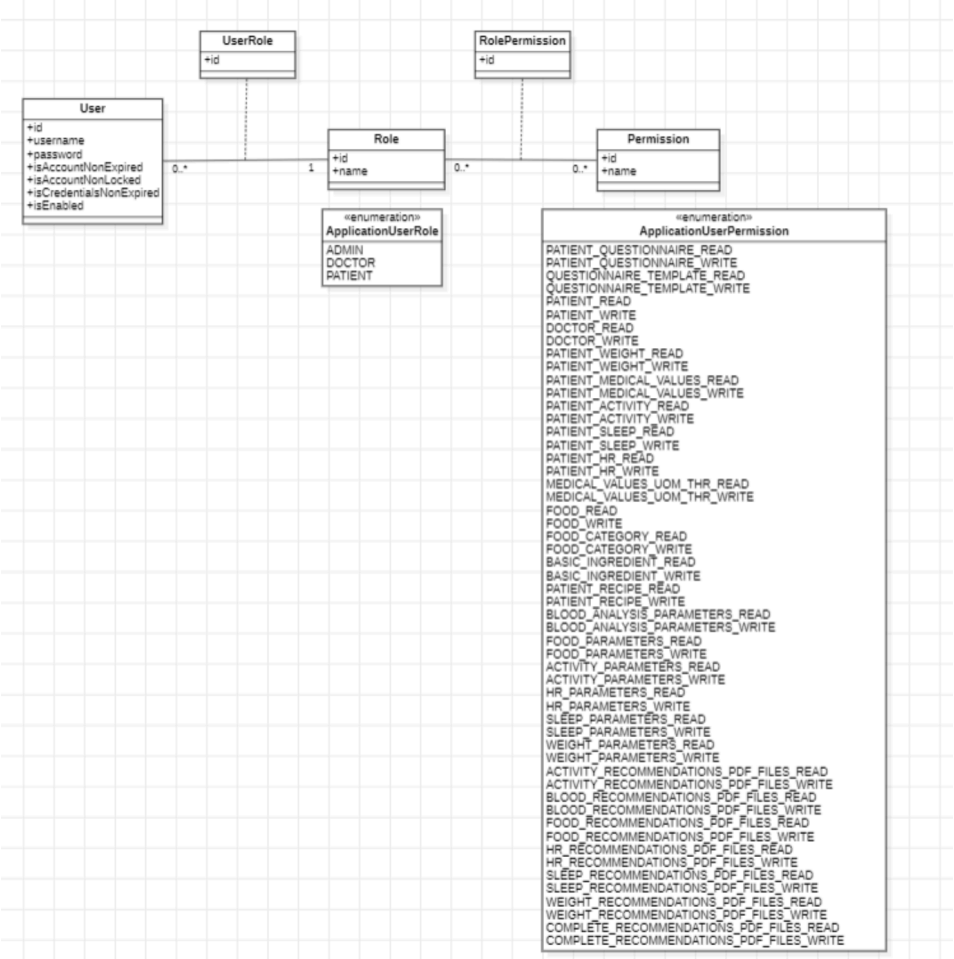
- **Date of Birth during Patient Registration:** It is crucial that the patient is an adult at the time of account creation.
- **Email:** All input emails are validated using regular expressions to ensure syntactic correctness.
- **Date Range:** It is verified that the start date is less than or equal to the end date.
- **Strings:** They are checked to ensure a minimum length of one character and a maximum length that varies depending on the nature of the value.

Regarding server security, a system is set up to filter requests based on the permissions possessed by the clients. Besides requiring that every request (except for login and password reset via email) be authenticated, it is crucial to establish which type of user should have access to which type of resources. Therefore, it is necessary to implement an access control system. To this end, three pre-authorization methods are defined:

- **hasIdAndIsPatient(ID: Long): Boolean** – This method checks if the authenticated user making the request is a patient and if their ID matches the ID of the resource they are requesting (whether for reading or writing). Thus, a patient with ID 1 can only access the data of patient ID 1 and not that of other patients. Depending on the outcome, the method returns a boolean. Additionally, if the ID verification fails but the authenticated user has doctor or administrator privileges, the method still returns true, allowing access to users with higher privileges than a patient.
- **isAdminOrSameUsername(userName: String): Boolean** – This method is used for requests to access user data. It allows a user with a specific username to access only their own data while

allowing an administrator to access data for all users. It returns a boolean based on whether the privilege check is successful.

- **isAdminOrDoctor(): Boolean** – This method checks if the authenticated user requesting access has administrator or doctor privileges and returns a boolean based on the verification result. It is mainly used in the context of recommendations to allow only doctors (and administrators) to modify customizable analysis parameters for each patient.



The roles and permissions system within HealthApp ensures that users have appropriate access levels tailored to their needs and responsibilities. Each user, identified by a username and password, can assume only one role, which corresponds to specific permissions. A permission represents the ability to access a particular entity in the database, either for reading or writing. The defined roles include:

- **PATIENT:** Represents the patient, the primary user of HealthApp. Permissions include read and write access to their own data related to nutrition, activity, health, and sleep, and read-only access to recommendation reports.
- **DOCTOR:** Represents the doctor, who supervises patient activities. Permissions include everything the patient has, plus the ability to view any patient's data, define thresholds for recommendations, and create new patient accounts.

- **ADMIN:** Represents the administrator, an exceptional user of HealthApp. Permissions include read and write access to every data structure in the application, allowing them to manipulate any information and user.

These roles and permissions ensure a well-structured and secure access control system, enhancing the overall security and integrity of the application.

8. Containerization, Deployment and Execution

8.1 Containerization

The first step involves creating a core image containing all the necessary elements to allow the system to run autonomously. For this purpose, Docker is used. Docker is an open-source software designed to create and run images in an isolated environment called a container [21]. Multiple containers can communicate through a network bridge, which connects the individual container networks. This bridge will be used for communication between the core container and the web frontend container.

A container allows an image to run internally, consisting of all the resources required (libraries, executable code, volumes, etc.) for a process to run autonomously. An image is built using a set of directives collected in a file known as a Dockerfile. Each image must have its own container; to run one or more, you can use a command-line interface, a graphical interface, or the Docker Compose tool, which can run multi-container applications based on a configuration file with a YAML extension.

For the HealthApp core, it is necessary to define the Dockerfile that will create the executable image. The file is structured as follows:

1. FROM adoptopenjdk/openjdk11:alpine-jre
2. VOLUME /data
3. ADD /data/db_data.mv.db /data/db_data.mv.db
4. ARG JAR_FILE=target/*.jar
5. COPY ${JAR_FILE} app.jar
6. EXPOSE 8080
7. ENTRYPOINT ["java", "-jar", "/app.jar"]

- Directive 1 specifies the base image used, containing all necessary libraries and ancillary tools.
- Directives 2 and 3 define the volume that will serve as the application's database and its path. In this case, the volume will be in the data folder, while the database will be a simple file copied from the local system (/data/db_data.mv.db) to the Docker image (same path), containing tables and data.
- Directive 4 designates the JAR file in the target folder as a variable named JAR_FILE.
- Directive 5 copies this file to the root path of the Docker image, naming it app.jar.
- Directive 6 exposes port 8080 on the container's network, making it accessible.
- Directive 7 specifies the command to execute to start running the application.

After defining the Dockerfile, the next step is to produce the JAR (Java Archive) file, which is an archive containing all the core source code used at runtime. To generate this file, run the maven package command in the project's root folder, allowing the tool to package the various classes.

Once the Dockerfile and JAR are defined, the final step is to create the image to run inside the Docker container. This can be done using the graphical interface provided by the IDE (IntelliJ in this case) or the docker build command. The creation process takes a few seconds and generates the image and the associated volume, ready to be executed within a container.

Finally, upload the image to the project's Docker repository and proceed to the second phase, which is execution.

8.2 Deployment

To make the backend accessible to web and mobile clients, it is necessary to transition from the development phase to the deployment phase. The code produced and tested so far must be "packaged, uploaded, and executed" on a machine that exposes a public IP address, making it reachable from the Internet. The deployment procedure can be illustrated in two phases: containerizing and execution.

The backend system will be collectively referred to as the "core". A similar deployment procedure will be followed by colleagues for the web and mobile frontends, making the web application and mobile application available to end users. This chapter will exclusively cover the deployment of the core.

8.3 Execution

To run the image, a machine (host) connected to the network and reachable by other machines via a public IP address is required. For HealthApp, there are two viable solutions:

- An internal machine within the Politecnico network, accessible through a path relative to the domain softeng.polito.it
- A machine offered for free for 12 months by Amazon Web Services (AWS)

Initially, the first executions were conducted on Softeng and then moved to AWS. This choice is motivated by having a host directly controllable by the developers without the need for intermediaries. Since Politecnico offers this possibility only after a lengthy request procedure, an intermediary is used only to verify the image's functionality on Softeng, while AWS is used for the actual execution. This ensures any issues can be corrected as quickly as possible, providing maximum server availability and guaranteeing proper communication between the client and core.

Regardless of the chosen solution, the steps to execute are the same:

1. **Install Docker**: Install the Docker application on the chosen machine.
2. **Download the Image**: Download the previously uploaded image from the repository.
3. **Run the Docker Container**: Execute the docker run command to run the image within the container, mapping port 8080 (previously exposed in the Dockerfile) to the chosen port. Additionally, add the --restart always flag to allow the container to restart automatically in case of crashes, exceptions, or unexpected interruptions.
4. **Access the Core**: Access the core via the path http(s)://BASE_URL:PORT, where BASE_URL represents the host's path or IP address, and PORT is the port chosen during mapping.

Deployment is now complete. The web and mobile clients can point to the core path to have the entire system operational.

9. Suggestion and Improvements

9.1. Use PostgreSQL Database Instead of H2 In-Memory Database

One significant improvement for the project would be transitioning from the H2 in-memory database to PostgreSQL. While H2 is convenient for development and testing due to its simplicity and speed, it is not suitable for production environments due to its limitations in terms of data persistence, scalability, and reliability. PostgreSQL, on the other hand, is a powerful, open-source object-relational database system with a strong reputation for reliability, feature robustness, and performance. It supports complex queries, ACID compliance, and provides advanced features such as full-text search, JSON support, and extensive indexing options. Migrating to PostgreSQL will ensure better data integrity, performance, and scalability, making the application more robust and production-ready.

9.2. Use Kubernetes for Proper Deployments and Have a Dedicated Web Server

To enhance deployment efficiency and scalability, leveraging Kubernetes for orchestration is highly recommended. Kubernetes automates the deployment, scaling, and management of containerized applications, ensuring high availability and scalability. It provides features like automatic bin packing, self-healing, horizontal scaling, and service discovery. Utilizing Kubernetes will streamline the deployment process, making it easier to manage and scale the application as demand grows. Additionally, having a dedicated web server for the application or utilizing cloud services like AWS, GCP, or Azure can further improve the application's performance, security, and scalability. These cloud platforms offer managed services, global distribution, and robust security features, allowing for efficient resource management and operational excellence.

9.3. Provide Monitoring with Grafana or Datadog

Implementing a robust monitoring system is crucial for maintaining the health and performance of the application. Tools like Grafana or Datadog can be used to monitor various metrics, providing real-time insights into the system's performance. Grafana, in combination with Prometheus, can offer comprehensive monitoring capabilities with powerful visualization options. Datadog, on the other hand, provides a comprehensive cloud-based monitoring solution with extensive integration options. Utilizing the Spring Boot Actuator endpoints, these tools can be configured to monitor application metrics, health, and performance, allowing for proactive issue detection and resolution. Although the migration to Kubernetes has been completed, incorporating monitoring tools remains an essential next step for ensuring the application's reliability and performance.

9.4. Support for Different Devices or Watches

Currently, the application primarily supports Fitbit devices. Expanding support to include other wearable devices, such as Apple Watch, Garmin, or Samsung Galaxy Watch, would make the application more

versatile and accessible to a broader user base. Each device offers unique features and data points, which could provide more comprehensive health insights for users. This expansion would involve integrating with different APIs and ensuring compatibility with various data formats. Exploring the support for different devices would be a valuable area of study and could significantly enhance the application's functionality and appeal.

9.5. PostgreSQL Migration and Kubernetes Deployment

The migration from H2 to PostgreSQL has been successfully implemented, significantly improving the application's data management capabilities. PostgreSQL's robustness and advanced features have enhanced the application's reliability and scalability. Additionally, the deployment has been transitioned to Kubernetes, which has streamlined the deployment process and improved the application's scalability and availability. These enhancements have laid a strong foundation for the application, providing a scalable and reliable infrastructure that can support future growth and improvements.

In conclusion, these suggestions aim to further enhance the application's performance, scalability, and usability. Transitioning to PostgreSQL, utilizing Kubernetes, implementing robust monitoring solutions, and expanding device compatibility are key steps that will ensure the application's continued success and growth.

10. Bibliography

[1] World Health Organization, "Global Health Estimates," 2019. [Online]. Available: https://www.who.int/data/global-health-estimates.

[2] National Institutes of Health, "Healthy Living," 2020. [Online]. Available: https://www.nih.gov/news-events/nih-research-matters/healthy-living.

[3] S. Pilotto, E. Panza, and S. Ferrucci, "Multidimensional Prognostic Index (MPI)," Journal of Gerontology, vol. 67, no. 5, pp. 527-534, 2012.

[4] A. C. De Saint-Hubert, P. Pepersack, and Y. Michel, "The Multidimensional Prognostic Index (MPI) accurately predicts mortality in older patients: A systematic review and meta-analysis," Age and Ageing, vol. 42, no. 6, pp. 699-708, 2013.

[5] Bluetooth SIG, "Bluetooth Low Energy," 2019. [Online]. Available: https://www.bluetooth.com/learn-about-bluetooth/bluetooth-technology/bluetooth-low-energy.

[6] Xiaomi Corporation, "Mi Band 3 User Manual," 2018. [Online]. Available: https://www.mi.com/global/miband3.

[7] Fitbit, Inc., "Fitbit API," 2020. [Online]. Available: https://dev.fitbit.com/build/reference/web-api.

[8] Pivotal Software, Inc., "Spring Boot Reference Guide," 2020. [Online]. Available: https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle.

[9] Liquibase, "Liquibase Documentation," 2020. [Online]. Available: https://docs.liquibase.com.

[10] M. Fowler, "Patterns of Enterprise Application Architecture," Addison-Wesley Professional, 2002.

[11] A. Cockburn, "Hexagonal Architecture," 2005. [Online]. Available: http://alistair.cockburn.us/Hexagonal+architecture.

[12] Bluetooth SIG, "GATT Specifications," 2019. [Online]. Available: https://www.bluetooth.com/specifications/gatt.

[14] JetBrains, "Kotlin Documentation," 2020. [Online]. Available: https://kotlinlang.org/docs/reference.

[15] Meta, "React - A JavaScript Library for Building User Interfaces," 2020. [Online]. Available: https://reactjs.org/docs/getting-started.html.

[16] Meta, "React Native - A Framework for Building Native Apps using React," 2020. [Online]. Available: https://reactnative.dev/docs/getting-started.

[17] H2 Database Engine, "H2 Database Documentation," 2020. [Online]. Available: http://www.h2database.com/html/main.html.

[18] PostgreSQL Global Development Group, "PostgreSQL Documentation," 2020. [Online]. Available: https://www.postgresql.org/docs.

[19] R. Fielding, "Architectural Styles and the Design of Network-based Software Architectures," Doctoral Dissertation, University of California, Irvine, 2000.

[20] Pivotal Software, Inc., "Spring Security Reference Guide," 2020. [Online]. Available: https://docs.spring.io/spring-security/site/docs/current/reference/html5.

[21] Docker Inc., "Docker Documentation," 2020. [Online]. Available: https://docs.docker.com.

[22] The Kubernetes Authors, "Kubernetes Documentation," 2020. [Online]. Available: https://kubernetes.io/docs/home.