DESARROLLO E IMPLEMENTACIÓN DE UNA HERRAMIENTA DE VISUALIZACIÓN Y SEGUIMIENTO DE PACIENTES EN ESTUDIO CLÍNICO

DEVELOPMENT AND IMPLEMENTATION OF A VISUALIZATION TOOL AND MONITORING OF PATIENTS IN CLINICAL STUDY

BORJA COLMENAREJO GARCÍA

MÁSTER EN INGENIERÍA INFORMÁTICA. FACULTAD DE INFORMÁTICA
UNIVERSIDAD COMPLUTENSE DE MADRID

Trabajo Fin Máster en Ingeniería Informática

17 de Junio de 2019
Nota: 5.0

Director y colaborador:

José Luis Risco Martín
Josué Pagán Ortiz

Autorización de difusión

Borja Colmenarejo García

17 de Junio de 2019

El abajo firmante, matriculado en el Máster en Investigación en Informática de la Facultad de Informática, autoriza a la Universidad Complutense de Madrid (UCM) a difundir y utilizar con fines académicos, no comerciales y mencionando expresamente a su autor el presente Trabajo Fin de Máster: "DESARROLLO E IMPLEMENTACIÓN DE UNA HERRAMIENTA DE VISUALIZACIÓN Y SEGUIMIENTO DE PACIENTES EN ESTUDIO CLÍNICO", realizado durante el curso académico 2018-2019 bajo la dirección de José Luis Risco Martín y con la colaboración externa de dirección de Josué Pagán Ortiz en el Departamento de Arquitectura de Computadores y Automática, y a la Biblioteca de la UCM a depositarlo en el Archivo Institucional E-Prints Complutense con el objeto de incrementar la difusión, uso e impacto del trabajo en Internet y garantizar su preservación y acceso a largo plazo.

Resumen en castellano

Los estudios clínicos mejoran la calidad de vida de los pacientes enfermos en la búsqueda de tratamientos clínicos para paliar las enfermedades. En la actualidad se realizan multitud de ensayos clínicos. La realización de éstos supone un gran coste y es necesario de buenas herramientas para llevar a cabo un seguimiento minucioso de los pacientes. Con la evolución de la tecnología lo sensato sería pensar que los estudios médicos han avanzado con ella. Pero a menudo nos encontramos con que muchos estudios clínicos no se han adaptado a las nuevas tecnologías o se han quedado a medio camino. Para ayudar con el seguimiento de los ensayos clínicos, en el presente trabajo se propone una arquitectura escalable, mantenible y de fácil adaptación para desarrollar e implementar aplicaciones móviles sin mucho esfuerzo.

Para demostrar la utilidad de la arquitectura propuesta se muestra un caso de uso centrado en el estudio clínico llevado a cabo por el proyecto BrainGuard. En dicho estudio el personal médico realiza el seguimiento de los pacientes a través del personal técnico, los cuales mantienen una base de datos con la información del estudio.

Mediante la herramienta desarrollada se trata de mejorar el estudio proporcionando los instrumentos necesarios para que el proceso sea más dinámico y fluido. Además, la herramienta conecta los resultados de los pacientes con los responsables implicados en el proyecto en tiempo real, permitiendo realizar análisis más rápidos. Esto dota a los estudios clínicos de la capacidad de adaptarse a las necesidades de los pacientes antes de que sea demasiado tarde y se tenga que descartar a los pacientes.

El presente trabajo pretende ser una referencia de arquitectura para el desarrollo de futuras herramientas enfocadas a otros estudios clínicos de otras enfermedades, pues la herramienta desarrollada es de código libre y se aporta toda la información para adaptarla sin que suponga un gran esfuerzo para los desarrolladores.

Palabras clave

Estudio clínico, herramienta de seguimiento, aplicación móvil, Kotlin, migraña, BrainGuard, aplicación multiplataforma, datos clínicos.

Abstract

Clinical studies improve the quality of life of sick patients, improving and discovering clinical treatments to alleviate diseases. At present, many clinical trials are carried out. The realization of these involves a great cost and it is necessary to have good tools to carry out an exhaustive monitoring of patients. With the evolution of technology, it would be prudent to think that medical studies have advanced with it. But we often find that many clinical studies have not been adapted to new technologies or have been halved. To help with the monitoring of clinical trials, this document proposes a scalable, easy to maintain and easily adaptable architecture to develop and implement mobile applications without much effort.

To demonstrate the utility of the architecture, a case of use centered on the clinical study carried out by the BrainGuard group is shown. In this study, the medical personnel follow up the patients through the technical personnel. Those who maintain a database with the information of the study.

Through the tool developed, the study is improved by providing the necessary instruments to make the study more dynamic and fluid. In addition, the tool connects the results of patients with those responsible involved in the project in real time, which allows a faster analysis. This gives clinical trials the ability to adapt to the needs of patients before it is too late and patients should be discarded.

The present work intends to be a reference of architecture for the development of future tools focused on other clinical studies of other diseases, because the developed tool is a free code and all the information is provided to adapt it without supposing a great effort for the developers.

Keywords

Clinical study, monitoring tool, mobile application, Kotlin, migraine, BrainGuard, multiplatform application, clinical data.

Índice general

Agradecimientos

Agradecer a mis tutores, José Luis Risco Martín y Josué Pagán Ortiz, que me han guiado durante todo el trabajo. Por todo su apoyo y sus sabios consejos.

Gracias.

Si la única oración que dijiste en toda tu vida fue 'gracias', eso sería suficiente.

Meister Eckhart

Dedicatoria

A mis compañeros de trabajo, mis amigos y mi familia.

Capítulo 1

Introducción

1.1. Motivación

Los ensayos o estudios clínicos son investigaciones que requieren de la participación de personas[1]. Gracias a los estudios clínicos los doctores en medicina y otros ámbitos relacionados con la salud encuentran nuevas formas de mejorar la calidad de vida de las personas mediante nuevos tratamientos.

Los estudios clínicos son la última fase de un largo procedimiento que comienza con la investigación en laboratorios. Antes de incluir a personas en un estudio clínico es necesaria la realización de dichas investigaciones para entender los efectos que puede suponer sobre las personas. Y es durante los estudios clínicos cuando se descubre el verdadero impacto que conlleva, así como los efectos secundarios que los pacientes pueden tener.

Para llevar a cabo los estudios clínicos se requiere tanto de personas especializadas como de herramientas que permitan realizar un seguimiento minucioso. Los estudios clínicos generan una gran cantidad de datos que necesitan ser procesados y analizados para poder obtener conclusiones. Los datos clínicos suponen de un problema en sí mismo por varios factores. Primero, no todas las personas tienen los conocimientos necesarios para la interpretación de los datos. Esto requiere de una persona con conocimiento suficiente para la

realización del análisis. Cada enfermedad es medida de forma diferente por cada centro que realiza sus propios ensayos clínicos. Al no existir un estándar para recoger y almacenar la información no es posible compartir datos entre los diferentes centros y en muchas ocasiones se tiende a repetir fases de los estudios clínicos que suponen un esfuerzo, tanto de tiempo como económico, para los promotores del estudio.

Con el avance de la tecnología, la intuición dice que los estudios médicos avanzan rápidamente. A menudo nos sorprenderíamos con la cantidad de información que se sigue recogiendo a mano y se almacena para posteriores estudios. Esta información es almacenada en grandes silos, donde se hace costosa la búsqueda para su reutilización. En muchas ocasiones ni siquiera se es consciente de la información que se está almacenando.

Por la naturaleza de los datos clínicos, datos sensibles, la realización de aplicaciones que faciliten los ensayos clínicos supone un problema para los promotores implicados.

Gracias al libre mercado internacional existe gran accesibilidad a los dispositivos móviles donde los precios varían dependiendo del modelo y la tarifa elegida. De esta forma, con apenas 10 euros al mes se puede conseguir un dispositivo con acceso a internet. Además, estos dispositivos son muy cómodos ya que se pueden llevar en cualquier sitio. Esto facilita una navegación veloz mediante aplicaciones que responden rápidamente, en ocasiones más rápido incluso que un ordenador[2].

Al existir multitud de dispositivos aparece la segregación del sistema operativo[3]. Muchos de los móviles antiguos no soportan determinadas versiones de los sistemas operativos, impidiendo instalar aplicaciones relativamente nuevas. Esto supone una barrera para desarrollar aplicaciones médicas ya que son muy costosas y en la mayor parte de los estudios clínicos los dispositivos empleados tienden a ser muy antiguos.

En este trabajo se diseña e implementa una arquitectura software que define una herramienta de gestión de datos clínicos fácilmente adaptable a cualquier dispositivo, debido a la

naturaleza del modelo propuesto.

1.2. Objetivos

El objetivo principal del presente trabajo consiste en desarrollar e implementar una herramienta de visualización y seguimiento de pacientes para estudios clínicos. Mediante las nuevas tecnologías se desarrollará una herramienta de fácil escalado y de mantenimiento sencillo que permitirá realizar un seguimiento con un solo vistazo.

Además la herramienta permitirá visualizar los resultados en tiempo real. Dotando a los doctores implicados en los estudios de la capacidad de análisis en tiempo real, los estudios se pueden volver más dinámicos y podrán evolucionar con mayor fluidez.

También se eliminarán completamente las hojas de papel de los estudios clínicos. Por un lado, esto supone la supresión de almacenes para guardar la información. Y por otro lado, implica la informatización de los datos. Digitalizar los datos a su vez permite establecer un estándar de los datos para cada uno de los estudios. Es decir, permite reutilizar y compartir información entre los diferentes estudios.

Por último, se proporcionará una herramienta de seguimiento genérica y de código libre que permitirá el escalado para su adaptación a otras enfermedades. Actualmente el trabajo se realizará enfocado a pacientes con migraña, pero existen enfermedades tales como ictus, afasia y esclerosis que pueden ser adaptadas fácilmente y que permitirían llevar el seguimiento a dispositivos portátiles como teléfonos inteligentes o tabletas.

1.3. Estado del arte

En la actualidad existen multitud de herramientas que permiten realizar el seguimiento médico de las enfermedades más comunes. Todas estas herramientas están orientadas a que

los pacientes lleven su propio seguimiento médico y que una vez en la consulta médica puedan hacer uso de ellas para mostrárselo a los doctores. Se podría decir que son aplicaciones para almacenar el historial de las enfermedades.

Cada una de estas aplicaciones se enfoca en una enfermedad en concreto. Por ejemplo, para el seguimiento de la migraña[4] nos encontramos con aplicaciones como *Migraine Buddy*[5], *iMigraine*[6] o *iHeadache*[7]. Algunas de ellas tratan de predecir el dolor, pero no están en desarrollo continuo ni se involucran en estudios clínicos.

En el mercado existen también otras aplicaciones como *NeuroScores*[8] que incluyen una serie de escalas para intentar diagnosticar enfermedades antes de que ocurran. Estas aplicaciones también están orientadas a las personas que sufren enfermedades y, que en este caso, además identifican los síntomas.

No podemos localizar ninguna aplicación que conecte a los pacientes con el personal clínico sin que requiera del desplazamiento de uno de los miembros para que ambas personas puedan reunirse. Tampoco podemos encontrar una herramienta que muestre en tiempo real los resultados del análisis de los pacientes que colaboran en estudios clínicos.

Por otro lado, dentro del ámbito tecnológico no existe una herramienta base, de fácil adaptación, para la realización de estudios clínicos que no requiera de un gran esfuerzo para su puesta en marcha.

1.4. Estructura de la memoria

El presente trabajo se ha estructurado de la siguiente manera:

En el Capítulo 1 se presenta el trabajo que se ha realizado durante el proyecto, escrito en Español. En el Capítulo 2, encontramos la misma información que en el Capítulo 1, pero escrito en Inglés. Estos capítulos contienen los objetivos, motivación y estado del arte del

proyecto. Además de la propia explicación de la estructura del documento.

El Capítulo 3 contiene una explicación de la arquitectura del sistema que se ha implementado durante el proyecto. Para ello primero se introduce la arquitectura elegida. Y tras ello, se explican cada uno de los componentes presentes en el proyecto. En cada componente se explican la organización interna que poseen. Además de las tecnologías y patrones de diseño de desarrollo que se han empleado.

El Capítulo 4 explica los detalles de la implementación que se han seguido para el proyecto. La explicación se realiza por cada pieza de la herramienta. Partiendo desde la base de datos, pasando por el servidor, hasta llegar a las aplicaciones cliente.

En el Capítulo 5 se realiza una exposición de un caso de uso. En este se ha llevado a cabo la puesta en escena de la arquitectura. Para ello, durante las primeras secciones se muestra una imagen general del proyecto al que pertenece la herramienta desarrollada. Para en la última sección explicar las mejoras que aporta la herramienta al estudio y la navegación que tienen las aplicaciones desarrolladas.

Los Capítulos 6 y 7 contienen las conclusiones y trabajo futuro que presenta el proyecto. En el Capítulo 6 nos encontramos con ello en Español, mientras que en el Capítulo 7 nos lo encontramos en Inglés.

Capítulo 2

Introduction

2.1. Motivation

Trials or clinical studies are investigations that require the participation of people[1]. Thanks to clinical studies, doctors in medicine and other health-related areas find new ways to improve the quality of life of people through new treatments.

Clinical studies are the last phase of a long procedure that begins with laboratory research. Before including people in a clinical study, it is necessary to carry out such research to understand the effects it may have on people. And it is during clinical studies that you discover what the real impact is, as well as the side effects that patients may have.

To carry out clinical studies requires specialized people and tools that allow a thorough monitoring. Clinical studies generate a large amount of data that must be processed and analyzed to draw conclusions. Clinical data is a problem in itself due to several factors. First, not all people have the necessary knowledge to interpret the data. This requires a person with sufficient knowledge to perform the analysis. Each disease is measured differently by each center that conducts its own clinical trials. In the absence of a standard for collecting and storing information, it is not possible to share data between the different centers and in many cases there is a tendency to repeat the phases of the clinical studies that imply an

effort, both temporary and economic, for the promoters of the study.

With the advance of technology, intuition says that medical studies are advancing speedly. We would be surprised at the amount of information that is still collected by hand and stored for further study. This information is stored in large silos, where it is expensive to search for information for reuse. In many cases, you are not even aware of the information that is stored.

Due to the nature of the clinical data, the confidential data, the realization of applications that facilitate clinical trials is a problem for the promoters involved.

Thanks to the free international market, there is great accessibility to mobile devices where prices vary according to the model and the chosen rate. In this way, with 10 euros per month you can get a device with internet access. In addition, these devices are super comfortable as they can be taken anywhere. This facilitates fast navigation through applications that respond quickly, sometimes faster even than a computer[2].

When there are many devices, the segregation of the operating system appears[3]. Many of the older mobile devices are not compatible with certain versions of the operating system, which prevents relatively new applications from being installed. This is a barrier to the development of medical applications because they are very expensive and in most clinical studies, the devices used tend to be very old.

2.2. Objectives

The main objective of this paper is to develop and implement a tool for the visualization and monitoring of patients for clinical studies. Through the new technologies, an easily scalable and easy-to-maintain tool will be developed that will allow tracking at a glance.

In addition, the tool will allow visualizing the results in real time. By providing the

doctors involved in studies of real-time analytical skills, studies can become more dynamic and evolve fluently.

Also, paper sheets from clinical studies will be completely eliminated. On the one hand, this supposes the suppression of stores to store the information. And on the other hand, it implies the computerization of the data. The digitization of the data in turn allows establishing a data standard for each of the studies. That is, it allows to reuse and share information between the different studies.

Finally, a generic monitoring tool that will allow scaling to adapt to other diseases will be provided. Currently, the work will focus on patients with migraine, but there are diseases such as strokes, aphasia and sclerosis that could be easily adapted and that would allow tracking portable devices such as smartphones or tablets.

2.3. State of the art

Currently there are many tools that allow medical monitoring of the most common diseases. All these tools are oriented so that the patients carry out their own medical follow-up and that once in the medical consultation they can use them to show them to the doctors. You could say that they are applications to store the history of diseases.

Each of these applications focuses on a specific disease. For example, for tracking migraine[4] we find applications such as *MigraineBuddy*[5], *iMigraine*[6] or *iHeadache*[7]. Some of them try to predict pain, but they are not in continuous development and do not participate in clinical studies.

In the market there are also other applications such as *NeuroScores*[8] that include a series of scales to try to diagnose diseases before they pass. These applications are also aimed at people suffering from diseases.

In addition, we can not locate any application that connects patients with clinical staff without requiring the displacement of one of the members so that both people can meet. Nor can we find a tool that shows in real time the results of patients who collaborate in clinical studies.

On the other hand, within the technological field there is no basic tool, easy to adapt, for clinical studies that do not require a great effort to start.

2.4. Memory Structure

This document has been structured as follows:

In Chapter 1 the work that has been done during the project is presented, written in Spanish. In Chapter 2, we find the same information as in Chapter 1, but written in English. These chapters contain the objectives, motivation and state of the art of the project. In addition to the explanation of the structure of the document itself.

Chapter 3 contains an explanation of the system architecture that has been implemented during the project. First, the chosen architecture is introduced. And after that, each of the components present in the project is explained. In each component they explain the internal organization they have. In addition to the technologies and design development patterns that have been employed, where appropriate.

Chapter 4 explains the details of the implementation that have been followed for the project. The explanation is made for each piece of the tool. From the database, through the server, to the client applications.

In Chapter 5 an explanation of the use case in which it has been carried out for the staging of the architecture is given. For this, during the Introduction and BrainGuard sections, a general image of the project to which the developed tool belongs is shown. In the last

section, explain the improvements that the tool brings to the study and the navigation that the developed applications have.

Chapters 6 and 7 contain the conclusions and future work presented by the project. In Chapter 6 we find it in Spanish, while in Chapter 7 we find it in English.

Capítulo 3

Arquitectura del sistema

3.1. Introducción a la arquitectura cliente-servidor

El sistema planteado para el proyecto se basa en la arquitectura clásica cliente-servidor[9]. Este modelo de arquitectura distribuida se basa en que las tareas son repartidos en diferentes proveedores, o servicios, para poder ser empleados desde los diferentes demandantes, o clientes. Un demandante solicita un recurso a un proveedor y tras realizar los procesos el cliente será satisfecho con la entrega del recurso. El modelo cliente-servidor permite diversificar las tareas que un cliente quiere realizar evitando así sobrecargar de trabajo a los clientes. Además de tener la funcionalidad localizada en el servidor permite centralizar y organizar mejor la información y separar las responsabilidades, así como proteger los datos sensibles y proporcionar una mejor seguridad al sistema. Esto es debido a que al no conocer los procesos en el cliente los usuarios tienen más complicado el intentar romperlos. Si los procesos estuvieran localizados en los clientes cualquier usuario con un mínimo de conocimiento de informática sería capaz de encontrarlo e intentar modificarlo.

Las principales ventajas que presenta este modelo son:

- Múltiples procesadores pueden ejecutar partes distribuidas de una misma aplicación, obteniendo así concurrencia en el acceso y uso de la aplicación por los usuarios.

- La migración de las aplicaciones es relativamente sencilla, ya que los cambios que se deben aplicar están centralizados en un único punto, pudiendo evolucionar el servidor sin necesidad de afectar a los clientes.
- La escalabilidad de la aplicación es total, permitiendo tanto la escalabilidad horizontal (se refiere a la capacidad de aumentar o disminuir los puestos de trabajo), como la escalabilidad vertical (aumentando la capacidad de procesado de cada uno de los puestos de trabajo).
- Se permite el acceso a los datos desde cualquier sitio para el cliente: mientras que un usuario tenga acceso a internet tendrá acceso a los datos residentes en el servidor, pues al centralizar todos los recursos en el servidor el cliente no necesitará hacer uso de ninguna otra aplicación para acceder a sus datos, solo necesitará acceso a internet.

La arquitectura de cliente-servidor permite la separación de funciones en 3 niveles[10] diferentes, tal y como se muestra en la Figura 3.1.

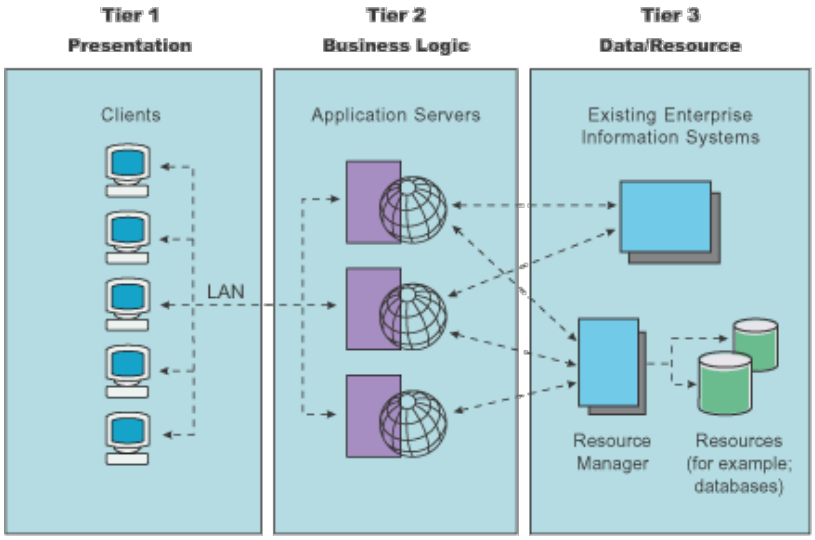

Figura 3.1: *Capas de funcionalidad de la arquitectura de cliente servidor (Fuente **IBM Knowledge Center**[11])*

La lógica de **presentación** es la encargada de las operaciones de I/O (entrada y salida) con los usuarios que hacen uso de la aplicación. Se trata de la interfaz de la aplicación

cuyas tareas son las de recoger, enviar y mostrar información. Recogiendo información de los usuarios a través de la interfaz proporcionada, la aplicación enviará está información al servidor que la procesará y devolverá una respuesta al cliente en función de si el procesado de la información fue satisfactorio o no. Esto se transformará en mostrar a los usuarios diferente información en función de la respuesta obtenida. En algunos casos la respuesta puede mostrar siempre la misma información, pues no todas las peticiones necesitan de una respuesta del servidor. Algunas peticiones pueden ser para guardado de estadísticas, o para realizar procesos que pueden tardar varios minutos, lo que se convierte en permitir a los usuarios en seguir navegando por la interfaz de la aplicación y no llegar a mostrar una respuesta. En su lugar es posible que alguno de los datos de alguna pantalla modifique sus valores y los usuarios solo sean conscientes cuando vuelvan a pasar por dicha pantalla.

La capa de **negocio**, también conocida como lógica de negocio o lógica de aplicación, es la encargada de gestionar los datos para su correcto procesamiento. Su tarea principal es la de ejecutar las reglas de negocio de la aplicación para interactuar entre los usuarios y los datos de la aplicación, es decir, entre la capa de presentación y la capa de datos. Una buena capa de negocio debe asegurarse de que las reglas son aplicadas en el orden correcto, de que no hay fallos en los procesos, y si los hubiese de informar de los fallos, de intentar recuperar el sistema de posibles errores e incluso de controlar la seguridad de la aplicación.

La capa de **datos** es la encargada de almacenar y gestionar los datos de la aplicación, así como de asegurar la consistencia e integridad de los mismos. Si ocurre un fallo en esta capa todo el sistema se verá afectado y posiblemente no será capaz de recuperarse ante una pérdida de datos o una malformación. Por ello, siempre es importante tener presente que para evitar pérdidas en el sistema habrá de existir un protocolo o plan de recuperación de datos. Como pueden ser copias de seguridad periódicas o mirroring de datos[12][1].

[1] El mirroring se trata de una técnica de almacenamiento de datos que consiste en tener almacenados los datos por duplicado en dos equipos diferentes, de tal forma que si un equipo queda dañado o inaccesible el otro equipo se ponga a funcionar y el sistema no se vea afectado por pérdida de datos

En el modelo de cliente servidor existen dos tipos de arquitecturas bien diferenciadas, la arquitectura de dos niveles y la arquitectura de tres niveles[?] . En la arquitectura de dos niveles las funcionalidades de presentación y negocio se encuentran localizadas en el cliente. Separando la lógica de datos y ubicándola en el servidor. En cambio, en la arquitectura de tres niveles todas las capas se separan en diferentes elementos. La capa de presentación se localiza en las aplicaciones cliente, mientras que las lógicas de negocio y de datos se ubican en servidores separados. Esta última arquitectura otorga al sistema de flexibilidad y escalabilidad, ya que al mantener todas las funcionalidades por separado, son fácilmente reemplazables y mejorables sin alterar el resto de elementos. En la arquitectura de tres niveles no siempre nos encontraremos con la lógica de negocio y de datos situadas en diferentes servidores. En muchas ocasiones ambas capas son ubicadas en el mismo servidor, pero esto no significa que no sea una arquitectura de tres niveles. Ambas capas pueden localizarse en el mismo servidor y seguir el paradigma de la arquitectura de tres niveles siempre y cuando la migración y actualización de las capas no suponga un esfuerzo superior al que supondría si ambas funcionalidades se encontrasen situadas en servidores diferentes.

3.2. Descripción del sistema implementado

El sistema desarrollado para la herramienta de visualización y seguimiento de pacientes en estudio clínico se basa en el modelo cliente servidor con arquitectura de tres niveles.

Tal y como muestra la Figura 3.2, las tres capas de la arquitectura están bien diferenciadas, la capa de presentación ubicada en los clientes tanto de Android como de iOS, y las capas de lógicas de negocio y de datos situadas en el servidor. Los componentes de presentación están divididos por plataforma aunque mantienen una pieza en común, ya que están construidos con un nuevo paradigma multiplataforma. Además el servicio de lógica de negocio, los servicios y la pieza común que comporten los clientes Android e iOS están escritos en el mismo lenguaje, Kotlin.

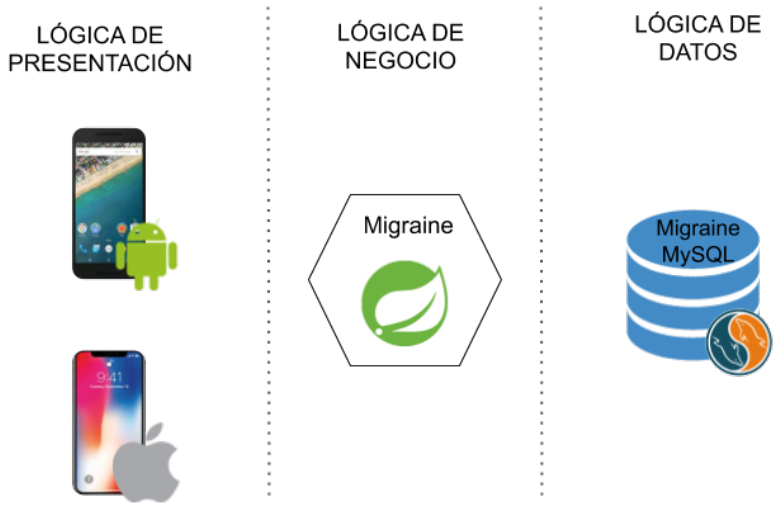

Figura 3.2: *Capas del sistema de la herramienta de visualización y seguimiento de pacientes en estudio clínico*

3.3. Servidor

El sistema desarrollado en la parte servidora o backend se compone de una única pieza que sigue la arquitectura de los microservicios[13]. Caracterizada por descomponer un servidor en pequeñas piezas que se comunican entre ellas a través, normalmente, del protocolo REST[14] y que actúa como un servidor monolítico que contiene la lógica de negocio de la aplicación.

Las principales ventajas de emplear microservicios[15], frente a una aplicación monolítica, reside principalmente en[16]:

- Aparece el desarrollo independiente, que permite evolucionar cada una de las piezas por separado, permitiendo la coordinación más eficaz de los equipos de trabajo. Por ejemplo, mientras que un equipo de trabajo está desarrollando una nueva funcionalidad en un microservicio determinado, otro equipo de trabajo puede ir solucionando pequeños bugs que aparecen en otro microservicio sin que los trabajos de ambos equipos colisionen. Todo esto es efectivo siempre y cuando los desarrollos se realicen sobre

diferentes microservicios.

- Despliegue independiente, al igual que en el anterior punto, el despliegue se puede realizar de manera autonómica. Esto permite liberar nuevas funcionalidades aisladas en un único microservicio sin afectar al comportamiento de los demás y proporcionando mayor funcionalidad al sistema en conjunto.

- Escalabilidad por componente, al tener disgregada la aplicación en diferentes piezas, aparece la posibilidad de escalar cada componente de manera individual permitiendo ajustar el sistema a las necesidades de cada momento. Por ejemplo, imaginemos que tenemos un microservicio encargado de realizar un cálculo muy pesado independiente para cada usuario y que en un momento determinado tenemos mucha concurrencia de usuarios y el dato que devuelve tarda mucho en responder. En este caso podríamos incrementar las instancias del microservicio reduciendo el tiempo de espera de devolución del dato sin necesidad de restringir la cantidad de usuarios concurrentes.

- Reutilizables, en muchos sistemas software aparecen cantidad de componentes comunes. Al dividir un gran sistema en piezas pequeñas cabe la posibilidad de reutilizar las piezas de un sistema en otro sistema, haciendo los desarrollos más cortos y permitiendo a los desarrolladores centrarse en desarrollar y evolucionar las funcionalidades más específicas.

Tal y como se muestra en la Figura 3.3, el desarrollo de la pieza back-end se compone de 5 tecnologías bien diferenciadas.

3.3.1. MySql

Es un sistema de gestión de base de datos relacional programado principalmente en ANSI C y C++[17]. Considerada la base de datos de código abierto más popular del mundo, sobre todo para entornos web[18]. Permite la gestión de multiusuario y multihilo, aportando la

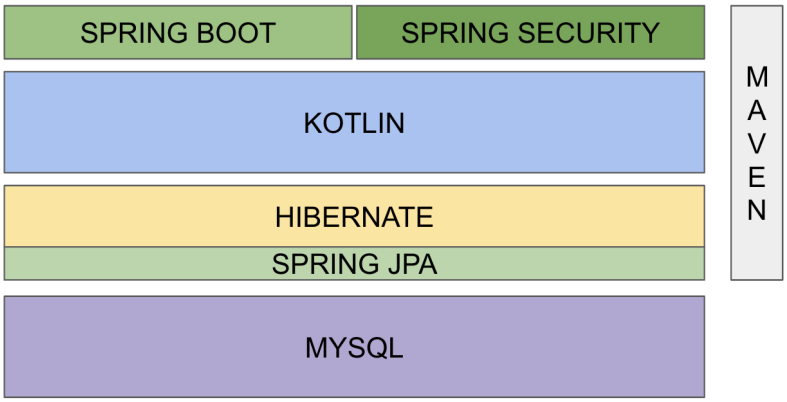

Figura 3.3: *Diagrama de tecnologías empleadas en del sistema back-end*

capacidad de realizar varias consultas a la vez por diferentes personas y a través de diferentes conexiones[19].

MySQL se emplea en el proyecto para guardar los datos de los usuarios, así como las migrañas, las sesiones y los síntomas que han tenido durante las mismas. Además se emplea para guardar información de consulta como son las medicinas empleadas, los estudios que se han ido realizando y los activos, los centros médicos implicados en los diferentes estudios, los tratamientos y la evolución de los pacientes.

La gestión se ha realizado desde la aplicación MySQLWorkbench[2] durante el desarrollo del sistema back-end en localhost, apoyado siempre de phpMyAdmin que ha permitido ir viendo los cambios que se han realizado en la máquina servidora tras el despliegue del servidor. Esto ha permitido verificar la consistencia de los cambios realizados y la correctitud de los datos almacenados.

3.3.2. Hibernate

Es un framework de mapeo de entidades objeto-relacional para la plataforma JAVA, esto también es conocido como un ORM, por sus siglas en inglés Object-Relational Mapping[20]. Es

[2] https://www.mysql.com/products/workbench/

una técnica de programación que transforma las entidades de las bases de datos relacionales en objetos para lenguajes de programación orientados a objetos[21]. Permite detallar las entidades utilizando todo el potencial de la programación orientadas a objetos sin romper los patrones de desarrollos de JAVA. Los ORMs ayudan a los desarrolladores a olvidarse de realizar las consultas más básicas, permitiendo centrarse en las realmente complejas con tan solo configurarlo correctamente. Hibernate permite esta configuración mediante anotaciones sencillas sobre el objeto que representa la base de datos[22].

Hibernate se utiliza en el servidor backend para permitir comunicar la base de datos con los microservicios, dotando al servidor de la posibilidad de operar sobre la base de datos de manera sencilla. Para ello, se emplean objetos como entidades de la base de datos, y en ocasiones con anidación de objetos mediante las relaciones existentes. Por ejemplo, en el servidor desarrollado nos encontramos con que las tablas *symptoms* y *symptomns_ name* son consultadas con anidación de clases, permitiendo realizar las consultas de los síntomas de las migrañas a través del objeto *SymptomsDto*, pero añadiendo a esta clase una subclase con un relación uno a uno con la tabla *symptomns_ name* mediante el objeto *SymptomsNameDto*. Esto permite acceder a las columnas de la tabla *symptomns_ name* a través de *symptomns* sin tener que realizar una consulta explícita para obtener el nombre de los síntomas.

Hibernate ha sido empleado mediante anotaciones en los objetos entidades de la base de datos que unidos a Spring (en concreto Spring-jpa, véase 3.3.3) han permitido una gestión sencilla de la base de datos.

3.3.3. Spring Framework

Como su propio nombre indica es un framework para el desarrollo de aplicaciones Java que puede ser empleado en cualquier tipo de aplicación[23]. Desde su aparición en 2004 se ha vuelto muy popular entre los desarrolladores Java y es considerado un complemento para los

EJB (Enterprise JavaBean[3]). El framework está compuesto de varios módulos que se han ido añadiendo en los años de desarrollo. Además Spring permite utilizar el framework completo, o solo partes de las funcionalidades que provee. El siguiente esquema (3.4) muestra todas las funcionalidades de Spring.

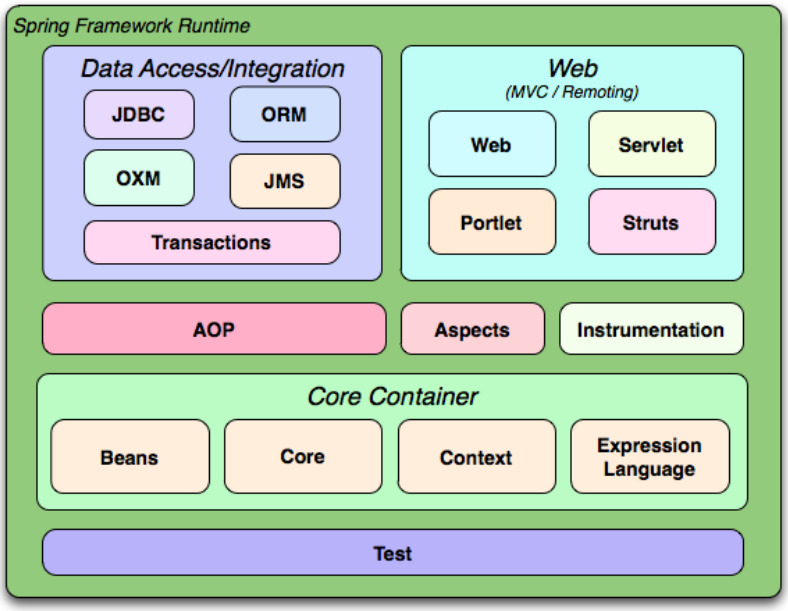

Figura 3.4: *Módulos de Spring Framework (Fuente* ***Programación J2EE. ¿Qué es Spring Framework?***[24]*)*

En el proyecto se han empleado los módulos de *core container*, *beans*, *core*, *context* y *expression language*. Estos representan los módulos mínimos para que una aplicación con Spring funcione. Por parte de Web, se han empleado los módulos de *web*. Y para acceder a la base de datos se han empleado los módulos de *ORM* y *transactions*.

Spring es separado por funcionalidad, de forma que es posible emplear partes de Spring sin incluir todas las dependencias. En nuestro caso, hemos empleados 3 funcionalidades de Spring[25].

- **Spring-boot**: Es la funcionalidad encargada de levantar una aplicación web con un

[3] EJB son una serie de interfaces de programación de aplicación que marcan un estándar para la creación de aplicaciones J2EE o JEE.

servidor embebido dentro del propio servicio. Esto permite emplear el paradigma de microservicios en todo su potencial. Ya que cada servicio es en sí mismo un servidor de aplicaciones que puede ejecutarse de manera autónoma sin necesidad de un servidor Apache o JBoss por detrás. Por defecto levanta un servidor Tomcat, pero en el proyecto ha sido sustituido con un servidor Jetty.

- **Spring-security**: Provee de seguridad al servicio web con tan solo implementar unas pocas clases de configuración. Spring-security soporta los principales protocolos de seguridad como pueden ser Basic Auth, Bearer token, OAuth 1.0 u OAuth 2.0. En nuestro caso el protocolo utilizado ha sido el de OAuth 2.0 combinado con JWT (JSON Web tokens).

- **Spring-jpa**: es la parte del framework encargada de simplificar el acceso a base de datos mediante repositorios. Un repositorio es una interfaz que extiende de otras ya existentes, y que necesita indicar mediante parámetros genéricos el objeto de la tabla al que hace referencia y el objeto de la clave primaria que lo relaciona. Estos repositorios vienen equipados con las principales tareas que se realizan sobre la tabla, como son el *select*, el *update*, el *insert* o el *delete*. Además permiten crear nuevas tareas de forma muy simple en una sola línea. En el proyecto ha sido empleado para crear los repositorios de acceso a los datos de pacientes y migrañas, así como a los datos de síntomas y sesiones asociadas a cada migraña.

3.3.4. Maven

También conocida como Apache Maven, es una herramienta cuyo objetivo principal es sintetizar y simplificar los procesos de generación de ejecutables a partir de los códigos fuente[26]. Antes de la aparición en 2001 de la herramienta de Maven, en cada proyecto existía una forma particular de compilar y crear los ejecutables a partir de código fuente[27]. Esto hacia perder mucho tiempo a los desarrolladores e incluso obligaba a tener a un desarrollador

exclusivamente dedicado para los procesos de *build*[4]. También, antes de Maven, cada vez que se quería cambiar una versión de alguna librería que emplease el código, como podían ser los frameworks de test, analizadores sintácticos, etc., había que detener todo el desarrollo de la aplicación hasta reajustar las nuevas dependencias de las versiones[28].

En el proyecto sirve para construir el ejecutable, que junto a Spring-boot se crea con un servidor Jetty embebido en la aplicación y con auto arranque. De esta forma, al ejecutar los comandos correspondientes de Maven, se ejecuta el servidor y este comienza a funcionar.

Para construir el ejecutable en la raíz del proyecto donde estará contenido el archivo pom.xml, se ejecuta el comando *mvn clean install*. Una vez creado el ejecutable, aparecerá una nueva carpeta, *target*, que contiene el ejecutable. Para arrancar el servicio, tan solo ejecutamos *mvn spring-boot:run*

3.3.5. Kotlin

Es un lenguaje de programación no tipado que corre sobre la máquina virtual de Java[29]. El mismo lenguaje infiere los tipos de las variables en tiempo de compilación para obtener así el mismo rendimiento que Java. Está diseñado para mantener la ***interoperabilidad*** con Java, lo que permite programar cada uno de los módulos con el lenguaje deseado. Actualmente es compatible con todas las librerías de Java conocidas, ya sea porque se han migrado a Kotlin o por la capacidad de Kotlin de interoperar con Java.

En el proyecto ha sido utilizado como lenguaje de programación principal. En particular, en el servidor el 100 % de los módulos están programados en Kotlin haciendo uso de los frameworks necesarios para crear microservicios con Spring, que tienen acceso a base de datos, y que hacen uso de un ORM, como Hibernate.

Para usarlo lo hemos combinado con Maven, permitiendo crear la base del proyecto en

[4] Los procesos de *build* son los procesos referentes a la compilación y creación de una aplicación completa para ser capaz de ejecutarse

apenas unos cuantos minutos.

Kotlin además de proveer una sintaxis más simple para la codificación de los objetos, proporciona nuevas palabras clave para facilitar algunas de las acciones más repetitivas de la programación orientada a objetos empleando Java, como es la creación de POJOS[30]. También ha sido enriquecido con mejoras del procesamiento de datos. Entre estas mejoras cabe destacar la propiedad de ***Null Safety***, o lo que es lo mismo la capacidad de crear un programa robusto que no sufra de la famosa excepción de *NullPointerException*[31]. Kotlin no elimina las excepciones completamente, pues aún es posible encontrarse con este tipo de excepciones en dos casos: si se lanza la excepción de manera explícita o si se usa los operadores `!!`. Los operadores `!!` indican en el código que una variable que puede ser nula se está asegurando que en el momento de acceso a sus datos esta no es nula. Estos dos casos son responsabilidad del programador ya que tanto este operador, como lanzar la excepción, son usados siendo conscientes de que puede provocar un *NullPointerException*.

Asimismo Kotlin es más ***rápido de compilar*** que Java, ya que es más ligero debido a que está optimizado y por tanto hace un mejor uso de los recursos de la máquina virtual de Java. Y añade la novedad de las ***co-rutinas***, que permiten crear secuencias asíncronas de llamadas que serán ejecutadas en el hilo principal de la aplicación y no requerirán de un procesado de la respuestas para poder interactuar con ellas desde el hilo principal[32].

3.4. Cliente

3.4.1. Kotlin multiplataforma

Ambos clientes están construidos con el paradigma de Kotlin multiplataforma[33]. En este modelo ambos clientes pueden compartir código que no es específica de ninguna plataforma, conocido como la parte común, y que a su vez puede invocar código externo específico para cada plataforma. Es decir, desde un código compartido en Kotlin, puede existir implemen-

taciones diferentes para determinados comportamientos para cada plataforma. Esto es muy útil para ciertas tareas como por ejemplo mantener un logger común, que es invocado desde el código compartido pero que necesita de implementaciones diferentes para poder funcionar en los clientes de Android e iOS.

El arquetipo de Kotlin multiplataforma se basa en la aplicación de los principios SOLID[34]. SOLID representa un acrónico de los 5 fundamentos básicos de la programación orientada a objetos y el diseño de los sistemas. Es Robert C. Martin quien acuña el significado de cada uno de las siglas, y de ello tenemos los siguientes principios[35]:

- **S**: Principio de responsabilidad única (Single responsibility principle - SRP): Define la responsabilidad de un objeto, el cual solo debe variar por una razón bien identificada. Es decir, un objeto debería tener el control sobre si mismo y variar solo en caso de ser necesario, no debería de tener la capacidad de mutar otros objetos. La responsabilidad de cambiar el objeto reside sobre el mismo objeto.

- **O**: Principio de abierto/cerrado (Open/closed principle - OCP): Un objeto debe ser abierto y cerrado en si mismo a la vez, es decir, un objeto debe permitir extender su funcionamiento a través de otra clase y a su vez no debe permitir modificarse. O sea, una clase debe ser abierta para permitir extenderse y cerrada para no permitir su modificación.

- **L**: Principio de sustitución de Liskov (Liskov sustitution principle - LSP): Dentro de una misma aplicación, un objeto puede ser sustituido por sus subclases, es decir, por las clases derivadas o extendidas de esta, sin alterar el comportamiento de la funcionalidad de la aplicación. De esta forma se permite extender la funcionalidad de la clase sin alterar las clases que ya hacen uso de este objeto.

- **I**: Principio de segregación de la interfaz (Interfaz segregation principle - ISP): La correcta separación de la funcionalidad pasa por la correcta definición de las interfaces.

El principio de segregación de la interfaz se refiere a la creación de las interfaces con las funcionalidades necesarias y suficientes. Dicho de otra manera, es mejor crear muchas interfaces con poca funcionalidad que una gran interfaz que recoja toda la funcionalidad. De esta manera los clientes se pueden ajustar convenientemente a sus necesidades e integrar solo lo necesario sin obligarles a tener que implementar métodos que nunca serán empleados.

- **D**: Principio de inversión de dependencias (Dependency inversion principle - DIP): este principio se explica con las dos siguientes definiciones:
  - Las clases de alto nivel, no deberían depender de las clases de bajo nivel. Ambas deberían depender de las abstracciones.
  - Las abstracciones no deberían depender de los detalles. Los detalles deberían depender de las abstracciones.

Como comentábamos, Kotlin multiplataforma está fundamentado en los principios SOLID, haciendo verdadero hincapié en el último de ellos, el principio de inversión de dependencias. Para entender el paradigma de Kotlin multiplataforma explicaremos con más detalle el problema que este principio intenta resolver. En cualquier modelo de programación, cuando un módulo depende de otro módulo, se crea una nueva instancia dentro del primer módulo y este será capaz de usar el segundo módulo sin problema.

En una aplicación, la lógica de negocio siempre es la parte más genérica del código, que depende de módulos que contienen los detalles de la implementación. Esto no permite cambiar fácilmente la lógica de la aplicación ya que queda muy acoplado a las dependencias que se crean. Además no quedan claras las dependencias, pues los módulos que contienen los detalles tienen a su vez dependencias de otros módulos que en la capa de lógica no son apreciables. Por lo que se crea un lío de dependencias en el que no quedan claras las definiciones o de dónde viene. Esto provoca que la lógica de negocio de la aplicación se

vuelva muy compleja de cambiar. Para poder resolver este problema se define el principio de inversión de dependencias. Pero, ¿cómo detectamos el problema?. Escribir tests ayuda a detectar la violación del problema, en cuanto no se pueda probar una clase con facilidad se está incurriendo en la violación del principio de inversión de dependencias. La solución más sencilla para evitar quebrar el principio es mediante la utilización de constructores y *setter*[5] que doten a los módulos dependientes de la funcionalidad necesaria para funcionar sin necesidad de inyección de dependencias[37].

Kotlin multiplataforma ofrece la posibilidad de usar los métodos tradicionales, mediante el uso de interfaces, para implementar la parte común de la aplicación. Además ofrece un mecanismo propio del lenguaje basado en las palabras expect/actual. Esta técnica dota al sistema de la capacidad de ofrecer declaraciones esperadas a través del módulo común, permitiendo proporcionar declaraciones reales que corresponden a las declaraciones esperadas. La declaración de elementos esperados se realiza mediante la utilización de la palabra clave *expect* y puede ser tanto de una función o de una clase completa. Para cada declaración esperada, en cada plataforma debe existir una implementación de cada uno de los elementos esperados. Para ello se debe utilizar la palabra clave *actual* antes del elemento esperado. Como toda técnica, este mecanismo también presenta ventajas e inconvenientes. Si lo enfrentamos al uso tradicional de interfaces, tenemos que, expect/actual otorga la posibilidad de definir el contrato mediante constructores y si existe una librería propia de una plataforma que se ajusta a las necesidades esperadas por la implementación propia de una plataforma, la librería puede ser usada mediante typealias sin necesidad de crear un wrapper, mapper o adapter. Por otro lado, el mecanismo de expect/actual define las dependencias como elementos estáticos, lo que elimina la posibilidad de realizar cambios en tiempo de ejecución. Esto es negativo, por ejemplo a la hora de crear test, ya que no es posible utilizar objetos en los test.

[5] En el lenguaje orientado a objetos se usa *setter* para referirse al método de acceso que permite asignar o modificar el valor de un atributo.[36]

3.4.2. Patrón de diseño del cliente

Para adaptar mejor Kotlin multiplataforma a los clientes, el desarrollo se ha realizado siguiendo el patrón de Modelo Vista Presentador (MVP). Este es una arquitectura derivada del patrón Modelo Vista Controlador (MVC) de Java. El modelo en el que se han basado los clientes es utilizado mayormente en el desarrollo de aplicaciones Android.

El patrón de diseño MVC se basa en separar la aplicación en tres capas de funcionalidad, el modelo, la vista y el controlador. Según los autores *Glenn E. Krasner* y *Stephen T. Pope*, pioneros en el fundamento del MVC:

> Model-View-Controller (MVC) programming IS the application of this three-way factoring. whereby objects of different classes take over the operations related to the application domain (the model). the display of the application's state (the view). and the user interaction with the model and the view (the controller).
>
> (A Cookbook for Using View-Controller User, Glenn E. K. y Pope S. T.)[38]

La capa de **modelo**, es la encargada de operar con el sistema. Gestiona tanto la consulta de los datos, como la actualización de los mismos. Además es la encargada de administrar los privilegios del sistema que se hayan descrito. También debe ejecutar las reglas de negocio asociadas a cada operación. Envía a la vista los datos necesarios en cada momento. Dentro de las capas de funcionalidad del modelo cliente-servidor, sería la capa encargada de ejecutar la funcionalidad o lógica de negocio.

La capa de **vista** es la capa encargada de interactuar con el usuario. Es la capa que representa la interfaz de usuario y debe ser cuidada al detalle para proporcionar a los usuarios una buena experiencia de usuario. En el paradigma de cliente-servidor se asocia a la funcionalidad de presentación.

La capa de **controlador** es la capa encargada de otorgar al presentador la información necesaria del modelo. Hace de puente entre el modelo y la vista y ayuda a mantener una

arquitectura de sistema limpia y sencilla, haciendo más mantenible la aplicación. Responde a cambios en el sistema, en forma de eventos o acciones de usuario que detonan peticiones a la capa de modelo. Además envía comandos a la vista si se solicita algún cambio en los datos del modelo, como actualización de datos de usuario. Dentro de las capas de cliente-servidor no es asociada a ninguna capa. Es un concepto nuevo que aparece para conectar las capas de modelo y vista.

MVP es la evolución del MVC[39]. Está orientado en la construcción de interfaces de usuario funcionales y bonitas, separando la lógica de negocio (modelo) de la interfaz de usuario (vista) utilizando para ello el presentador. La principal diferencia entre MVP y MVC reside en el comportamiento de la capa que actúa de conexión entre el modelo y la vista. En el caso de MVC, el controlador actúa de puente, mandando datos desde la vista al modelo y viceversa. En el caso de MVP, el presentador se encarga de dar formato a los datos recogidos desde el modelo para presentarlos correctamente a la vista, de modo que a la vista solo le llegue la información que necesita para "pintar" las pantallas. En MVP, el presentador permite a las capas de modelo y vista abstraerse de las necesidades de la otra. Mientras que en MVC el controlador no conoce las necesidades del modelo ni de la vista y son estas capas las encargadas de ajustar los datos para su posterior envío y gestión de los datos.

3.4.3. Descripción de los clientes

En el modelo de Kotlin multiplataforma aparece un nuevo módulo, el cual en el proyecto es conocido como módulo común (*common*)[40]. En esta pieza es donde reside la parte común de las aplicaciones, la capa de lógica de negocio a nivel de aplicación.

Para la implementación de esta pieza hemos seguido la estructura que se muestra en la Figura 3.5. Tal y como vemos, los clientes implementados se componen de cuatro partes separadas. Siguiendo el patrón Modelo Vista Presentador (MVP) compuesto de tres capas.

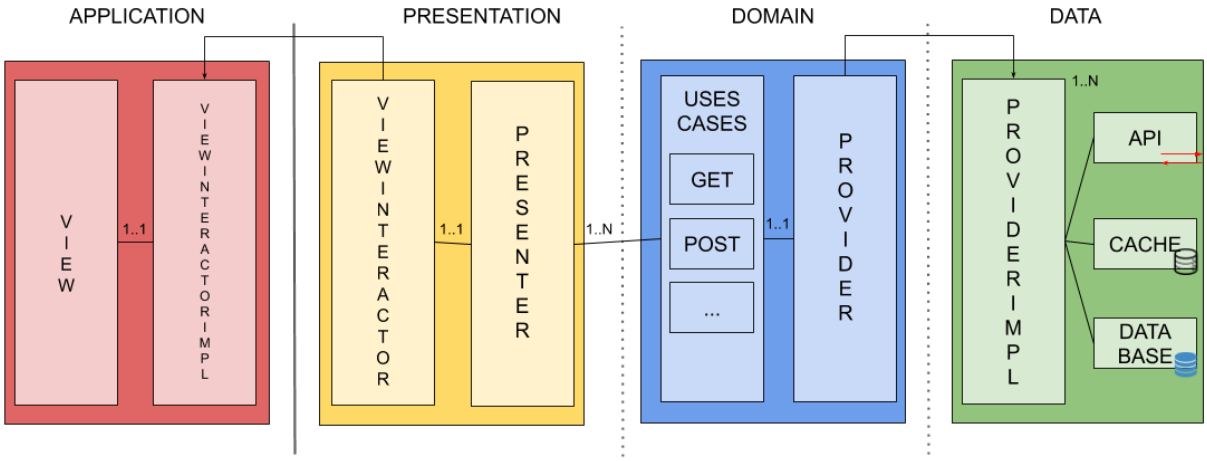

Figura 3.5: *Estructuras de las aplicaciones, en la figura se muestra las diferentes capas, las cuales algunas son compartidas en el módulo común y otras son exclusivas por plataforma*

El módulo común cubriría las necesidades de modelo y presentador, implementadas en un único punto y delegando la responsabilidad de implementar la vista a cada una de las plataformas. Estos clientes al contener solamente la vista, se realizan de manera sencilla con la funcionalidad necesaria para pintar y representar los datos.

En el módulo común recae toda la responsabilidad de gestión y administración de los datos en el nivel aplicación. En la Figura 3.5 podemos ver que el módulo común contiene tres partes y estamos hablando de que en este módulo se resuelven tan solo las capas de modelo y presentador. Entonces, ¿por qué se necesitan tres capas dentro del módulo común para cubrir las necesidades de dos tercios de la funcionalidad de la arquitectura MVP? En la arquitectura desarrollada durante el proyecto se busca alcanzar la mayor flexibilidad, el mejor mantenimiento y plantear el mejor paradigma posible conocido hasta la fecha. Por ello, aparece un nuevo concepto dentro del modelo MVP, que intenta mejorar, si es posible, el paradigma presentado. Los casos de uso, que aparecen dentro del la capa de dominio (Domain) otorgan al sistema de MVP una manera para crear código reutilizable por diferentes presentadores (presenter de la capa presentation) que permite gestionar y distribuir los datos por diferentes presentadores sin la necesidad de reutilizar un mismo

presentador para diferentes interfaces de la aplicación.

Para entender este concepto mejor, vamos a desgranar cada una de las piezas del módulo común. Empezando por las capas, el módulo se compone de cuatro piezas, que describimos a continuación.

Capa de Datos

En la Figura 3.5 de derecha a izquierda la primera capa con la que nos enfrentamos es la capa de datos (Data). La pieza de datos es la encargada de obtener los datos y propagarlos por el resto de las capas hasta llegar a la vista y finalmente ser representados. En la arquitectura empleada vemos que esta capa está compuesta por cuatro elementos. El elemento API son las clases, conocidas como servicios (*services*), encargadas de obtener los datos del servidor correspondiente. Estos servicios pueden conectarse con más de un servidor para alimentarse de diferentes fuentes de datos. En el caso ideal, seguido en el proyecto, los clientes solo se proveerán de datos del servidor de la aplicación y será el servidor de la aplicación el encargado de integrar los diferentes distribuidores de datos para enviarlos a los clientes.

El elemento de caché se encarga de almacenar los datos de uso continuo que no sufren de transformaciones a largo plazo. Es decir, los datos de consulta que apenas son modificados durante el uso de las sesiones de los usuarios. Datos como pueden ser los datos de usuario, datos de contacto de la aplicación o incluso datos de uso intensivo durante la sesión de usuario, como por ejemplo un token de seguridad para realizar llamadas al API. Para Kotlin multiplataforma la cache debe ser implementada para cada una de las plataformas por separado, pero esto es sencillo si se aplica la técnica expect/actual (véase 3.3.1).

El elemento *database* o base de datos es una pequeña base de datos que almacena datos necesarios para poder inicializar la aplicación normalmente, sin necesidad de realizar llamadas al API o de acceder a la cache. Se almacenan datos como el nombre de usuario registrado en ese cliente, así como los estados de los diferentes usuarios en la sesión. La base

de datos otorgará a los clientes la capacidad de recuperar las sesiones de los usuarios de forma rápida desde el punto en el que se quedaron.

Los tres elementos descritos anteriormente son elementos de consulta de datos, los cuales devuelven siempre un dato hacia el resto de las piezas. El último de los elementos es el *providerimpl*. Se trata de la implementación del provider o distribuidor. Este es requerido para poder distribuir los datos. En la arquitectura MVP, los datos son pedidos desde la vista al modelo, pasando por el presentador para atender a las necesidades de los usuarios. Por ello el modelo de datos leído debe ajustarse a los datos pedidos por el presentador. Y es el elemento de providerimpl el encargado de distribuir los datos al resto de piezas a través de la implementación del provider definido en la capa superior.

Capa de Dominio

Siguiendo de derecha a izquierda en la Figura 3.5 la siguiente capa que tenemos es la capa de Dominio (Domain). Esta capa representa un nuevo concepto dentro del modelo MVP, pues trata de resolver las carencias que aparecen al crear presentadores genéricos que representan varias vistas de usuario. Estas deficiencias se resumen en que los presentadores se cargan de demasiada funcionalidad para dar soporte a varias vistas, cuando lo ideal, es tener un presentador por cada una de las vistas. De esta forma si se quieren añadir o quitar vistas se hace completamente sin afectar a ninguna otra vista.

La capa de dominio se compone de dos elementos. El primero son los casos de uso, que definen cada una de las acciones a realizar por los presentadores. Los casos de uso se definen con el verbo del protocolo y una palabra que identifique la acciones a realizar. Por ejemplo, la acción de login sería definida con el verbo POST más la acción login, entonces quedaría PostLogin. El segundo elemento, el provider o distribuidor, crea un contrato con la la capa de datos mediante interfaces que representan a cada uno de los distribuidores necesarios para implementar cada uno de los casos de uso. Por cada acción que aparece en los distribuidores

se crea un caso de uso único.

Capa Presentador

La tercera de las capas, la capa presentador (presentation) es la última de las capas perteneciente al módulo común. En esta pieza nos encontramos con dos elementos, ambos muy conectados por su funcionalidad. El primero de ellos, el presentador, contiene uno o más casos de uso. Por cada caso de uso el presentador contiene una o varias acciones a realizar. Estas dependen de las posibles respuestas que se apliquen a cada acción. Siguiendo con el ejemplo de la acción de login, introducido en la capa anterior, para este caso de uso tendremos al menos dos posibles escenarios a implementar en el presentador. La respuesta correcta, cuando el login ha sido resuelto satisfactoriamente, y la respuesta de error, cuando por algún motivo no ha sido posible realizar login.

El número de acciones implementadas en el presentador por cada caso de uso varía. Por norma, cuanto mayor es el número de respuestas implementadas por cada acción, mayor será el detalle con el que se podrá definir la vista asociada. De nuevo, en el ejemplo del login, al tener solo dos posibles respuestas, en la vista solo podremos avanzar de pantalla en caso de recibir una respuesta correcta. O mostrar un error genérico al usuario, con un mensaje de imposible realizar login en caso de fallo. Si el presentador, en lugar de implementar dos respuestas, implementase tres respuestas (añadiendo una nueva respuesta que diferencia el caso en el que el login no ha sido posible realizarse debido a que la contraseña es incorrecta), entonces la vista podría tratar este error de forma diferente y llevar al usuario a volver a introducir su contraseña, indicando que la empleada en ese momento es incorrecta. De esta forma el usuario obtendría una mejor experiencia en el uso, ya que sabría qué tiene que hacer.

También es el presentador el encargado de transformar los datos para adaptarlos a las necesidades de la vista. Las vistas son definidas mediante los interactuadores de las vistas

(*viewinteractor*). Estos son especificados mediante una interfaz en la capa de presentación. En este interactuador se define el modelo de la respuesta que la vista recibirá. El actuador de la vista también evita que la vista se acople al presentador, de esta forma se concede la posibilidad de evolucionar los presentadores sin afectar a las vistas, ya sea porque a un presentador se le añade nueva funcionalidad o porque se decida crear una nueva versión del presentador.

Capa de Vista

La capa de vista es única para cada plataforma. En cada implementación aparece una interfaz que será única para cada pantalla y que está asociada de forma única con un presentador. Es decir, existe una relación uno a uno entre una pantalla y un presentador. En esta capa aparecen pequeñas lógicas para navegar entre las pantallas y crear una experiencia de usuario satisfactoria, pero en ningún momento aparecerán lógicas que impliquen transformaciones de datos o cálculos complejos. Todos los datos son extraídos de la implementación del interactuador de la vista, el cual se ajusta al contrato definido mediante la interfaz en la capa de presentación.

Cada implementación por plataforma se realiza en el lenguaje correspondiente a la plataforma. De esta forma, para Android la vista será desarrollada en Java/Android o en Kotlin/Android. Mientras que para iOS las vistas serán desarrolladas en Swift. Esto impide que las vistas sean compartidas por las diferentes plataformas. Todo esto es posible debido a que Kotlin multiplataforma compila el módulo común o núcleo en lenguaje máquina creando librerías para cada plataforma (véase 3.4.1).

Capítulo 4

Implementación

En este capítulo se explican los conceptos que se han empleado para el desarrollo de cada una de las piezas que componen la aplicación de visualización y seguimiento de pacientes en estudio clínico. Para proporcionar una aplicación de escalado fácil se ha desgranado la aplicación. Separar todas las piezas también permite realizar un mantenimiento sencillo por parte del personal técnico que es capaz de localizar y solventar las incidencias en cada uno de los módulos que componen la aplicación.

Además, para que el desarrollo futuro pueda realizarse rápida y cómodamente, en este capítulo se explican las pautas básicas que un desarrollador debe seguir para no romper la arquitectura de cada uno de los componentes.

4.1. Base de datos

Se trata de una base de datos relacional creada en MySQL gestionada por los administradores del servidor, responsables de cargar, mantener y gestionar los datos. Para entender mejor la importancia de la base de datos durante esta sección se explican los pasos llevados a cabo para el Caso de Uso de la Migraña detallado en el Capítulo 4.

Existen scripts y procesos para automatizar las tareas de limpieza y carga de la infor-

mación, ya que los datos son extraídos de diferentes herramientas (véase 5.2.1). Durante el proyecto la base de datos se ha ido evolucionando para, entre otras cosas, crear claves foráneas que relacionan las tablas correctamente. Además, se ha creado la tabla de pacientes para almacenar la información básica de los pacientes. Al comienzo del proyecto, solo se almacenaba la información sociodemográfica de los pacientes, pero con cada una de las iteraciones se ha ido ajustando la base de datos hasta el resultado mostrado en la Figura 4.1.

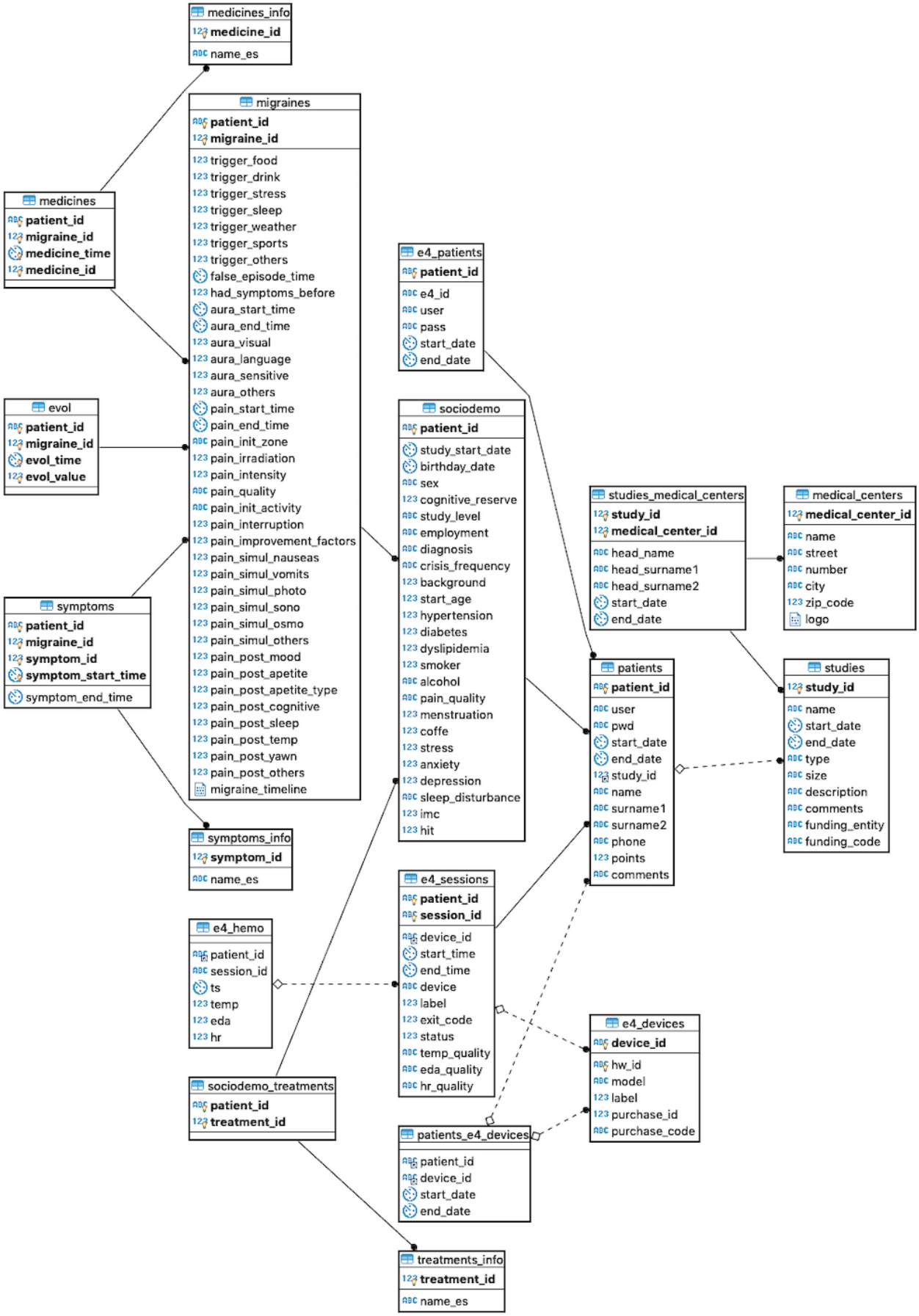

Figura 4.1: *Diagrama de la base de datos.*

Al alimentarse de dos fuentes diferentes de datos, en la base de datos podemos ver que existe cierta separación entre las tablas que almacenan la información recogida por el dispositivo E4 y la información que es cargada a mano por los usuarios desde la aplicación. Para diferenciar las tablas, las que se corresponden con la pulsera comienzan con el prefijo *e4_* indicando así que esas tablas guardan la información de la pulsera, mientras que las tablas que no tienen ningún prefijo son las tablas que almacenan la información de la crisis migrañosas. Estas crisis son cargadas por los pacientes usando la aplicación.

Antes de iterar sobre la base de datos esta separación de tablas era más evidente, pues no existía la tabla de pacientes que relaciona ambos dispositivos. La tabla fue creada para solventar esta segregación en la base de datos, de modo que ahora una crisis es relacionada completamente a través de las tablas mediante claves de paciente.

Toda la información gestionada en la base de datos gira en torno a tres tablas: *migraines*, *sociodemo* y *patients*. Estas tres tablas almacenan de forma única las crisis que sufren los usuarios. Si durante una crisis no existe registro en alguna de ellas, la carga de datos no se ha completado satisfactoriamente y es necesario revisar qué salió mal.

Partiendo de la tabla de migrañas vemos que existen tres tablas que dependen totalmente de una migraña y un usuario, así como del momento en el que se produce la migraña. Por ejemplo, en la tabla *symptoms* encontramos que existe una relación con la tabla migrañas de cuatro columnas: el identificador de paciente, el identificador de la migraña, el identificador del síntoma y el momento en el que empezó el síntoma. Con esta relación es posible asociar cada síntoma con una migraña de un usuario en concreto, así como averiguar en qué momento se produjo dicho síntoma. Además, los síntomas son definidos completamente en la base de datos permitiendo asociarles un nombre mediante la tabla *symptoms_info*. Para la tabla de medicinas (*medicines*) y evolución (*evol*) la lógica empleada es la misma que para la tabla de síntomas, con la pequeña diferencia de que la tabla donde se registra la evolución de la crisis no está asociada a un elemento concreto como el nombre del síntoma o de la medicina.

Siguiendo con el núcleo principal, observamos la tabla *sociodemo_treatments* que almacena los tratamientos que se aplican a cada uno de los factores que pueden afectar al desarrollo de la migraña. De modo que los tratamientos son diferentes para cada paciente, pero no para cada migraña. Por ello, en este punto, la relación se hace mediante el identificador de paciente. Al igual que los síntomas y las medicinas, existe una tabla descriptiva del tratamiento que almacena el nombre del tratamiento (*treatments_info*).

En última instancia encontramos la tabla de pacientes (*patients*) que completa el núcleo. Sobre esta tabla recae la responsabilidad de asociar los pacientes a los estudios mediante la tabla *studies*. La tabla de estudios almacena la información relacionada con el estudio al que pertenece el paciente. Los pacientes pueden pertenecer a uno o más estudios. Además, para completar la información del estudio, estos se relacionan mediante la tabla *studies_medical_centers* con los centros médicos que participan en cada estudio, permitiendo así que existan estudios multicéntricos. Para terminar de almacenar la información relacionada con los estudios, observamos la tabla *medical_centers* que almacena la información de los centros médicos, es decir de los hospitales y/o centros de investigación que están involucrados en cada uno de los estudios.

Por otro lado, la tabla de pacientes relaciona toda la información recogida por la pulsera E4 durante las crisis de los pacientes. En este punto encontramos dos relaciones separadas. En primer lugar tenemos la tabla *e4_patients* que almacena la información de los pacientes para la pulsera. Las pulseras E4 son asociadas a un identificador único, pero una pulsera no es asociada a un paciente de forma indefinida. Por ello, es necesario conocer qué paciente lleva qué pulsera en cada momento. Gracias a esta tabla podemos llevar el seguimiento de dicha tarea. En segundo lugar, nos encontramos con la tabla *e4_sessions* que almacena la información relacionada con las crisis. Una sesión almacena la información de las variables hemodinámicas durante las crisis de los pacientes. Esta información es necesaria para la creación de modelos predictivos, ya que estos son las que marcan si una crisis es buena o no para su carga dentro del modelo de predicción.

Por último, nos encontramos con la relación doble, de los dispositivos con los pacientes y las sesiones. Cada pulsera tiene una serie de características que es necesario conocer para poder obtener los datos de la plataforma donde E4 almacena los resultados. Por ello, se decidió guardar la información de las pulseras en base de datos, ya que es una información invariante que en ciertos momentos de confusión de datos puede resultar útil para volver a cargar datos o comprobar que los datos asociados a un paciente coinciden con el dispositivo.

4.2. Servidor

El servidor ha sido desarrollado siguiendo la arquitectura de microservicios mediante Spring framework[14]. Cada servicio lleva embebido un servidor Jetty. Al ser un servidor creado completamente en Java soporta perfectamente los microservicios, además de estar optimizado para levantarse con tan solo las necesidades básicas de un servicio. sin cargar complementos adicionales que conllevan a una ejecución más lenta y pesada del servicio[41].

Cada servicio está desarrollado en su totalidad en el lenguaje Kotlin, y ha sido desarrollado empleando la herramienta IntelliJ IDEA. Esta herramienta es un entorno de desarrollo integrado, creado por JetBrains, que además son los responsables de la creación del lenguaje de programación Kotlin, por lo que el entorno se ajusta perfectamente a las necesidades del proyecto. El IDE[1] proporciona un complemento, que entre otras cosas, detecta errores semánticos en código, ayuda con la inyección automática de dependencias, permite la navegación del proyecto a través de los métodos de las clases, y ordena y estructura el código para proporcionar una lectura sencilla, etc.

[1] IDE de sus siglas en inglés Integrated Development Environment, es un entorno de desarrollo integrado para aplicaciones informáticas

4.2.1. Paquetes del proyecto

Para facilitar el desarrollo y mantenimiento, el servicio está divido en paquetes que ordenan los diferentes elementos del servidor.

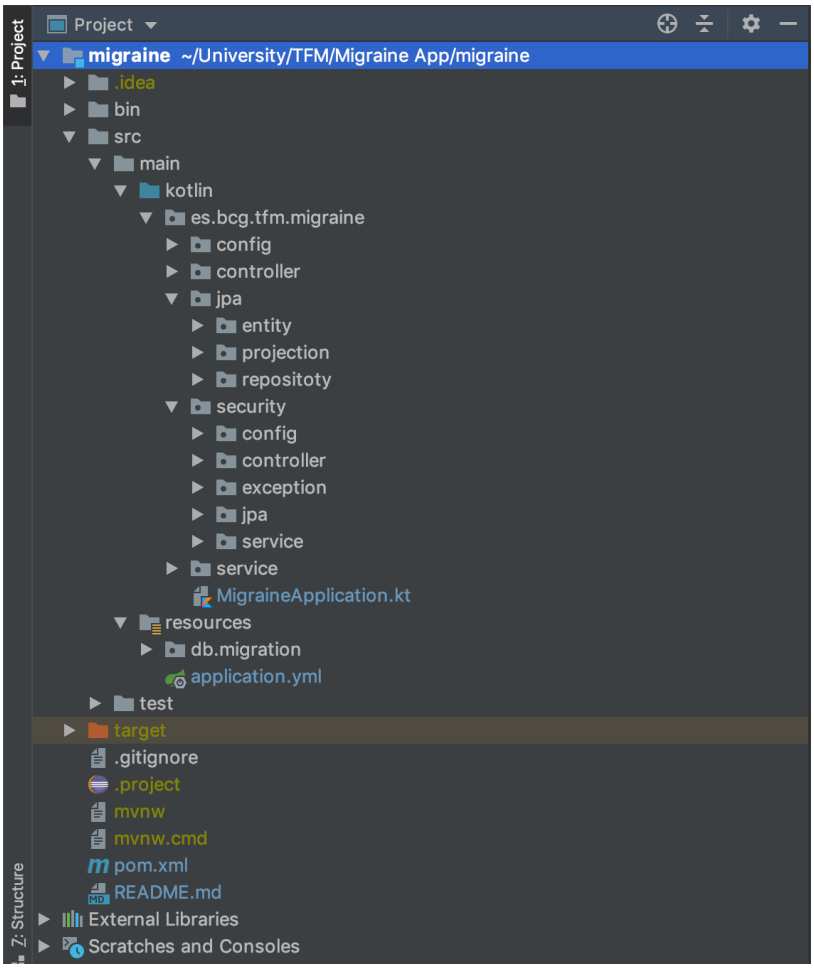

Figura 4.2: *Estructura de los paquetes del servidor.*

En la Figura 4.2 aparece *MigraineApplication.kt* en la raíz del proyecto bajo la jerarquía de los paquetes principales que indican los aspectos básicos del proyecto. Esta clase es ubicada en la raíz ya que es la clase principal desde la cual se ejecuta el proyecto de Spring-boot. Para que la clase sea entendida como la clase principal es necesario anotarla con *@SpringBootApplication* indicando que este será el punto de arranque de la aplicación. Kotlin como es de esperar simplifica la creación de estas clases reduciéndolas tal y como

podemos ver en el ejemplo del Apéndice A.1.

Paquete config

En la Figura 4.2 vemos que el servidor ha sido dividido en cinco partes. El primer paquete que nos encontramos, *config*, contiene la configuración del servicio. Al ser un servicio desarrollado con Spring-boot, las clases de configuración son anotadas con la anotación *@Configuration* para indiciar que son una configuración. Mediante esta anotación, Spring carga estas configuraciones antes de cargar los beans o componentes de la aplicación. En el caso de este proyecto no ha sido necesario crear ningún bean especial ya que no se requiere de ninguna configuración especial y es válida la que trae Spring por defecto. Como vemos en la parte de configuración encontramos una sola clase de configuración. Esta clase contiene la configuración necesaria para crear el Docker que contiene el Swagger.

Un Swagger es una herramienta que proporciona una manera de hacer accesible las APIs creadas en Java de forma estándar[42]. Para la creación de Swagger es necesario configurarlo correctamente, en este caso mediante la clase SwaggerConfig, que contiene la configuración necesaria para que el Swagger sea levantado a la vez que se levanta el servicio sin interferir en los recursos del propio servicio.

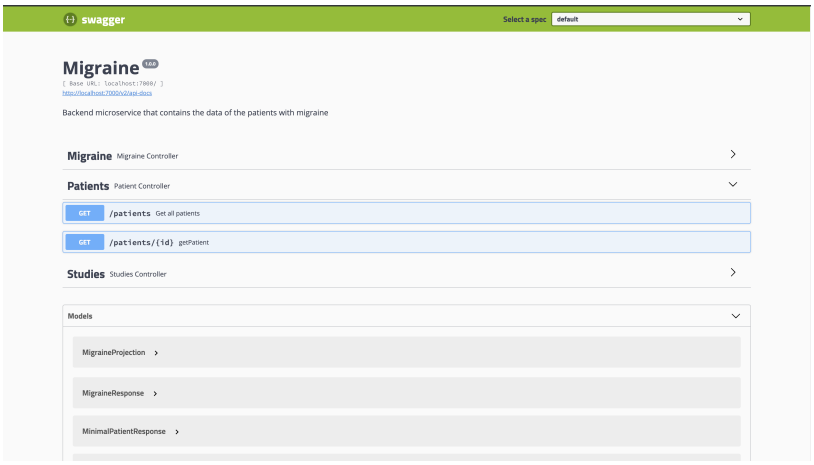

Figura 4.3: *Ejemplo de swagger.*

En la Figura 4.3 podemos ver el Swagger creado para el servicio *migraine*. En este caso vemos que aparecen varias secciones. Cada una de ellas atiende a uno de los recursos que se proporcionan. En cada sección aparecen una serie de endpoints. Pulsando sobre los endpoints se extiende un ejemplo que puede ser ejecutado para aprender a usar la API de forma intuitiva. Swagger también proporciona los modelos de datos que son necesarios para poder enviar y recibir respuestas correctamente. De esta forma los JSON son más fáciles de realizar ya que son especificados de forma completa, indicando si los parámetros son opcionales o no y el tipo de datos que esperan recibir.

Paquete controller

El siguiente paquete que nos encontramos, *controller*, contiene los controladores. Los controladores son clases que mapean los endpoints expuestos por el servicio. Para hacer más mantenible la aplicación, se han separado los endpoints expuestos en diferentes controladores. Para una correcta organización de los mismos se decidió crear un controlador por cada uno de los recursos a los que hace referencia. De esta forma nos encontramos con tres controladores diferentes, uno para migrañas, otro para pacientes y otro para estudios. El primer controlador atiende las necesidades de las migrañas exponiendo endpoints de consulta de migrañas por paciente. El segundo controlador expone endpoints consultivos para los pacientes. Y por último el controlador de estudios se creó para cubrir las consultas de información sobre los estudios.

Para el desarrollo de un controlador, siguiendo el patrón de Spring y empleando los recursos que proporciona el framework, se requiere del uso de la anotación *@RestController* para indicar que la clase se levantará como un controlador REST. Un controlador puede o no exponer endpoints, y si los tiene se levantan de manera automática. Para explotar al máximo el potencial de la herramienta de Swagger es necesario añadir una serie de anotaciones que ayudan al mapeo de los endpoints, así como al uso del Swagger.

En el Apéndice A.2 podemos ver un ejemplo de controlador. En este caso, se trata del controlador de pacientes. La clase comienza con el nombre del paquete y la importación de las dependencias necesarias para el correcto funcionamiento de la clase, tal y como se haría en una clase Java. Tras esto, aparecen dos anotaciones, la primera *@Api* marca la etiqueta a la que serán asociados los endpoints para su ordenación en el Swagger. La segunda anotación indica que es un controlador REST.

Un controlador hace uso de un servicio, que es el encargado de ejecutar la lógica de negocio asociada a un endpoint. Para poder usar el servicio dentro de un controlador, es necesario instanciar el servicio. Para esto, con la ayuda de Spring tan solo es necesario anotar el servicio con *@Autowired*. En Kotlin para poder cargar una variable en tiempo de ejecución es necesario que se declare la variable como *var*, es decir, variable mutable, y con *lateinit*, es decir, variable que será inicializada tras crear la clase. Recordemos que Kotlin es un lenguaje no tipado, y que para poder obtener un rendimiento igual al de Java, las variables son sustituidas en tiempo de compilación (véase 3.3 y 3.4.1).

Para la declaración de los endpoints tan solo hay que crear una función mediante la declaración *fun*, el nombre que se le quiere asignar al método y el tipo que devolverá. Una vez tenemos la función creada, se anota con *@RequestMapping* y se configuran los parámetros necesarios. En el Apéndice A.2 podemos ver el ejemplo de dos endpoints expuestos en el controlador. Tal y como vemos las funciones contienen argumentos. Estos son los parámetros de entrada de los endpoints. En este ejemplo, la primera función contiene un **query param** y en la segunda un *path param*. Para completar el Swagger se puede añadir una descripción a los endpoints mediante la anotación *@ApiOperation*, de modo, que al lado del endpoint, aparezca una pequeña descripción de para qué sirve (Figura 4.3).

Paquete jpa

El paquete *jpa* contiene las entidades y los repositorios empleados para el acceso a la base de datos. Para mapear las entidades se hace uso del framework de Hibernate. Este proporciona anotaciones que convierten el proceso de creación de entidades en un proceso trivial. Además, gracias al potencial de Kotlin y su incorporación de la palabra clave *data*, no es necesario crear las funciones básicas de la entidad quedando una entidad muy sencilla.

Tal y como encontramos en el Apéndice A.3, la declaración de la entidad se hace mediante anotaciones que mapean la tabla a un objeto de manera sencilla. También permite crear la entidad con las restricciones correspondientes. El ejemplo es la entidad que mapea la tabla que contiene los estudios de la aplicación. Esta tabla, llamada *studies* es referenciada a través de la anotación *Entity* y enlazada de manera automática con la tabla. Para indicar que un atributo de la clase hace referencia a la clave primaria de la clase, únicamente se requiere anotar el atributo con *@Id*.

Vemos que en la clase aparecen atributos anotados con *@Column* y otros que no se han marcado con nada. Hibernate es un framework que trata de facilitar la vida a los desarrolladores. Para ello es capaz de mapear entidades a través de la declaración de atributos. De forma que un atributo declarado sin la anotación serán mapeados con la declaración más sencilla. Es decir, con el tipo inferido y sin ninguna restricción adicional. Los atributos que aparecen anotados con *@Column* tienen una serie de restricciones que es conveniente indicar para evitar tener que llegar a la base de datos para fallar por incumplimiento de alguna de las limitaciones. La anotación permite enlazar el nombre de la columna a la que hace referencia con un atributo con nombre diferente, así como indicar una serie de restricciones, tales como: si la columna puede ser nula, el tamaño que tiene esa columna, si se permite actualizar o insertar en la columna a la que hace referencia, etc. Cabe destacar, que para declarar una enumeración se realiza con la misma anotación indicando que la columna está declarada con una definición concreta.

Los repositorios en cambio, son creados como interfaces gracias a la gran potencia que ofrece Spring-data. Para crear un repositorio es necesario extender una interfaz a la que se le indican dos atributos. De izquierda a derecha, el primero indica la clase que mapea la tabla y el segundo la clave primaria que será necesario para acceder a determinados datos.

Al extender JpaRepository se proporcionan varios métodos por defecto que permiten realizar acciones a partir del identificador. También se pueden declarar nuevas acciones (consultas) usando una convención de nomenclatura inventada por Spring y fácil de entender. Permite extender distintos repositorios base, atendiendo a diferentes necesidades. Si a través de la nomenclatura proporcionada no alcanzamos la consulta necesaria, podemos recurrir al recurso de la anotación *@Query* que permite realizar cualquier tipo de consulta. En nuestro caso podemos ver cómo para el repositorio de migrañas se ha creado una consulta que devuelve una proyección de la unión de varias tablas, con los datos necesarios para analizar una migraña. (véase A.4)

Paquete service

El paquete *service* contiene los servicios del servidor. Los servicios son los encargados de conectar todas las piezas, ejecutar la lógica de negocio y devolver datos a los controladores, para que estos puedan responder. En los servicios es donde más atención hay que poner a la hora de desarrollar, pues una mala lógica, una excepción no controlada o una llamada mal realizada puede provocar que el servidor quede colgado. Mediante la potencia de Kotlin y la capacidad de *null safety* para evitar los nullpointers el servicio queda bastante más protegido contra errores inesperados.

Los servicios son definidos mediante interfaces que definen el contrato entre el controlador y el servicio. Esto se realiza debido a que los servicios son focos de muchos cambios ya que la lógica de negocio está en continuo desarrollo. Aunque los endpoints expuestos no deben cambiar de cara a los clientes, ya que esto puede provocar que alguna versión de las

aplicaciones deje de funcionar correctamente. Por ello, cuando una función de un servicio cambia tanto que es necesario re-declarar las funciones del servicio, es momento de crear nuevas versiones de los endpoints.

En el Apéndice A.5 tenemos un ejemplo de implementación de servicio. En él vemos el servicio que contiene la lógica de los recursos de los estudios. En este caso solo es necesario hacer uso del repositorio de estudios, debido a que solo requiere acceder a la información de los estudios. Como podemos ver, se declara un mapeador como una función lambda que convierte el objeto entidad en el objeto de respuesta que espera el controlador. El resto del código define la lógica de negocio a ejecutar. En este caso, es una lógica sencilla que accede a base de datos mediante el repositorio y transforma las entidades en objetos de respuesta para devolverlos al controlador.

Paquete security

EL último paquete que nos encontramos, es el paquete de *security*. En este se encuentran todas las clases relacionadas con la seguridad del servicio. En la Figura 4.2 podemos ver que dentro del paquete de seguridad cuelga una estructura de subpaquetes similar a la explicada para el servicio. Esto es debido a que el paquete de seguridad puede llegar a constituir un servicio en sí mismo que debido a las dimensiones del proyecto y sobretodo por mantenimiento del servidor se decidió incluir como una funcionalidad del servicio principal.

Este paquete proporciona seguridad de tipo Oauth2 a las peticiones. La Seguridad Oauth2 se ha basado en la creación de un token de usuario, de tipo JWT, que se crea a la hora de hacer el login con un usuario y se utiliza para realizar el resto de peticiones[43]. En el token viaja la información de qué usuario está realizando las peticiones y otros datos importantes como el momento de creación del token, el tiempo de duración del mismo, etc. Estos datos sirven, primero para verificar si el token es valido cuando se realiza la petición. Y segundo, para guardar un registro de uso de la aplicación por los usuarios.

El JWT o JSON Web Token, es un estándar basado en JSON para la creación de tokens de acceso que permite la propagación de privilegios e identidades a través de las peticiones web. La estructura del token se compone de tres partes, cabecera o *header*, contenido o *payload* y firma o *signature*. La cabecera contiene la información relativa al algoritmo que se usó para firmar el token. El contenido contiene los privilegios o *claims* que proporcionan la información que se quiere propagar en el token. Y la firma, se calcula codificando la cabecera y el contenido en base 64. En el RFC 7519 se definen una serie de atributos con una nomenclatura de tres letras que tienen un determinado significado. Por ejemplo, *exp* hace referencia al tiempo de expiración del token, identificando a partir de qué momento el token deja de ser válido[44].

4.2.2. Archivos de configuración

Los ficheros de configuración son archivos necesarios para conseguir que el servicio implementado con Spring y Maven funcione correctamente. En ellos se encuentran las propiedades y demás configuraciones del servicio. En este apartado se explican los ficheros de configuración incluidos en este servicio. Podrían aparecer más ficheros de configuración dependiendo de las necesidades del servicio. Los ficheros explicados en esta sección, son los archivos que como mínimo deben contener un servicio implementado con Spring y gestionado a través de Maven.

application.yml

El fichero .yml o .yalm es un sustituto de los fichero de propiedades clásicos, los .properties, que definen las propiedades del servicio[45]. Estas propiedades son leídas y propagadas por el servicio en el momento de arranque del sistema. De modo que para modificar, cambiar o añadir una propiedad y que la modificación se haga efectiva es necesario, además de realizar los cambios necesarios en el fichero de propiedades, reiniciar el servicio para que la

aplicación conozca las nuevas configuraciones o propiedades. La principal diferencia entre los ficheros *.yml* y los *.properties* es la forma en la que se estructuran los datos. En los archivos .properties las propiedades son definidas en una misma línea, por ejemplo para definir los datos básicos para conectarse con la base de datos, necesitaríamos tres propiedades: la URL, el usuario y la contraseña. En este caso quedaría con la siguiente forma:

```
...
spring.datasource.url = jdbc:mysql://localhost:3306/migraine
spring.datasource.username = mysql
spring.datasource.password = mysqlpass
...
```

En cambio en el archivo de configuración .yml las propiedades son definidas de forma jerárquica, donde no es necesario repetir las categorías a las que pertenece la propiedad. Siguiendo con las propiedades que son necesarias para conectar la base de datos con el servicio, obtendríamos el siguiente resultado:

```
...
spring:
    datasource:
        url: jdbc:mysql://localhost:3306/migraine
        username: mysql
        password: mysqlpass
...
```

La principal ventaja que presentan los archivos .yml frente a los ficheros .properties es la legibilidad del documento cuando el número de propiedades comienza a hacerse muy elevado. En los .properties al definir cada una de las propiedades sin depender de ninguna jerarquía, aparece la posibilidad de crear propiedades relacionadas en diferentes partes del fichero, aumentando más la complejidad de lectura. En cambio en los ficheros .yml no se permite definir varias veces la misma jerarquía, obligando a definir todas las propiedades pertenecientes a una misma categoría en un mismo punto.

En el Apéndice A.6 encontramos el fichero de configuración empleado en el servicio de *migraine*. En este archivo podemos ver, tal y como se explica en los párrafos anteriores, que la configuración queda agrupada bajo la jerarquía de categorías creada, de modo que todo queda agrupado exponiendo las propiedades de forma simple y limpia.

pom.xml

El archivo POM (de sus siglas en inglés Project Object Model, Modelo de Objeto de Proyecto) es el fichero necesario para el funcionamiento de Maven[46]. Es un archivo XML, estructurado mediante etiquetas, que contiene la información necesaria para construir y configurar el proyecto a través de Maven. Incluye valores por defecto para la mayor parte de la configuración, como las rutas de donde leer los archivos fuente o las carpetas donde crear y almacenar los ejecutables. Las configuraciones más importantes que se pueden especificar en el POM son las dependencias del proyecto, los complementos, los objetivos (*goals*) y los perfiles de configuración.

Las **dependencias**, hacen referencia a las librerías externas que serán importadas por el proyecto para su compilación y ejecución. Para poder importar las librerías externas, deben existir en algún repositorio, ya sea en el repositorio local donde se está compilando el proyecto o en algún repositorio alojado en la nube. El repositorio más conocido para almacenar librerías Java es *https://mvnrepository.com/repos/central*.

Los **complementos o plugins** dotan al proyecto de la capacidad de ejecutar y crear nuevos elementos que sin la aparición de Maven era necesario ejecutarlos a parte. Por ejemplo, a través de plugins podemos ejecutar scripts que se ejecuten en el momento de construcción elegido por el administrador del proyecto. Por ejemplo, se puede ejecutar un script para la creación de un contenedor Docker que almacene el proyecto tras ser construido mediante el plugin de Spotify[2].

[2] https://github.com/spotify/docker-maven-plugin

Los ***goals* u objetivos**, marcan el ciclo de vida de construcción de un proyecto a través de Maven[47]. Existen tres ciclos de vida predeterminados en el proyecto; por defecto (*default*), limpio (*clean*) y sitio (*site*). Cada uno de ellos se encarga de una tarea. El ciclo de vida por defecto es el responsable de compilar y construir el proyecto utilizando la implementación del proyecto. El ciclo de vida limpio maneja la limpieza del proyecto. Y el ciclo de vida sitio, se encarga de la creación de la documentación del proyecto.

Cada ciclo de vida está definido por una serie de fases, donde una fase representa una etapa del ciclo de vida[48]. Existen 23 fases diferentes[3] que se combinan de diferentes maneras para cada ciclo de vida. Por ejemplo, en el ciclo de vida por defecto nos encontramos con siete fases, que van desde la validación del código, hasta el despliegue la de aplicación. Para ejecutar cada uno de los objetivos existen comandos. Los comandos son construidos con la nomenclatura *mvn plugin:goal*, donde el primer elemento marca el plugin que se empleará para la ejecución y el segundo elemento marca el objetivo a realizar. De esta forma algunos de los comandos más famosos que nos encontramos son: *mvn clean:clean*, *mvn install:install*, *mvn deploy:deploy*, *mvn site:site*, *mvn deploy:deploy*, *mvn site:deploy*. Para evitar tener que escribir el comando completo, cuando plugin y objetivo coinciden los comandos pueden ejecutarse escribiendo solo la palabra clave, así por ejemplo para ejecutar *mvn clean:clean* se podría ejecutar usando *mvn clean*.

Los **perfiles o *profiles*** son creados para proporcionar diferentes configuraciones en base a determinadas características[49]. En entornos de desarrollo cerrados es común que aparezcan perfiles de construcciones donde las dependencias son adquiridas de un repositorio privado que es actualizado manualmente de forma que se controlen cada una de las librerías que emplea el proyecto. De esta forma se impide que se usen librerías con mala reputación y de desarrolladores poco conocidos. Cuando Maven es empleado por empresas durante sus diferentes fases, como son desarrollo y producción es común que existan perfiles para diferenciar donde se van a crear los artefactos o validar las dependencias que se usan para

[3] Podemos encontrar todos en https://maven.apache.org/ref/3.6.1/maven-core/lifecycles.html

producción, por ejemplo evitando emplear snapshots.

4.3. Cliente

Los clientes han sido desarrollados con Kotlin multiplataforma. Siguiendo la lógica de separar las interfaces nos encontramos con dos piezas independientes para la implementación de los clientes. El módulo común y las interfaces por plataforma. En esta sección profundizaremos en los desarrollos de cada una de las piezas sin entrar en los detalles arquitectónicos explicados en el capitulo anterior (véase 3.4.3).

4.3.1. Modulo común

La Figura 4.4 muestra la estructura de paquetes del módulo común. Como se puede ver se diferencian tres grandes categorías, *data*, *domain* y *presentation*. Estas capas ya son explicadas con sus características en la Sección 3.4.3.

Cada una de las capas es tratada por separado tanto como se puede durante todo el desarrollo, aunque se debe tener en cuenta que las capas se conectan entre sí, ya sea mediante interfaces o mediante instancia de objetos creados en otras capas. Esto hace que la división de las capas a menudo se haga compleja. Por otro lado, para facilitar el mantenimiento y escalabilidad del proyecto, en cada una de las capas se incluye un paquete *core*, que contiene elementos base y genéricos que tratan de fijar unas pautas de desarrollo para futuras implementaciones.

Data

En la Figura 4.5 podemos diferenciar en el primer nivel (justo debajo de data), que aparecen cinco grandes categorías de empaquetamiento. ***Core***, en este caso se compone de

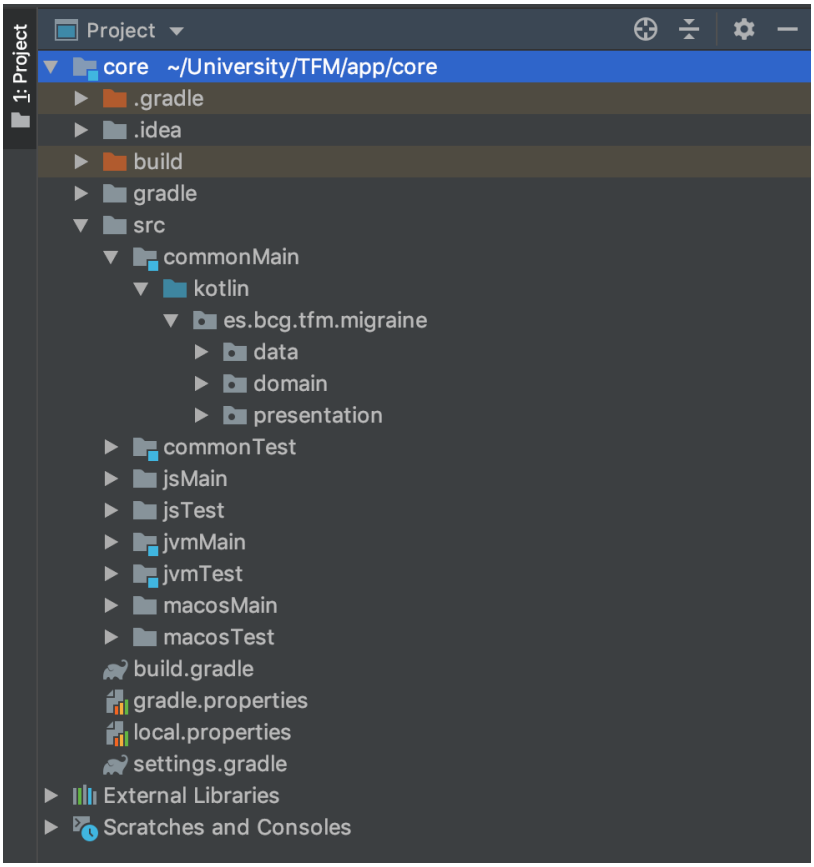

Figura 4.4: *Estructura de paquetes del modulo común.*

un servicio[4] con métodos comunes para realizar llamadas al API y una clase Call[5], que sirve de respuesta común para hacer llamadas. De modo que empleando genéricos el mapeo de datos se haga correctamente.

Entity almacena los modelos de entrada y salida usado por los servicios para realizar llamadas al API. ***Service***, como se adelanta, almacena los servicios que realizan las llamadas al API, son clases similares a las fachadas[6] de un servicio, con la ligera diferencia que en este caso, si se requiere de alguna transformación de datos se realizarán en el mismo servicio.

El paquete ***local*** contiene las implementaciones de las caches para las diferentes plata-

[4] Véase Apéndice B.1
[5] Véase Apéndice B.2
[6] Las fachadas en los servicios API, son las clases encargadas de realizar llamadas a otros servicios externos.

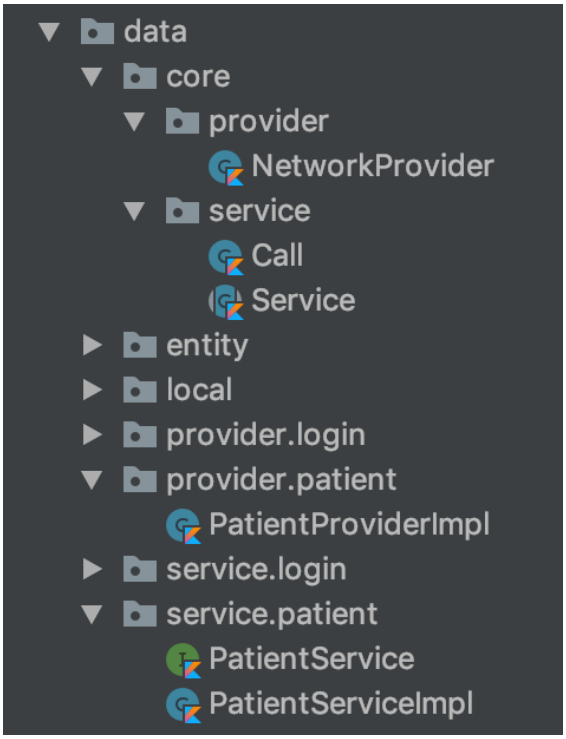

Figura 4.5: *Estructura de paquetes del modulo común de la capa de datos.*

formas, sirviendo como almacenamiento de datos en memoria para almacenar datos de corta duración, como por ejemplo el token requerido para realizar llamadas al API.

Por último, nos encontramos con las implementaciones de los ***providers***. Recordemos que la conexión entre la capa de datos y de dominio se hace a través de los distribuidores de datos y que estos son definidos en la capa superior, en este caso en la capa de dominio. También, como en el caso de los servicios, además de implementar la interfaz necesaria, los distribuidores deben extender el base (*NetworkProvider*[7]) que conecta a través de la función *request* las llamadas realizadas desde el servicio hacia el API. De esta forma, los distribuidores quedan con una implementación sencilla. Podemos encontrar un ejemplo de implementación de provider en el Apéndice B.4. En el cual vemos que tras extender la clase base *NetworkProvider*, y haciendo uso da la función definida por defecto, la obtención de datos a través de la API se reduce a una llamada empleando del servicio definido en el

[7] Véase Apéndice B.3

provider al cual se le pasan los parámetros necesarios para su correcta ejecución.

Domain

En la capa de dominio nos encontramos con dos elementos principales dentro del paquete ***core***, el ejecutor (Executor[8])), que define a través de una interfaz la co-rutina que debe ejecutar el dominio para obtener los datos de la capa de datos. Las co-rutinas son una alternativa sencilla para crear llamadas asíncronas de forma secuencial, de modo que en lugar de trabajar con callbacks y crear una lógica compleja que analice y mezcle los datos una vez obtenidos, se puede crear una secuencia de llamadas que cuando se requieran pueden suspender la ejecución y esperar hasta que el resultado esté disponible. El otro elemento principal, es la clase base para crear los casos de uso, *UseCase*[9]. Este recibe el ejecutor, que será diferente para cada plataforma, y realiza una invocación al distribuidor (*provider*) correspondiente para obtener los datos que serán propagados al presentador.

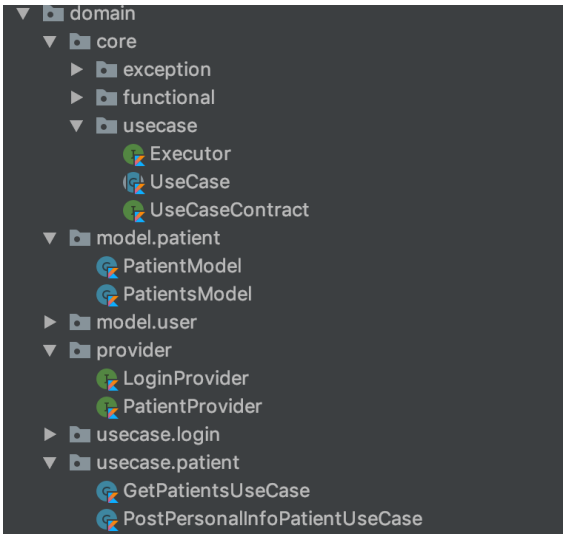

Figura 4.6: *Estructura de paquetes del módulo común de la capa de dominio.*

El resto de paquetes contiene los modelos y las implementaciones de los casos de uso. Para la creación de los casos de uso y partiendo del objeto base tan solo necesitamos definir qué

[8] Véase Apéndice B.5
[9] Véase Apéndice B.6

valores recibe y cómo se ejecuta. Como podemos ver en el Apéndice B.7, las funciones son creadas con la palabra clave *suspend* que indica que el método será ejecutado en asíncrono. Haciendo uso de la sintaxis simplificada que Kotlin ofrece, podemos ver que para declarar un caso de uso, vale con implementar la función *run* definida en el base e indicarle qué *provider* va a proporcionar los datos y qué modelo de datos será pasado para la correcta ejecución.

Presentation

Se trata de la capa que conecta con las interfaces de usuarios. Esta capa tan solo tiene que definir el contrato que las aplicaciones tendrán que implementar. De esta forma, el paquete de *core* contiene una base de contrato (*PresentationContract*[10]) para que al menos todos los presentadores tengan ciertos elementos en común, como son los métodos para inicializar y destruir los presentadores, así como una función para enlazar datos necesarios como el interactuador de las vistas. Además, existe una implementación básica de presentador (BasePresenter[11]) que contiene las funciones esenciales del contrato de forma que un presentador sea destruido por defecto, sea capaz de enlazar el interactuador y sepa analizar y manejar los errores tratándolo como un error genérico al menos.

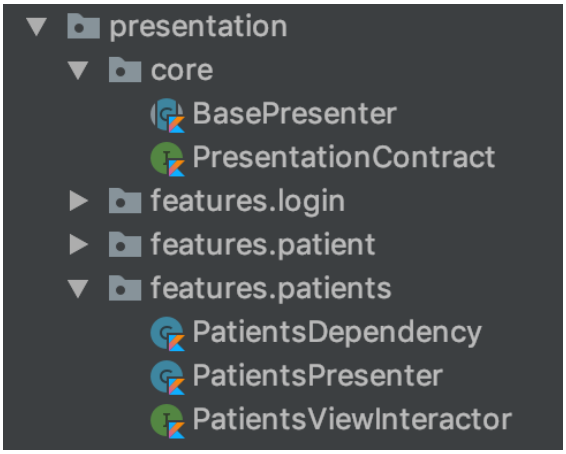

Figura 4.7: *Estructura de paquetes del modulo común de la capa de dominio.*

[10] Véase Apéndice B.8
[11] Véase Apéndice B.9

En los Apéndices B.10, B.11 y B.12 se puede ver un ejemplo del presentador de pacientes, el cual es definido mediante tres clases. La primera de ellas, define las dependencias que tiene con los casos de uso que empleará para ejecutar las diferentes co-rutinas que obtendrán los datos necesarios de la capa de datos. La interfaz del interactuador que define las acciones que el presentador puede realizar, así como las respuestas que proporciona para cada una de las acciones. En este caso podemos ver que el interactuador de la vista, contiene la función de *patients* y dos callbacks para que el presentador pueda realizar diferentes acciones en función de cual de los dos sea recibido. La interfaz es avisada mediante callbacks asíncronos, ya que en la capa de dominio aparece una sincronía que puede dejar a la interfaz esperando datos durante un tiempo bastante elevado. Además, el presentador ejecutará el callback de error en caso de no recibir una respuesta en un tiempo aceptable. Por último, tenemos el presentador, este es el encargado de interactuar con la capa de dominio para enviar y recibir los datos necesarios que permitan obtener los datos precisos para enviar a la interfaz e interactuar con el usuario.

4.3.2. Aplicación

Con la estructura elegida para el desarrollo de los proyectos, el desarrollo de las aplicaciones queda relegado a un segundo plano. Ahora tan solo es necesario crear las interfaces para cada una de las plataformas. Salvando las pequeñas diferencias que pueden tener una u otra plataforma, las vistas son similares de desarrollar. Como se adelantaba en el apartado anterior, cada plataforma tiene su propio ejecutor que es necesario crear a nivel de aplicación. Por ello vemos en la Figura 4.8 que tenemos un paquete que contiene el ejecutor propio (*core*) y otro que contiene las vistas (*ui*).

Cada pantalla de la aplicación corresponde a una vista. Recordemos que las vistas tienen una relación uno a uno con los presentadores, por ello, todas las vistas que requieran de la obtención de datos tendrán un presentador que será único para dicha vista. No todas las

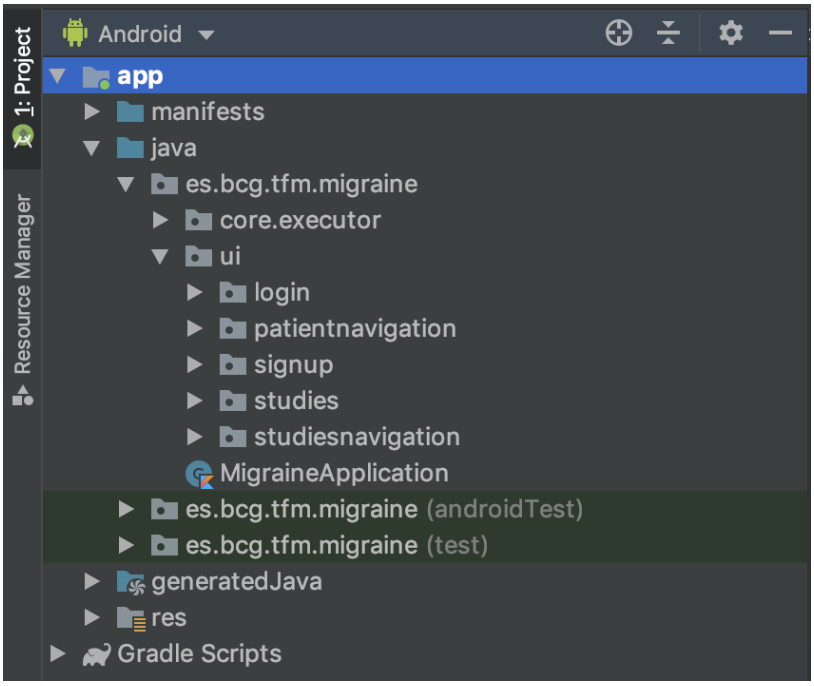

Figura 4.8: *Estructura de la aplicación Android.*

vistas necesitan datos, y por ello cabe la posibilidad de que no tengamos el mismo número de presentadores que de vistas. Además, se ha hecho uso de fragmentos para el desarrollo de la navegación entre las diferentes pantallas. Esto permite compartir un presentador entre varios fragmentos. Un presentador debe devolver los datos de una vista completa, y la vista es la composición de todos sus fragmentos. Si la vista reciclase sus fragmentos, cada uno de ellos debería tratarse como un caso particular y podría darse la posibilidad de crear un presentador para cada grupo de fragmentos que representasen una vista.

De esta manera las vistas han sido creadas de dos formas diferentes. Vistas simples (Figura 4.9), que no contienen fragmentos, o vistas navegables construidas con fragmentos. Las vistas simples se componen de dos elementos, la implementación del interactuador definido en la capa de presentación y la actividad que define los elementos de la vista, así como las acciones que realizan cada uno de los elementos, en caso de tener acción. Podemos encontrar un ejemplo de cada uno de los componentes de la vista simple en los Apéndices C.1 y C.2.

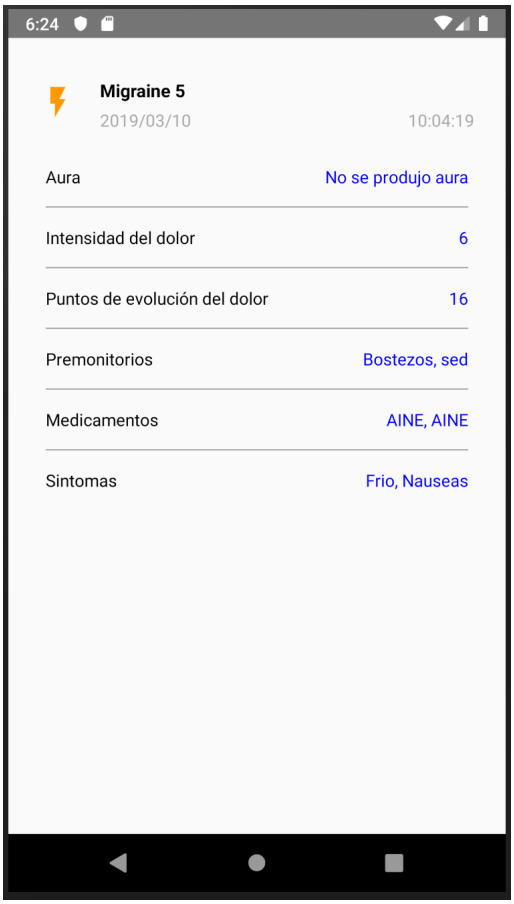

Figura 4.9: *Ejemplo de vista simple.*

En la actividad de la vista de login (C.2), podemos ver que se compone de una función *onCreate*, que se ocupa de cargar el recurso donde está definida la vista y de asociar cada elementos con su acción, si tiene. En este caso, podemos ver que aparecen tres elementos: (1) el botón de login, y los textos activos que permiten (2) crear nuevo usuario y (3) recuperar la contraseña respectivamente. Las dos últimas acciones desencadenan en una navegación de la aplicación, mientras que pulsar el botón de login realiza una llamada al módulo común que conecta con el microservicio de migraine y devuelve una respuesta a través de los callbacks. Podemos ver que existe la función *onDestroy* que implementa la destrucción de los elementos a través del interactuador de la vista definido en la capa de presentación.

Las vistas navegables han sido construidas con una actividad base donde se define el

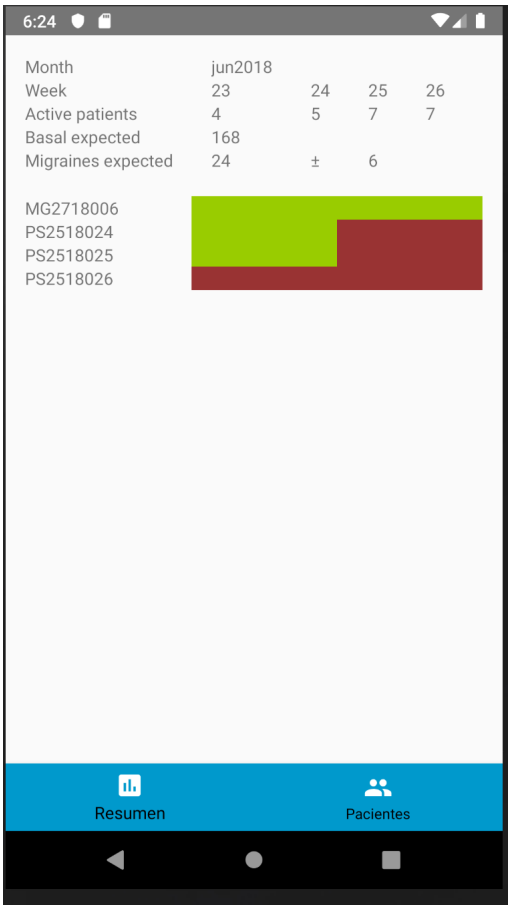

Figura 4.10: *Ejemplo de vista navegable.*

espacio que el fragmento va a ocupar, y una barra de navegación que permite el desplazamiento por los diferentes fragmentos. De esta forma no necesitamos cambiar de pantalla para navegar por la aplicación. En el caso de la aplicación de monitorización de los pacientes, esta navegación nos proporciona fluidez y una manera cómoda y sencilla de agrupar los datos de los estudios y los pacientes.

En la Figura 4.10 vemos que el espacio que ocupa la barra de navegación se encuentra en la parte inferior de la pantalla. Para navegar por los diferentes fragmentos, tan solo es necesario pulsar sobre los iconos que parecen en la barra de navegación y un fragmento nuevo será cargado en la vista navegable. En los Apéndices C.3, C.4 y C.5 encontramos un ejemplo de vista navegable, que coincide con la vista mostrada en la figura anterior. Vemos que para

crear este tipo de vistas se crea en primer lugar la clase contenedora de los fragmentos, en la cual se definen los fragmentos que contendrá a través del método *onCreate*. Además, para facilitar la navegación se ha creado una función que permite realizar el intercambio de fragmentos. Este método es llamado en cada uno de los elementos de la navegación con la posición correspondiente.

Cada uno de los fragmentos es luego creado como una vista simple con los mismos elementos: la implementación del interactuador de la vista y la clase que define la vista. Pero con la diferencia de que en lugar de extender una Actividad, estas vistan necesitan extender un fragmento. Al extender el fragmento los elementos son cargados en la función *onCreate*, pero implementan su funcionalidad en la función *onViewCreated*.

Capítulo 5

Caso de uso: estudio clínico de migraña

5.1. Introducción

El modelo arquitectónico presentado en los capítulos anteriores se ha puesta a prueba mediante la creación e implementación de una aplicación completa para llevar a cabo el seguimiento y visualización del estudio clínico realizado por los profesionales encargados del proyecto BrainGuard.

En este caso se requería de una herramienta orientada a personal sanitario con el conocimiento tecnológico de nivel usuario. La propuesta es desarrollar una aplicación para que los médicos y enfermeros participantes en el estudio puedan realizar un seguimiento de los pacientes de forma rápida y sencilla.

El proyecto BrainGuard trata de mejorar la vida de las personas enfermas de migraña, mediante la creación de un modelo de predicción que avise a los pacientes de que van a sufrir una crisis migrañosa con una anticipación suficiente para que puedan tomar las precauciones correspondientes.

5.1.1. Qué es la migraña

La migraña es una cefalea que afecta al 15 % de la población mundial, un 2 % sufre dicha cefalea durante 15 días al mes, es decir durante medio año. La enfermedad consiste en ataques de dolor de cabeza episódicos y recurrentes de intensidad media y grave. En sus variantes presenta síntomas neurológicos y/o gastrointestinales. Alrededor de un 25 % de los pacientes presentan síntomas neurológicos previos al dolor, esto es conocido como aura[?] .

Según la Organización Mundial de la Salud (OMS) la migraña es la sexta enfermedad más discapacitante en días de capacidad por años vividos. El 40 % de las personas que sufren migrañas tiene más de un ataque al mes, que dura 24 horas o más con una intensidad de dolor medio del 20 % y una intensidad de dolor grave del 80 %. Las migrañas tienen un pico de prevalencia que se sitúa entre los 25 y los 55 años, la etapa de las personas donde son más productivas.

Hasta un 4 % de los pacientes registrados que sufren migraña presentan una fase de migraña crónica, lo que consiste en más de 15 días de cefalea al mes, siendo cefalea migrañosa al menos 8 días al mes.

El dolor producido por esta enfermedad afecta a la calidad de la vida de las personas, obligando a las personas que lo sufren a cambiar sus hábitos de vida. Los pacientes muchas veces no pueden ir a trabajar, realizar sus tareas diarias o disfrutar de su tiempo libre ya que los dolores les obligan a acostarse.

5.2. Proyecto BrainGuard de gestión de migrañas

En 2014 nace BrainGuard motivado por un grupo de investigadores que quieren demostrar que el dolor de las crisis de la migraña se puede predecir paliando los dolores sufridos durante las crisis de tal forma que los pacientes no tengan que cambiar sus hábitos. Para

ello, BrainGuard se compone de profesionales formados en medicina, biotecnología e informática[50].

La finalidad del estudio es la creación de un modelo de predicción y alerta de la aparición de la crisis migrañosas para dotar a los pacientes de la posibilidad de actuar con antelación y evitar que el dolor aparezca. El estudio comenzó hace ya cinco años y ha permitido sacar las primeras conclusiones clínicas y técnicas sobre la predicción de migrañas.

5.2.1. Herramientas actualmente en uso en el proyecto BrainGuard

Para poder entrenar los algoritmos de predicción y crear modelos estadísticos con el mayor nivel de acierto posible, es necesario recabar datos de pacientes que sufran migrañas. Para ello, el proyecto posee varias herramientas que permiten obtener datos de las migrañas. Una vez recabada la información es necesario analizarlos para su posterior uso.

Las herramientas que emplean los pacientes durante el estudio para recopilar información se utilizan en dos puntos diferentes de la migraña. Por un lado, la aplicación del paciente recopila la información derivada de la crisis. Mientras que la pulsera se encarga de leer los síntomas previos producidos durante el aura para predecir las migrañas. Para realizar el seguimiento previo los pacientes son monitorizados mediante un dispositivo ambulatorio y no intrusivo.

Aplicación BrainGuard del paciente

La aplicación BrainGuard está dirigida a los pacientes que sufren de migraña y participan en el estudio[51]. Está orientada a que sean los propios pacientes los encargados de cargar los datos relacionados con la migraña, como pueden ser los síntomas y dolores que sufren durante una migraña. Para cargar los datos los pacientes disponen de dos pantallas fundamentalmente. En la Figura 5.1, vemos la primera pantalla a la que los pacientes son dirigidos para introducir los datos de una crisis. En ella los pacientes deben marcar cuándo

se inició el aura y cuándo se inició el dolor, así como cuándo terminan cada uno de ellos.

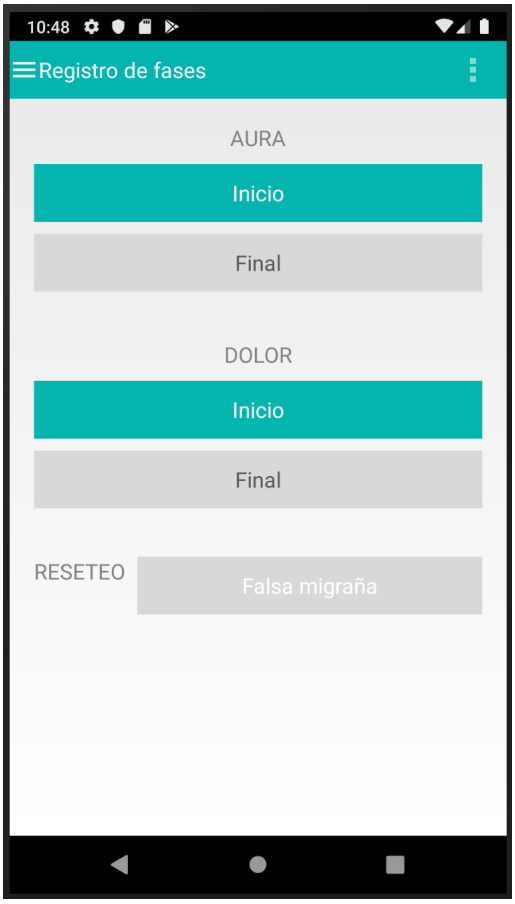

Figura 5.1: *Pantalla de la aplicación de los pacientes en la cual indicar cuando se inicia una crisis.*

Tras indicar esto, los pacientes deben navegar por la aplicación a través del menú lateral para llegar a la pantalla donde tienen que introducir los datos de la crisis. Para ello, disponen de un formulario, tal y como se muestra en la Figura 5.2. A través del formulario se indican los síntomas y dolores que han padecido durante la crisis y finalmente se envía el formulario.

Además, la aplicación de paciente permite ver un registro de la evolución del dolor durante las crisis, así como introducir los medicamentos que se han ido empleando. Antes de indicar el comienzo de una crisis se pueden introducir los síntomas prodrómicos[1]. Para ello, solo es necesario marcar el inicio del aura y el fin de la misma, habilitando la posibilidad de

[1] Síntomas que preceden a la aparición de una enfermedad

marcar los síntomas prodrómicos en la pantalla de donde se introducen los síntomas de la crisis.

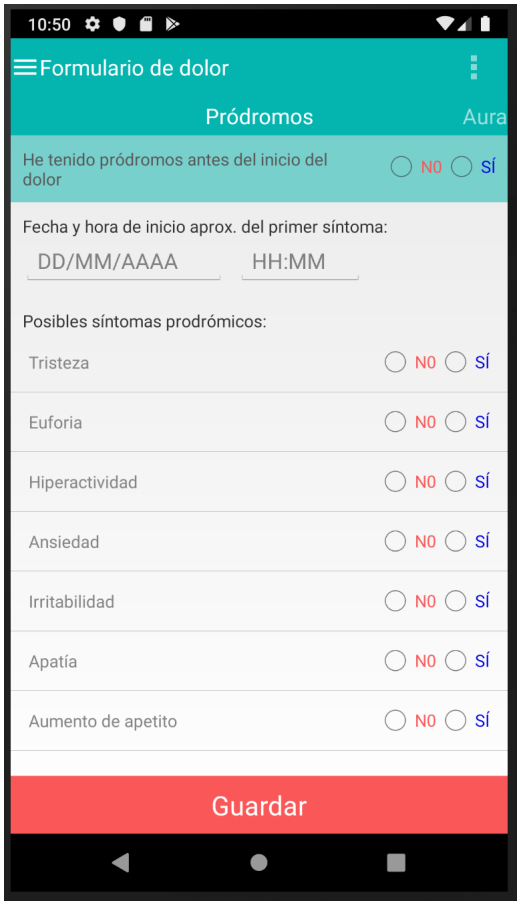

Figura 5.2: *Pantalla de la aplicación de los pacientes en la cual indicar los síntomas de una crisis.*

Pulsera E4 wristband

La pulsera empática E4 wristband es un dispositivo portátil que ofrece captación de datos en tiempo real y el software necesario para la visualización y análisis de los datos adquiridos[52]. Equipado con un sensor PPG que mide el pulso del volumen sanguíneo a partir del cual se calcula la variabilidad de la frecuencia cardiaca. Un sensor EDA, que mide los cambios constantes de ciertas propiedades de la piel. El EDA es un índice psicofisiológico sensible a los cambios que se integran con los estados emocionales y cognitivos de las

personas.

Incluye un acelerómetro de tres ejes, que captura la actividades basadas en el movimiento. Incluye un termómetro de infrarrojos que lee la temperatura de la piel periférica permitiéndole capturar los cambios que se producen. Además, incluye un reloj interno y un botón que permite marcar los eventos que se producen y vincularlos con las constantes psicofisiológicas leídas durante el tiempo que se marca el evento facilitando la monitorización de los datos.

También incorpora una memoria que permite grabar datos hasta 60 horas, lo cual lo hace perfecto para la captura de datos de pacientes que sufren migraña durante un largo tiempo. Es compatible con los sistemas operativos Windows y Mac y es necesario conectar la pulsera con los dispositivos vía USB para transferir los datos a un almacenamiento seguro a través de ***E4 Manager***. El software proporcionado permite además actualizar el firmware de la pulsera una vez que se conecta con un ordenador.

E4 connect es la plataforma de almacenamiento seguro en la nube, que almacena los datos encriptados. Permite además descargar los datos de la actividad electrotérmica (EDA), respuesta galvánica de la piel (GSR), pulso de volumen de sangre (BVP), aceleración, ritmo cardiaco (HR) y la temperatura del paciente que lleva puesta la pulsera. De esta forma los datos pueden ser transferidos a un tercero para su procesamiento. La pulsera posee la posibilidad de conectarse en tiempo real con ***E4 realtime*** para monitorizar un sesión en tiempo real empleando Bluetooth. *E4 realtime* es una aplicación con la opción de navegar con los datos en tiempo real y la posibilidad de mandar los datos a la plataforma *E4 connect* a través de internet sin la necesidad de tener que conectarse a un ordenador por USB.

5.2.2. Desarrollo del estudio

Dentro del proyecto de gestión de migrañas BrainGuard nos encontramos con varios estudios, los cuales poseen 5 fases principales que permiten el desarrollo y conclusión de los

estudios. La primera fase consiste en la **elección e inclusión de pacientes en el estudio**. Para ello es necesario que el personal clínico realice un pequeño seguimiento del paciente. De forma que a través de una serie de factores, como la recurrencia de las migrañas, elijan si los pacientes son buenos o no para su incorporación al estudio.

Tras seleccionar a los pacientes, el personal clínico con el apoyo del personal técnico imparten un **entrenamiento a los pacientes**, de apenas unos minutos, para que entiendan la aplicación del paciente (véase 5.2.1) y puedan cargar los datos de la crisis, durante la misma o una vez superada. El entrenamiento incluye una explicación del estudio haciéndoles entender a los pacientes la importancia de la aplicación, ya que es una de las principales fuentes de información.

Una vez explicado a los pacientes la aplicación es momento de darles de alta en el **inicio del estudio**, para ello, el personal técnico carga los datos del paciente en la base de datos y comunica mediante un correo cuáles son las credenciales del paciente. Estas credenciales pueden ser modificadas cuando el paciente lo desee a través de la aplicación. Esta fase concluye cuando los pacientes rellenan un formulario que aparece una sola vez con todos los datos sociodemográficos necesarios para la realización del estudio.

Después de dar de alta al paciente comienza el **seguimiento** del paciente. En esta fase, lo más importante es recabar información antes, durante y después de las crisis de los pacientes. Por ello, la participación de los pacientes es sumamente importante.

Por último, tenemos la fase de **fin del estudio**. En esta fase se saca al paciente del estudio. Para ello, primero se deja de hacer un seguimiento del paciente y segundo se marca al usuario en la base de datos como que terminó el estudio para evitar generar datos falsos de un paciente inactivo en el estudio.

5.2.3. La problemática de los datos

La gestión y mantenimiento de los datos supone el mayor problema para el correcto avance del estudio. En su estado actual se hace necesario cargar los datos en una hoja de cálculo de manera manual. Este cometido es acarreado por el personal técnico. La carga de datos en la hoja de cálculo requiere consultar la base de datos en la que se encuentran e interpretar los resultados. Esta acción se vuelve muy complicada para el personal técnico, que en muchas ocasiones necesita consultar los datos con algún miembro del personal clínico. La comunicación entre las diferentes personas del proyecto se realiza mediante correo electrónico en el mayor de los casos. Esto implica que la carga de datos suponga de mucho tiempo y gran esfuerzo para todo el personal implicado en el estudio.

Además, al cargar la información en una hoja de cálculo la navegación entre los datos de los pacientes y los estudios se vuelve muy enrevesada. Para cada estudio existe una hoja de cálculo que contiene el resumen del estudio. Además, para cada paciente existe una hoja de cálculo. Actualmente esto supone más de 30 hojas de cálculo que deben ser mantenidas en paralelo.

A los problemas particulares sobre los datos que presenta este estudio, se le suman los problemas genéricos de los datos clínicos. Primero no existe un estándar para medir los resultados de las enfermedades y permitir una interpretación general de los datos por cualquier persona. Además, en muchas ocasiones se requiere de personas con conocimientos específicos en cada campo para la interpretación de la información. Estas personas son conocidas como especialistas.

Al no existe un estándar cada centro sigue sus propias pautas a la hora de recabar y analizar los datos. Esto supone un problema para compartir información entre diferentes centros pertenecientes a un estudio. Para solucionar dicho problema, el personal técnico intenta homogeneizar los datos al máximo, pero no siempre es una tarea afable.

5.3. Adaptación de la arquitectura propuesta al seguimiento del paciente que sufre migraña

La aplicación desarrollada en este trabajo de fin de máster para poner a prueba la arquitectura afecta principalmente a las fases de inicio del estudio y seguimiento del pacientes. Estas son las fases más importantes para la obtención de datos. Actualmente no existe una herramienta que permita a los responsables en el estudio, personal clínico y técnico, realizar un seguimiento sencillo de los pacientes.

Para visualizar los datos, existe una persona técnica que es responsable de ejecutar peticiones a una base de datos y plasmar los datos en una hoja de cálculo que luego comparte con el resto de personas involucradas en el seguimiento de los pacientes. En cambio la aplicación propuesta en el presente trabajo permite acceder a los datos actualizados en cualquier momento.

5.3.1. Inicio del estudio

El inicio del estudio tiene dos etapas separadas. Durante la primera el personal técnico registra al paciente en la base de datos y envía un correo electrónico al paciente para que realice el login y complete sus datos. Al registrar los usuarios en la base de datos, estos comienza a aparecer en la aplicación desarrollada en este trabajo. Los usuarios que no han completado sus datos personales aparecen sombreados como se muestra en la Figura 5.3[2].

Esto permite a los responsables del seguimiento recordar a los pacientes que deben terminar su registro en la aplicación para poder sumarse al estudio. En caso de no registrarse durante un periodo de tiempo, los encargados pueden decidir dar de baja a los usuarios, dando por finalizado su participación en el estudio. La adherencia de los pacientes al estudio es escasa y en raras ocasiones un paciente que elige participar en el estudio termina antes

[2] Imagen pixelada para no incurrir en la filtración de datos sensibles.

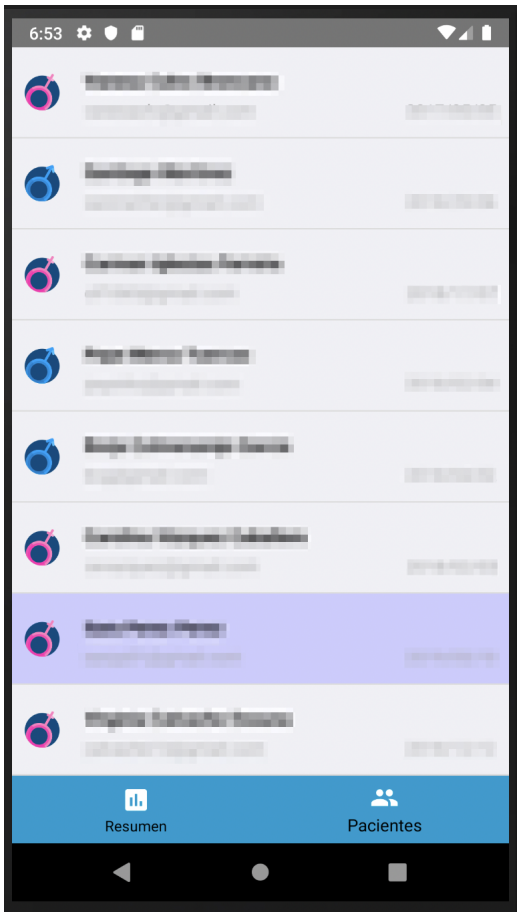

Figura 5.3: *Pantalla de pacientes. En ella podemos ver que los usuarios que no han completado su registro aparecen marcados en morado.*

de completar el inicio en el mismo. Con la evolución del estudio, la idea es liberar a los responsables de tener que realizar un seguimiento de qué pacientes no han completado el estudio. Además de crear un proceso por el cual los usuarios sean notificados un determinado número de veces. Tras no darse de alta en un tiempo razonable, los pacientes serán automáticamente dados de baja del estudio sin intervención de los responsables clínicos del seguimiento.

5.3.2. Seguimiento del paciente

Antes de que ocurra una migraña hay cambios en sus señales vitales, las cuales son monitorizadas de manera sencilla por las pulseras descritas anteriormente (véase 5.2.1). Cada pulsera proporciona entre otros el ritmo cardiaco, la temperatura corporal y el oxígeno en sangre, los cuales son factores clave en el diagnóstico y predicción de los síntomas previos a la aparición de una crisis[53].

Lo ideal en esta fase, es que los pacientes reciban un seguimiento fiel durante las crisis llevado a cabo por el personal clínico. El principal problema que existe con el seguimiento es que esta fase se ve "contaminada" por el continuo intercambio de correos con los pacientes con cuestiones puramente técnicas. Con la aplicación de seguimiento de pacientes se elimina este flujo de correos entre el personal clínico y los pacientes, pues la aplicación tiene un registro de cómo están los datos y los responsables del seguimiento son capaces de interpretar en tiempo real los datos de los pacientes, de modo que se puede enviar un correo con consejos para mejorar las cosas que crean que están haciendo mal, evitando así la comunicación de detalles técnicos.

Semanalmente se generan los correos electrónicos y se analizan por parte del personal técnico, para que posteriormente el personal clínico los revise y comunique a los pacientes el progreso del estudio. La generación de los correos electrónicos requiere de un análisis de los datos manualmente por parte del personal técnico que a menudo no tiene conocimientos clínicos, lo que resulta en un esfuerzo muy costoso y la realización de varios correos hasta quedar satisfechos. Gracias a los scripts de automatización de datos y a la aplicación de monitorización del estudio desarrollada en este trabajo, el personal clínico puede ver de un solo vistazo el estado de cada uno de los pacientes en el estudio. Esto facilita la tarea de seguimiento llevada a cabo por el personal clínico y libera de esta responsabilidad al personal técnico. El personal técnico por su parte en esta fase debe encargarse de revisar los datos verificando que los scripts funcionan correctamente y realizando las tareas de mantenimiento

necesarias para el correcto funcionamiento de la aplicación. Por ejemplo, el personal técnico debe asegurarse de que los datos insertados son correctos y no aparecen datos incoherentes o mezclados entre los pacientes.

La aplicación, además, ayuda a romper la dependencia que existe entre el equipo clínico y el equipo técnico que por temas laborales y personales a menudo se les hace difícil coincidir en un mismo horario, retrasando aún más los diagnósticos semanales a los pacientes. Con la aplicación de monitorización se facilita la tarea al personal clínico y permite que durante la fase más importante los pacientes reciban sus evaluaciones sin demoras en el proceso y directamente por una persona cualificada que es capaz, a través de sus propios medios, de obtener y analizar los datos sin dependencia de ninguna otra persona.

5.3.3. Navegación

La instanciación de la arquitectura propuesta en este proyecto de fin de máster para el caso específico de la migraña se compone de nueve pantallas y un cuadro de diálogo. El punto de entrada de la aplicación lleva a la pantalla número 1, la pantalla de login que, además de permitir acceder a los estudios del proyecto, permite al usuario realizar varias acciones. Pulsando sobre el texto activo (que responde a una pulsación) cuyo texto reza *Registrar* pasamos a la pantalla número 2. En esta podemos registrar un nuevo usuario rellenando el formulario que se muestra. También tenemos un texto activo (*¿Ya eres usuario? Login*) que nos devuelve a la pantalla número 1. En la pantalla número 1 existe otro texto activo, *¿Has olvidado tu contraseña?*, que muestra un cuadro de diálogo, tal y como podemos ver en la pantalla número 3. En este cuadro aparece un campo para introducir el correo electrónico del usuario que olvidó su contraseña y enviar un correo para recuperar la contraseña.

Desde la pantalla número 1, rellenando el formulario que incluye el correo electrónico y la contraseña, pulsando sobre el botón de *login*, realizamos la acción de login y accedemos al área de estudios. El área de estudios comienza en la pantalla número 4, donde se muestran

los estudios implicados en el proyecto. Pulsando sobre un estudio accedemos a la pantalla número 5, donde nos encontramos con el resumen del estudio. En el área de estudios, podemos encontrar que un estudio se compone de sus datos de resumen y los pacientes que pertenecen al estudio. Para navegar por estos datos desde la pantalla número 5 podemos navegar hacia la pantalla de pacientes, la pantalla número 6, empleando la barra inferior. Una vez en la pantalla número 6, aparecerá un scroll en caso de haber más pacientes en el estudio de los que la pantalla puede contener. En esta pantalla, vemos los pacientes y la fecha en la que se dio de alta en el estudio. Es en esta pantalla donde se puede ver la funcionalidad que mejora la fase del estudio de inicio. Aquí vemos que los pacientes que no han terminado el registro son sombreados en morado.

Desde la pantalla número 6 pulsando sobre un paciente, accedemos al área de paciente, donde nos encontramos 3 pantallas. En estas pantallas los datos en negro definen la información que representan, mientras que los datos azules representan los valores para cada campo. La primera pantalla que aparece es la pantalla número 7, que muestra un resumen de las migrañas del paciente durante el estudio. Además muestra un calendario que marca los días que el paciente ha sufrido migrañas. Para navegar por las pantallas 8 y 9, que pertenecen al área de pacientes, pulsamos sobre los iconos de la barra inferior. En la pantalla número 8 encontramos los datos sociodemográficos con los que el usuario se ha registrado en el estudio. Estos datos no varían, pero son de carácter consultivo para el personal clínico a la hora de realizar diagnósticos.

En la pantalla número 9 encontramos las migrañas que el paciente ha sufrido durante el estudio. Para diferenciar las migrañas tenemos un código de colores, verde, amarillo y rojo, que indica si una migraña es buena, falsa o mala respectivamente para su inclusión en los modelos de predicción. Esta pantalla al igual que la pantalla número 6 crea un scroll dinámico si las migrañas no caben en la pantalla.

Por último, para acceder al área de migrañas en la pantalla número 9 pulsamos sobre

una migraña y nos dirige a la pantalla número 10, que contiene el resumen de la migraña seleccionada.

Las pantallas de la 4 a la 10 mejoran la fase de seguimiento. Pues mientras que antes era necesario navegar en hojas de cálculo para revisar toda esta información, ahora tan solo es necesario marcar el estudio para acceder a la información del paciente que se desea. Además ya no es necesario realizar el mantenimiento de hojas de cálculo, y mucho menos de un mantenimiento en paralelo.

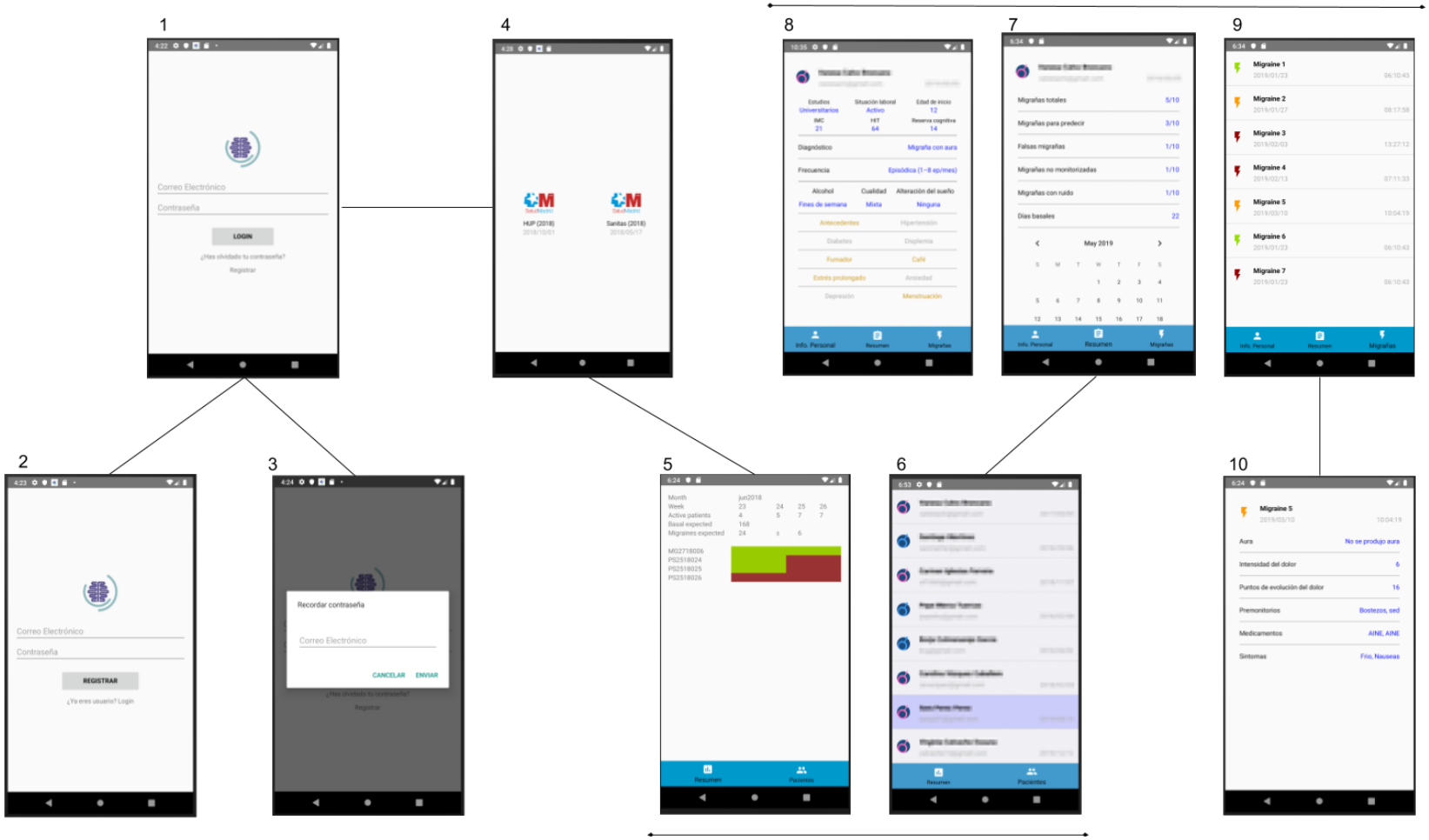

Figura 5.4: *Diagrama de navegación de la aplicación.*

Capítulo 6

Conclusiones y Trabajo Futuro

En el presente trabajo se ha elaborado el planteamiento, desarrollo e implementación de una arquitectura para elaborar aplicaciones médicas orientadas al seguimiento y monitorización de estudios clínicos. Para poner a prueba la arquitectura se ha aplicado a un estudio clínico en proceso realizado a través del proyecto BrainGuard. Esta arquitectura se adapta especialmente bien a aquellos estudios clínicos que requieren de la gestión de grandes cantidades de datos, pues simplifica considerablemente el trabajo a desarrollar tanto por el personal clínico como técnico.

Gracias a las nuevas tecnologías se ha conseguido implementar una herramienta en forma de aplicación, escalable, adaptable y de fácil mantenimiento. Esta aplicación se centra en el estudio de las personas que padecen de migraña y que son monitorizadas por diferentes herramientas para recabar información. Con la aplicación desarrollada se ha logrado mejorar algunas de las fases del estudio y se ha conseguido una aplicación que sirve como base para el desarrollo de futuras aplicaciones orientadas a seguimiento de pacientes en estudios clínicos complejos en su gestión de datos. Cabe destacar que para ello se han creado elementos genéricos que proporcionan interfaces y funcionalidades por defecto.

La gestión de los datos clínicos se convierte en la tarea más compleja dentro de los ensayos clínicos. En principio se trata de mejorar los datos hasta el punto de crear un estándar que

sirva para futuros estudios clínicos asociados a las migrañas. Pero recabar información desde diferentes herramientas y cargarla de manera automática aún no es posible.

Para tratar de alcanzar dicho objetivo se ha evolucionado la base de datos, lo cual no formaba parte del plan original, aplicando varias mejoras sobre la misma y sobre los scripts responsables de la carga de datos. Aún con la ayuda del personal técnico participante en el proyecto BrainGuard, la carga de la información sigue siendo un proceso semi-automático. Primero porque los datos recogidos por las diferentes herramientas no siempre son datos completos. Segundo porque es necesario analizar los fallos que provocan los scripts por si se pueden resolver manualmente. Y tercero por la complejidad de los datos clínicos en sí mismos.

El mundo sanitario y el mundo tecnológico aún tienen un largo camino que recorrer para unirse y disfrutar de las ventajas que proporciona la tecnología. Por ejemplo, la digitalización de los datos ayuda a los estudios conectando a los doctores con los pacientes en tiempo real. Pero para llevar a cabo dicha digitalización es necesario crear un estándar para los datos clínicos. Para ello es necesario que personal sanitario y técnico trabajen unidos y propongan las bases para llevar a cabo la tarea. Esta tarea requiere de un largo tiempo que esperamos se lleve a cabo poco a poco durante los próximos años.

En el caso de uso de la migraña la aplicación ha conseguido que los doctores hagan seguimiento del estudio clínico de los pacientes participantes en el estudio en tiempo real, agilizando los procesos y proporcionando dinamismo a los gestión de los datos. La herramienta ha permitido mejorar varias fases presentes en el estudio, como la fase de inicio y la fase de seguimiento.

En cuanto al trabajo futuro, enfocado en el caso de uso, la fase de entrenamiento se puede mejorar integrando vídeos, imágenes y demos para enseñar en la consulta a los pacientes que reciben el entrenamiento. La fase de seguimiento se puede mejorar con el desarrollo de una nueva funcionalidad que permita asignar permisos al equipo participante en el estudio.

Actualmente la aplicación permite crear nuevos usuarios para realizar el seguimiento de los pacientes, pero los usuarios pueden ver toda la información de todos los estudios. Con esta funcionalidad se podría asignar permisos para ver ciertos estudios.

En la fase de finalización, hoy en día una persona del equipo técnico se encarga de, mediante la ejecución de una actualización del registro de la base de datos, marcar al paciente como finalizado. Esta funcionalidad se puede trasladar a la aplicación de seguimiento. Por otro lado la fase de inicio se puede mejorar mediante una funcionalidad que permita cargar a los usuarios desde la aplicación de seguimiento. Con estas dos funcionalidades ya no seria necesario acceder a la base de datos de manera manual durante los estudios. Además, para mejorar la aplicación, se puede realizar una gestión de errores más fina, actualmente cada proceso solo tiene implementado un tipo de error.

Si nos referimos al trabajo futuro que se puede aplicar sobre la arquitectura, se puede mejorar el sistema añadiendo nuevos módulos. En este caso cualquier módulo puede aportar más funcionalidad. Pero si preguntamos a las personas que participaron en el caso de uso nos encontramos con la necesidad de crear un módulo que se encargue de la gestión de notificaciones, así como de otro módulo que se encargue de realizar informes y que sean exportados a formatos tales como PDF.

Siguiendo con el trabajo futuro encontramos la necesidad de extender la herramienta para adaptarla a más enfermedades. Gracias a la arquitectura diseñada esto se puede realizar como un módulo aparte y reutilizando los componentes dedicados a la gestión de usuarios. El código puede encontrarse en https://gitlab.com/colmenarejo-tfm.

Capítulo 7

Conclusions and Future Work

In the present document an approach, development and implementation of an architecture has been carried out to develop medical applications oriented to the monitoring of clinical studies. To test it, this architecture has been applied in an ongoing clinical study conducted by the BrainGuard group. Specifically, in clinical trials the follow-up phase requires a tool that fits the needs of the studies.

Thanks to new technologies it has been possible to implement a tool, in the form of application, scalable, adaptable and easy to maintain. This application focuses on the study of people who experiment migraine and who are monitored by different tools to collect information. With the application developed, it has been possible to improve some of the phases of the study and an application has been achieved that is the base for the development of future applications aimed at monitoring patients in clinical studies. It should be noted that for this purpose, generic elements have been created that provide interfaces and functionalities by default.

The management of clinical data becomes the most complex task within clinical trials. Initially, it is about improving the data to the point of creating a standard that will serve for future clinical studies associated with migraines. But collecting information from different tools and loading it automatically is not possible.

To try to reach this goal the database has evolved, which was not part of the original plan, applying several improvements on it and on the scripts responsible for data loading. Even with the help of the technical personnel involved in the BrainGuard project, information loading remains a semi-automatic process. First, because the data collected by the different tools is not always complete data. Second because it is necessary to analyze the failures caused by the scripts if they can be solved manually. And third because of the complexity of the clinical data themselves.

The world of health and the technological world still have a long way to go to join and enjoy the advantages that technology provides. For example, the digitization of the data helps the studies connecting the doctors with the patients in real time. But to carry out said digitalization it is necessary to create a standard for clinical data. For this it is necessary that the sanitary and technical personnel come together and propose the bases to carry out the task. This task requires a lot of time and we hope that it will be carried out little by little during the next years.

In the case of the use of migraine, the application has managed to connect the doctors with the studies in which they participate in real time, streamlining the processes and providing dynamism to the studies. The tool has allowed to improve several phases present in the study, such as the start phase and the follow-up phase.

With regard to future work, focused on the use case, the training phase can be improved by integrating videos, images and demonstrations to teach patients who receive training in the consultation. The follow-up phase can be improved with the development of a new functionality that allows assigning permissions to the team participating in the study. Currently, the application allows creating new users to track patients, but all users can see all the information of all studies. With this functionality you could assign permissions to view certain studies.

In the finalization phase, today a person from the technical team is responsible for,

when executing an update of the database record, to mark the patient as finished. This functionality can be moved to the tracking application. The start-up phase can be enhanced by a feature that allows users to load from the tracking application. With these two features it would no longer be necessary to access the database manually during the studies.

On the other hand, to improve the application, you can perform a more precise error management, currently each process has only one type of error implemented.

If we refer to the future work that can be applied to the architecture, the system can be improved by adding new modules. In this case, any module can provide more functionality. But if we ask the people who involved in the use case, we find the need to create a module that is responsible for the management of notifications, as well as another module that is responsible for preparing reports and that are exported in formats such as PDF.

Continuing with future work, we find the need to expand the tool to adapt it to more diseases. Thanks to the designed architecture, this can be done as a separate module and reusing the components dedicated to user management. The code can be found at https://gitlab.com/colmenarejo-tfm.

Bibliografía

[1] "¿Qué son los estudios clínicos?." https://www.cancer.gov/espanol/cancer/tratamiento/estudios-clinicos/que-son-estudios.

[2] "Las ventajas de la telefonía móvil en la actualidad." https://www.elconfidencialdigital.com/articulo/gadgets/ventajas-telefonia-movil-actualidad/20160914191750082823.html.

[3] "Versiones de Android y sus instalaciones activas." https://www.androidsis.com/versiones-de-android-y-sus-instalaciones-activas/.

[4] "¡3 Apps muy útiles para controlar el dolor de cabeza y la migraña!." http://blogdelalivio.com/dolor-de-cabeza/estas-las-apps-mas-utiles-controlar-dolor-cabeza-la-migrana/.

[5] "Migraine Buddy." https://migrainebuddy.com.

[6] "iMigrane." http://imigraine.io/.

[7] "iHeadache." http://www.iheadache.com/.

[8] "NeuroScores." https://www.estevefarma.com/medico/neuroscores.

[9] L. M. Jiménez, R. Puerto, and L. Payá, *Sistemas distribuidos: Arquitectura y aplicaciones*. Universidad Miguel Hernández de Elche, 2017.

[10] E. Marini, *El Modelo Cliente/Servidor*. 2012.

[11] "IBM Knowledge Center." https://www.ibm.com/support/knowledgecenter/en/SSAW57_8.5.5/com.ibm.websphere.nd.multiplatform.doc/ae/covr_3-tier.html.

[12] A. Cobo, *Diseño y programación de bases de datos*. Editorial Visión Libros, 2007.

[13] A. C. Fermín and M. S. Real, "Arquitectura de microservicios con RESTful," tech. rep., E.T.S.I. de Sistemas Informáticos (UPM), 2017.

[14] E. Amodeo, *Principios de Diseño de APIs REST*. Leanpub, 2013.

[15] "¿Qué son los microservicios?." https://www.redhat.com/es/topics/microservices.

[16] "La muerte de la locura de los microservicios en 2018." https://www.campusmvp.es/recursos/post/la-muerte-de-la-locura-de-los-microservicios-en-2018.aspx.

[17] "MySQL." https://www.mysql.com.

[18] S. Pachev, *Understanding MySQL Internals: Discovering and Improving a Great Database*. O'Reilly, 2007.

[19] "Qué es y para que sirve MySQL." http://culturacion.com/que-es-y-para-que-sirve-mysql/.

[20] "Hibernate." https://hibernate.org.

[21] A. M. Hart, "Hibernate in the Classroom," *J. Comput. Sci. Coll.*, vol. 20, pp. 98–100, Apr. 2005.

[22] "¿Qué es Java Hibernate?." https://blog.educacionit.com/2013/02/07/que-es-java-hibernate/.

[23] "Spring." https://spring.io/.

[24] "Programación J2EE. ¿Qué es Spring Framework?." http://www.programacionj2ee.com/que-es-spring-framework/.

[25] A. Saxena, N. Kaushik, and N. Kaushik, "Implementing and Analyzing Big Data Techniques with Spring FrameWork in Java & J2EE Based Application," in *Proceedings of the Second International Conference on Information and Communication Technology for Competitive Strategies*, ICTCS '16, (New York, NY, USA), pp. 5:1–5:6, ACM, 2016.

[26] "Welcome to Apache Maven." https://maven.apache.org/.

[27] B. Varanasi and S. Velida, *Introducing Maven*. Apress, 2014.

[28] J. Garzás, "Simple y rápido. Entiende qué es maven en menos de 10 minutos." https://www.javiergarzas.com/2014/06/maven-en-10-min.html.

[29] "Kotlin." https://kotlinlang.org/.

[30] "¿Debemos de elegir Kotlin para desarrollar Android? - gigigo." https://gigigo.com/2017/05/17/debemos-de-elegir-kotlin-para-desarrollar-android/.

[31] "Kotlin." https://kotlinlang.org/docs/reference/null-safety.html.

[32] A. Leiva, "Un primer vistazo a las co-rutinas de Kotlin en Android." https://www.genbeta.com/desarrollo/un-primer-vistazo-a-las-co-rutinas-de-kotlin-en-android.

[33] "Kotlin. Multiplatform Programming." https://kotlinlang.org/docs/reference/multiplatform.html.

[34] M. J. Martín, "SOLID: los 5 principios que te ayudarán a desarrollar software de calidad." https://profile.es/blog/principios-solid-desarrollo-software-calidad/principio_de_responsabilidad_unica.

[35] E. Chebanyuk and K. Markov, "An approach to class diagrams verification according to solid design principles," in *4th International Conference on Model-Driven Engineering and Software Development (MODELSWARD)*, pp. 435–441, 2016.

[36] "Setters & Getters." https://gl-epn-programacion-ii.blogspot.com/2010/03/setters-getters.html.

[37] J. S. Fernández, "Kotlin Multiplatform y el Principio de Inversión de Dependencias." http://xurxodev.com/kotlin-multiplatform-y-el-principio-de-inversion-de-dependencias/.

[38] G. E. Krasner and S. T. Pope, *A Cookbook for Using the Model-View-Controller User Inferface Paradigm in Smalltalk-80*. 1988.

[39] "Modelo Vista Presentador (MVP) en Android." http://www.develapps.com/es/noticias/modelo-vista-presentador-mvp-en-android.

[40] "(Code) Sharing is caring - An Introduction to Kotlin Multiplatform Projects." https://blog.novoda.com/introduction-to-kotlin-multiplatform/.

[41] "Kotlin. Creating a RESTful Web Service with Spring Boot." https://kotlinlang.org/docs/tutorials/spring-boot-restful.html.

[42] "Swagger y Swagger UI: ¿Qué es y por qué es imprescindible para tus APIs?." https://www.chakray.com/swagger-y-swagger-ui-por-que-es-imprescindible-para-tus-apis/.

[43] "JSON Web Token." https://es.wikipedia.org/wiki/JSON_Web_Token.

[44] "RFC 7519 - JSON Web Token (JWT)." https://tools.ietf.org/html/rfc7519.

[45] C. Álvarez Caules, "Spring boot Yaml y propiedades." https://www.arquitecturajava.com/spring-boot-yaml-y-propiedades/.

[46] "Maven. Introduction to the POM." https://maven.apache.org/guides/introduction/introduction-to-the-pom.html.

[47] "Maven Goals and Phases." https://www.baeldung.com/maven-goals-phases.

[48] "Maven. Introduction to the Build Lifecycle." https://maven.apache.org/guides/introduction/introduction-to-the-lifecycle.html.

[49] "Maven. Introduction to Build Profiles." https://maven.apache.org/guides/introduction/introduction-to-profiles.html.

[50] "BrainGuard." https://brainguard.life/?lang=es.

[51] A. B. Gago-Veiga, J. Pagán, K. Henares, P. Heredia, N. González-García, M.-I. D. Orbe, J. L. Ayala, M. Sobrado, and J. Vivancos, "To what extent are patients with migraine able to predict attacks?," *Journal of pain research*, vol. 11, 2018.

[52] "Real-time physiological signals. The E4 wristband." https://www.empatica.com/en-eu/research/e4/.

[53] J. Pagán, M. I. D. Orbe, A. Gago, M. Sobrado, J. L. Risco-Martín, J. V. Mora, J. M. Moya, and J. L. Ayala, "Robust and accurate modeling approaches for migraine per-patient prediction from ambulatory data," *Sensors*, vol. 15, no. 7, pp. 15419–15442, 2015.

Apéndice A

Códigos de ejemplo del microservicio *migraine*

Listing A.1: *MigraineApplication.kt*

```
package es.bcg.tfm.migraine

import org.springframework.boot.autoconfigure.SpringBootApplication
import org.springframework.boot.runApplication

@SpringBootApplication
class MigraineApplication

    fun main(args: Array<String>) {
        runApplication<MigraineApplication>(*args)
    }
```

Listing A.2: *PatientController.kt*

```
package es.bcg.tfm.migraine.controller

import es.bcg.tfm.migraine.controller.model.v1.patient
                                                .response.MinimalPatientsResponse
import es.bcg.tfm.migraine.controller.model.v1
                                                .patient.response.PatientResponse
import es.bcg.tfm.migraine.service.PatientService
import io.swagger.annotations.Api
import io.swagger.annotations.ApiOperation
import org.springframework.beans.factory.annotation.Autowired
```

```
import org.springframework.web.bind.annotation.*
import org.springframework.web.bind.annotation.RequestMethod.GET

@Api(tags = ["Patients"])
@RestController
class PatientController {

    @Autowired
    private lateinit var service : PatientService

    @ApiOperation(value = "Get all patients")
    @RequestMapping(value = ["/patients"], method = [GET],
        produces = ["application/json"])
    fun getPatients(@RequestParam(required = true) studyId : Int)
                                            : MinimalPatientsResponse {
        return service.getPatients(studyId)
    }

    @RequestMapping(value = ["/patients/{id}"], method = [GET])
    fun getPatient(
        @PathVariable(name = "id", required = true, value = "id")
            id : String)
                                            : PatientResponse {
        return service.getPatient(id)
    }

}
```

Listing A.3: *StudiesDto.kt*

```
package es.bcg.tfm.migraine.jpa.entity

import java.util.*
import javax.persistence.Column
import javax.persistence.Entity
import javax.persistence.Id

@Entity(name = "studies")
data class StudiesDto(

        @Id
        @Column(name = "study_id", nullable = false, length = 5)
        var id: Int,
```

```
        @Column(nullable = false, length = 255)
        var name: String,

        var startDate: Date? = null,

        var endDate: Date? = null,

        @Column(columnDefinition = "enum('case series','medical records',"
                + "'prevalence and incidence','cohort',"
                + "'cases and controls','natural experiments',"
                + "'clinicalessay','meta-analysis')")
        var type: String? = null,

        @Column(columnDefinition = "enum('unicentric','multicentric')")
        var size: String? = null,

        @Column(length = 2047)
        var description: String? = null,

        @Column(length = 2047)
        var comments: String? = null,

        @Column(length = 255)
        var fundingEntity: String? = null,

        @Column(length = 255)
        var fundingCode: String? = null
)
```

Listing A.4: *MigraineRepository.kt*

```
package es.bcg.tfm.migraine.jpa.repositoty

import es.bcg.tfm.migraine.jpa.entity.MigraineDto
import es.bcg.tfm.migraine.jpa.entity.MigraineId
import es.bcg.tfm.migraine.jpa.projection.MigraineProjection
import org.springframework.data.jpa.repository.JpaRepository
import org.springframework.data.jpa.repository.Query

interface MigraineRepository : JpaRepository<MigraineDto, MigraineId> {

    @Query("SELECT DISTINCT new es.bcg.tfm.migraine.jpa"
```

```
            + ".projection.MigraineProjection(e4.patientId, "
            + "sy.migraineId, e4.sessionId, e4.startTime, "
            + "e4.endTime, e4.tempQuality, e4.edaQuality, "
            + "e4.hrQuality, sy.symptomStartTime,"
            + " mi.painStartTime, mi.painEndTime, mi.falseEpisodeTime)"
            + " FROM e4_sessions AS e4, symptoms AS sy, migraines"
            + " AS mi WHERE e4.patientId = sy.patientId"
            + " AND sy.migraineId = mi.migraineId"
            + " AND e4.patientId = :patientId")
    fun findAllMigrainesByPatient(patientId: String)
                                        : List<MigraineProjection>

    fun findAllByPatientId(patientId: String) : List<MigraineDto>

}
```

Listing A.5: *StudiesServiceImpl.kt*

```
package es.bcg.tfm.migraine.service

import es.bcg.tfm.migraine.controller.model.v1.study
                                        .response.StudiesResponse
import es.bcg.tfm.migraine.controller.model.v1.study
                                        .response.StudyResponse
import es.bcg.tfm.migraine.jpa.entity.StudiesDto
import es.bcg.tfm.migraine.jpa.repositoty.StudiesRepository
import org.springframework.beans.factory.annotation.Autowired
import org.springframework.stereotype.Service
import java.text.SimpleDateFormat



@Service
class StudiesServiceImpl : StudiesService {

    private val studiesRepository: StudiesRepository

    var format = SimpleDateFormat("yyyy-MM-dd")

    private val mapper = { study: StudiesDto ->
        StudyResponse(study.id, study.name,
                      format.format(study.startDate)) }
```

```
    @Autowired
    constructor(studiesRepository: StudiesRepository) {
        this.studiesRepository = studiesRepository
    }

    override fun studies(): StudiesResponse {

        val studies = studiesRepository.findAll()

        val studiesResponse : MutableList<StudyResponse>
                                                = mutableListOf()
        for (study : StudiesDto in  studies){
            studiesResponse.add(mapper(study))
        }

        return StudiesResponse(studiesResponse)
    }

    override fun study(id: String): StudiesDto {
        return studiesRepository.findById(id).get()
    }

}
```

Listing A.6: *application.yml*

```
server:
  port: 7000

info:
  app:
    version: '@project.version@'
    name: '@project.name@'
    description: '@project.description@'
  development-info:
    creator: Borja Colmenarejo
    owner: Borja Colmenarejo
    language: Kotlin
    create-date: 28/12/2018
    last-modification: 05/02/2019

token:
  expiration-time: 3600000 # 1 hora
```

```
logging:
  level:
    es: DEBUG

spring:
  datasource:
    url: jdbc:mysql://localhost:3306/migraine
    username: mysql
    password: mysqlpass
  jpa:
    hibernate:
      ddl-auto: validate

flyway:
  password: ${spring.datasource.password}
  user: ${spring.datasource.username}
  url: ${spring.datasource.url}
```

Apéndice B

Códigos de ejemplo del módulo común de clientes

B.1. Data

Listing B.1: *Service.kt*

```
package es.bcg.tfm.migraine.data.core.service

import io.ktor.client.HttpClient
import io.ktor.client.call.receive
import io.ktor.client.request.HttpRequestBuilder
import io.ktor.client.request.header
import io.ktor.client.request.request
import io.ktor.client.response.HttpResponse
import io.ktor.http.HttpMethod
import io.ktor.http.HttpStatusCode
import io.ktor.http.takeFrom
import kotlinx.io.core.use

internal abstract class Service {

    companion object {
        private const val HEADER_TOKEN = "token"
        private const val HEADER_CONTENT_TYPE = "Content-Type"
        private const val HEADER_CONTENT_TYPE_JSON = "application/json"
        private const val HEADER_AUTHORIZATION = "Authorization"
    }
```

```
protected abstract fun getEndPoint(): String

protected abstract fun getClient(): HttpClient

protected suspend inline fun <reified T> requestLogin(
    _method: HttpMethod,
    path: String,
    _body: Any? = null
): Call<T> =
    try {
        getClient().request<HttpResponse> {
            header(HEADER_AUTHORIZATION, HEADER_AUTHORIZATION_BASIC)
            apiUrl(path)
            method = _method
            _body?.let {
                header(HEADER_CONTENT_TYPE, HEADER_CONTENT_TYPE_JSON)
                body = it
            }
        }.use {
            Call(it.status, it.receive())
        }
    } catch (e: Throwable) {
        Call(HttpStatusCode.BadRequest, null, e.toString())
    }

protected suspend inline fun <reified T> requestWithoutAuthentication(
    _method: HttpMethod,
    path: String,
    _body: Any? = null
): Call<T> =
    try {
        getClient().request<HttpResponse> {
            apiUrl(path)
            method = _method
            _body?.let {
                header(HEADER_CONTENT_TYPE, HEADER_CONTENT_TYPE_JSON)
                body = it
            }
        }.use {
            Call(it.status, it.receive())
        }
    } catch (e: Throwable) {
```

```
            Call(HttpStatusCode.BadRequest, null, e.toString())
        }

    protected suspend inline fun <reified T> request(
        _method: HttpMethod,
        path: String,
        token: String? = null,
        _body: Any? = null
    ): Call<T> =
        try {
            getClient().request<HttpResponse> {
                token?.let {
                    header(HEADER_TOKEN, token)
                }
                apiUrl(path)
                method = _method
                _body?.let {
                    header(HEADER_CONTENT_TYPE, HEADER_CONTENT_TYPE_JSON)
                    body = it
                }
            }.use {
                Call(it.status, it.receive())
            }
        } catch (e: Throwable) {
            Call(HttpStatusCode.BadRequest, null, e.toString())
        }

    protected fun HttpRequestBuilder.apiUrl(path: String) {
        url {
            takeFrom(getEndPoint())
            encodedPath = path
        }
    }
}
```

Listing B.2: *Call.kt*

```
package es.bcg.tfm.migraine.data.core.service

import io.ktor.http.HttpStatusCode

internal class Call<T>(
    val statusCode: HttpStatusCode,
```

```
    val body: T?,
    val error: String? = null
)
```

Listing B.3: *NetworkProvider.kt*

```
package es.bcg.tfm.migraine.data.core.provider


import es.bcg.tfm.migraine.data.core.service.Call
import es.bcg.tfm.migraine.domain.core.exception.FailureType
import es.bcg.tfm.migraine.domain.core.functional.Either
import io.ktor.http.HttpStatusCode

@Suppress("UNCHECKED_CAST")
internal open class NetworkProvider {

    inline fun <T, R> request(call: Call<T>, transform: (T) -> R)
                                    : Either<FailureType, R> {
        return try {
            when (call.statusCode) {
                HttpStatusCode.OK -> Either.Right(
                                            transform(call.body as T))
                HttpStatusCode.Unauthorized -> Either.Left(
                                            FailureType.TokenError)
                else -> Either.Left(FailureType.ServerError(
                                "${call.statusCode} -> ${call.error}"))
            }
        } catch (exception: Throwable) {
            Either.Left(FailureType.ServerError(exception.toString()))
        }
    }
}
```

Listing B.4: *PatientProviderImpl.kt*

```
package es.bcg.tfm.migraine.data.provider.patient

import es.bcg.tfm.migraine.data.core.provider.NetworkProvider
import es.bcg.tfm.migraine.data.entity.patients.request
                                                .PatientPersonalInfoRequest
import es.bcg.tfm.migraine.data.entity.patients.response.PatientResponse
import es.bcg.tfm.migraine.data.entity.patients.response.PatientsResponse
```

```
import es.bcg.tfm.migraine.data.local.LocalData
import es.bcg.tfm.migraine.data.service.patient.PatientService
import es.bcg.tfm.migraine.domain.core.exception.FailureType
import es.bcg.tfm.migraine.domain.core.functional.Either
import es.bcg.tfm.migraine.domain.model.patient.PatientModel
import es.bcg.tfm.migraine.domain.model.patient.PatientsModel
import es.bcg.tfm.migraine.domain.provider.PatientProvider

internal class PatientProviderImpl(
    private val patientService: PatientService,
    private val localData: LocalData
) : PatientProvider, NetworkProvider() {

    override suspend fun getPatients(): Either<FailureType, PatientsModel> =
        request(patientService.getPatients(localData.getToken())) {
            PatientsResponse.transform(it)
    }

    override suspend fun postPersonalInfo(uuid: String, personalInfoRequest:
        request(patientService.postPersonalInfo(localData.getToken(),
                                                uuid, personalInfoRequest)) {
            PatientResponse.transform(it)
        }

}
```

B.2. Domain

Listing B.5: *Executor.kt*

```
package es.bcg.tfm.migraine.domain.core.usecase

import kotlinx.coroutines.CoroutineDispatcher

interface Executor {

    val main: CoroutineDispatcher
}
```

Listing B.6: *UseCase.kt*

```
package es.bcg.tfm.migraine.domain.core.usecase
```

```
import es.bcg.tfm.migraine.domain.core.exception.FailureType
import es.bcg.tfm.migraine.domain.core.functional.Either
import kotlinx.coroutines.GlobalScope
import kotlinx.coroutines.Job
import kotlinx.coroutines.launch

internal abstract class UseCase<out T, in Params>(
                                    private val executor: Executor) :
    UseCaseContract where T : Any {

    private var job: Job? = null

    abstract suspend fun run(params: Params): Either<FailureType, T>

    operator fun invoke(onResult: (Either<FailureType, T>)
                                                -> Unit, params: Params) {
        job = GlobalScope.launch(context = executor.main) {
                                                    onResult(run(params)) }
    }

    override fun cancel() {
        job?.cancel()
    }

    object None
}
```

Listing B.7: *GetPatientsUseCase.kt*

```
package es.bcg.tfm.migraine.domain.usecase.login

import es.bcg.tfm.migraine.domain.core.exception.FailureType
import es.bcg.tfm.migraine.domain.core.functional.Either
import es.bcg.tfm.migraine.domain.core.usecase.Executor
import es.bcg.tfm.migraine.domain.core.usecase.UseCase
import es.bcg.tfm.migraine.domain.model.patient.PatientsModel
import es.bcg.tfm.migraine.domain.provider.PatientProvider

internal class GetPatientsUseCase (
    private val patientProvider: PatientProvider,
    executor: Executor
) : UseCase<PatientsModel, GetPatientsUseCase.Params>(executor) {
```

```
    override suspend fun run(params: Params): Either<FailureType,
                                                     PatientsModel> =
        patientProvider.getPatients()

    data class Params(val notVariable : String)
}
```

B.3. Presentation

Listing B.8: *PresentationContract.kt*

```
package es.bcg.tfm.migraine.presentation.core

import es.bcg.tfm.migraine.domain.core.exception.FailureType

interface PresentationContract {

    interface Presenter<in VI : ViewInteractor> {

        fun attachInteractor(viewInteractor: VI)

        fun initialize() {}

        fun destroy()
    }

    interface ViewInteractor {

        fun initialize()

        fun destroy()

        fun showError(error: FailureType)

        fun logOut()
    }
}
```

Listing B.9: *BasePresenter.kt*

```
package es.bcg.tfm.migraine.presentation.core
```

```
import es.bcg.tfm.migraine.domain.core.exception.FailureType
import es.bcg.tfm.migraine.domain.core.usecase.UseCaseContract

abstract class BasePresenter<T : PresentationContract.ViewInteractor>(
    private vararg val useCaseList: UseCaseContract
) : PresentationContract.Presenter<T> {

    protected var mViewInteractor: T? = null

    override fun attachInteractor(viewInteractor: T) {
        mViewInteractor = viewInteractor
        mViewInteractor?.initialize()
    }

    override fun destroy() {
        for (useCase in useCaseList) {
            useCase.cancel()
        }
        mViewInteractor = null
    }

    protected open fun handleError(error: FailureType) {
        mViewInteractor?.showError(error)
    }

    protected fun checkError(error: FailureType) {
        if (error is FailureType.TokenError) {
            mViewInteractor?.logOut()
            return
        }
    }
}
```

Listing B.10: *PatientsDependency.kt*

```
package es.bcg.tfm.migraine.presentation.features.patients

import es.bcg.tfm.migraine.data.local.LocalData
import es.bcg.tfm.migraine.data.provider.patient.PatientProviderImpl
import es.bcg.tfm.migraine.data.service.patient.PatientService
import es.bcg.tfm.migraine.data.service.patient.PatientServiceImpl
import es.bcg.tfm.migraine.domain.core.usecase.Executor
```

```
import es.bcg.tfm.migraine.domain.provider.PatientProvider
import es.bcg.tfm.migraine.domain.usecase.login.GetPatientsUseCase

class PatientsDependency (
    private val executor: Executor,
    private val localData: LocalData) {

    fun providePresenter(): PatientsPresenter =
        PatientsPresenter(
            provideGetPatientsUseCase(executor, localData))

    internal fun provideGetPatientsUseCase(executor: Executor,
                                           localData: LocalData) =
        GetPatientsUseCase(providePatientProvider(localData), executor)

    internal fun providePatientProvider(localData: LocalData)
                                            : PatientProvider =
        PatientProviderImpl(providePatientService(), localData)

    internal fun providePatientService(): PatientService
                                            = PatientServiceImpl()
}
```

Listing B.11: *PatientsPresenter.kt*

```
package es.bcg.tfm.migraine.presentation.features.patients

import es.bcg.tfm.migraine.domain.core.exception.FailureType
import es.bcg.tfm.migraine.domain.model.patient.PatientsModel
import es.bcg.tfm.migraine.domain.usecase.login.GetPatientsUseCase
import es.bcg.tfm.migraine.presentation.core.BasePresenter

class PatientsPresenter internal constructor(
    private val getPatientsUseCase: GetPatientsUseCase
) : BasePresenter<PatientsViewInteractor>() {

    fun patients() {
        getPatientsUseCase(
            { it.either(::handleError, ::handleLoginSuccess) },
            GetPatientsUseCase.Params("")
        )
    }
```

```
    private fun handleLoginSuccess(patientsModel: PatientsModel) {
        mViewInteractor?.showPatientsSuccess(patientsModel)
    }

    override fun handleError(error: FailureType) {
        super.handleError(error)
        when (error) {
            is FailureType.TokenError -> mViewInteractor?.logOut()
            else -> mViewInteractor?.showPatientsError(error)
        }
    }

}
```

Listing B.12: *PatientsViewInteractor.kt*

```
package es.bcg.tfm.migraine.presentation.features.patients

import es.bcg.tfm.migraine.domain.core.exception.FailureType
import es.bcg.tfm.migraine.domain.model.patient.PatientsModel
import es.bcg.tfm.migraine.presentation.core.PresentationContract

interface PatientsViewInteractor : PresentationContract.ViewInteractor {

    fun init()

    fun patients()

    fun showPatientsSuccess(patientsModel: PatientsModel)

    fun showPatientsError(error: FailureType)

}
```

Apéndice C

Códigos de ejemplo del la aplicación

Listing C.1: *LoginViewInteractorImpl.kt*

```kotlin
package es.bcg.tfm.migraine.ui.login

import es.bcg.tfm.migraine.domain.core.exception.FailureType
import es.bcg.tfm.migraine.domain.model.user.UserModel
import es.bcg.tfm.migraine.presentation.features.login.LoginPresenter
import es.bcg.tfm.migraine.presentation.features.login.LoginViewInteractor
import es.bcg.tfm.migraine.ui.login.viewmodel.UserViewModel

class LoginViewInteractorImpl(
    private val view: LoginContractView,
    private val presenter: LoginPresenter
) : LoginViewInteractor {

    override fun initialize() {
    }

    override fun init() {
        presenter.attachInteractor(this)
        presenter.initialize()
    }

    override fun loginUser(userId: String, password: String) {
        presenter.loginUser(userId, password)
    }

    override fun showLoginUserSuccess(userModel: UserModel) {
```

```
        view.showLoginOk(UserViewModel.transform(userModel))
    }

    override fun showLoginUserError(error: FailureType) {
        view.showLoginError()
    }

    override fun destroy() {
        presenter.destroy()
    }

    override fun showError(error: FailureType) {
    }

    override fun logOut() {
    }
}
```

Listing C.2: *LoginActivity.kt*

```
package es.bcg.tfm.migraine.ui.login

import android.content.Intent
import android.os.Bundle
import androidx.appcompat.app.AppCompatActivity
import es.bcg.tfm.migraine.R
import es.bcg.tfm.migraine.core.executor.AndroidExecutor
import es.bcg.tfm.migraine.presentation.features.login.LoginViewInteractor
import es.bcg.tfm.migraine.ui.login.viewmodel.UserViewModel
import es.bcg.tfm.migraine.ui.patientnavigation.PatientNavigationActivity
import es.bcg.tfm.migraine.ui.studiesnavigation.fragmet.patients
                                                    .PatientsFragment
import es.bcg.tfm.migraine.ui.patientnavigation.fragment.migraines
                                                    .MigrainesFragment
import es.bcg.tfm.migraine.ui.patientnavigation.fragment.patientsummary
                                                    .PatientSummaryFragment
import es.bcg.tfm.migraine.ui.signup.SignupActivity
import es.bcg.tfm.migraine.ui.studies.StudiesActivity
import kotlinx.android.synthetic.main.activity_login.*


/**
 * A login screen that offers login via email/password.
```

```
 */
class LoginActivity : AppCompatActivity(), LoginContractView {

    private var loginViewInteractor: LoginViewInteractor? = null

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_login)
        btn_login.setOnClickListener { login() }
        link_new_user.setOnClickListener { register() }
        link_forgot_password.setOnClickListener { rememberPassword() }

        val loginPresenter = LoginDependency(AndroidExecutor(),
        MigraineApplication.localData).providePresenter()
        loginViewInteractor = LoginViewInteractorImpl(this, loginPresenter)
        loginViewInteractor?.init()
    }

    override fun onDestroy() {
        super.onDestroy()
        loginViewInteractor?.destroy()
    }


    private fun register() {
        startActivity(Intent(baseContext, SignupActivity::class.java))
    }

    private fun rememberPassword() {
        ForgotPasswordDialog().show(supportFragmentManager,
                                            "ForgotPasswordDialog")
        startActivity(Intent(this, PatientNavigationActivity::class.java))
    }

    // Action login
    private fun login() {
        if (email.text != null && password.text != null) {
            loginViewInteractor?.loginUser("${email.text}",
                                                "${password.text}")
            btn_login.isEnabled = false
            link_new_user.isEnabled = false
            link_forgot_password.isEnabled = false
            email.isEnabled = false
```

```
            password.isEnabled = false
        }
    }

    override fun showLoginOk(userViewModel: UserViewModel) {
        startActivity(Intent(baseContext, StudiesActivity::class.java))
        enableScreen()
    }

    override fun showLoginError() {
        enableScreen()
    }

    private fun enableScreen(){
        btn_login.isEnabled = true
        link_new_user.isEnabled = true
        link_forgot_password.isEnabled = true
        email.isEnabled = true
        password.isEnabled = true
    }
}
```

Listing C.3: *StudiesNavigationActivity.kt*

```
package es.bcg.tfm.migraine.ui.studiesnavigation

import android.os.Bundle
import android.view.MenuItem
import androidx.appcompat.app.AppCompatActivity
import androidx.fragment.app.Fragment
import com.google.android.material.bottomnavigation.BottomNavigationView
import es.bcg.tfm.migraine.R
import es.bcg.tfm.migraine.ui.studiesnavigation.fragmet
                                                .patients.PatientsFragment
import es.bcg.tfm.migraine.ui.studiesnavigation.fragmet
                                                .summary.SummaryFragmet

class StudiesNavigationActivity : AppCompatActivity() {

    private var fragments = ArrayList<Fragment>(2)

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
```

```
        setContentView(es.bcg.tfm.migraine.R.layout.activity_studies_info)

        val bottomNavigation: BottomNavigationView =
        findViewById(es.bcg.tfm.migraine.R.id.navigationView)

        bottomNavigation.setOnNavigationItemSelectedListener(
            object : BottomNavigationView.OnNavigationItemSelectedListener {
                override fun onNavigationItemSelected(item: MenuItem)
                                                                : Boolean {
                    when (item.itemId) {
                        R.id.studies_navigation_summary -> {
                            switchFragment(0)
                            return true
                        }
                        R.id.studies_navigation_patients -> {
                            switchFragment(1)
                            return true
                        }
                    }
                    return false
                }
            })

        fragments.add(SummaryFragmet())
        fragments.add(PatientsFragment())

        switchFragment(0)
    }

    private fun switchFragment(pos: Int) {
        supportFragmentManager
            .beginTransaction()
            .replace(R.id.frame_patient_navigation, fragments[pos])
            .commit()
    }
}
```

Listing C.4: *PatientsFragment.kt*

```
package es.bcg.tfm.migraine.ui.studiesnavigation.fragmet.patients

import android.content.Context
import android.content.Intent
```

```
import android.os.Bundle
import android.view.LayoutInflater
import android.view.View
import android.view.ViewGroup
import android.widget.AdapterView
import android.widget.ArrayAdapter
import android.widget.ImageView
import android.widget.TextView
import androidx.fragment.app.Fragment
import es.bcg.tfm.migraine.MigraineApplication
import es.bcg.tfm.migraine.R
import es.bcg.tfm.migraine.core.executor.AndroidExecutor
import es.bcg.tfm.migraine.presentation.features.patients
                                                .PatientsDependency
import es.bcg.tfm.migraine.presentation.features.patients
                                                .PatientsViewInteractor
import es.bcg.tfm.migraine.ui.patientnavigation
                                                .PatientNavigationActivity
import es.bcg.tfm.migraine.ui.signup.SignupActivity
import es.bcg.tfm.migraine.ui.studiesnavigation.fragmet
                                .patients.viewmodel.MinimalPatientViewModel
import es.bcg.tfm.migraine.ui.studiesnavigation.fragmet
                                .patients.viewmodel.PatientsViewModel
import kotlinx.android.synthetic.main.activity_patients.*

class PatientsFragment : Fragment(), PatientsContractView {

    private var patientsViewInteractor: PatientsViewInteractor? = null

    override fun onCreateView(inflater: LayoutInflater,
                container: ViewGroup?, savedInstanceState: Bundle?): View? {
        return inflater.inflate(R.layout.activity_patients, container, false)
    }

    override fun onViewCreated(view: View, savedInstanceState: Bundle?) {
        super.onViewCreated(view, savedInstanceState)
        list_patients.adapter = PatientArrayAdapter(context!!, list)

        list_patients.onItemClickListener = AdapterView.OnItemClickListener()
        { parent, view, position, id ->
            startActivity(Intent(context,
                                    PatientNavigationActivity::class.java))
        }
```

```
        val patientsPresenter = PatientsDependency(AndroidExecutor(),
        MigraineApplication.localData).providePresenter()
        patientsViewInteractor = PatientsViewInteractorImpl(this,
                                                            patientsPresenter)
        patientsViewInteractor?.init()

        patientsViewInteractor?.patients()
    }

    override fun showPatientsOk(patientsViewModel: PatientsViewModel) {
        val list = patientsViewModel.patients
        list_patients.adapter = PatientArrayAdapter(context!!, list)
        list_patients.onItemClickListener = AdapterView.OnItemClickListener()
        { parent, view, position, id ->
            val intent = Intent(context, PatientDetailFragment::class.java)
            var bundle = Bundle()
            bundle.putParcelable("patient", list.get(position))
            intent.putExtra("bundlePatient",bundle)
            startActivity(intent)
        }
    }

    override fun showPatientsError() {
    }


    private inner class PatientArrayAdapter
                                : ArrayAdapter<MinimalPatientViewModel> {

        internal var items: List<MinimalPatientViewModel>

        constructor(context: Context,
                            objects: List<MinimalPatientViewModel>)
                : super(context, R.layout.item_patient, objects) {
            this.items = objects
        }

        private inner class ViewHolder(
            var sex: ImageView? = null,
            var name: TextView? = null,
            var email: TextView? = null,
```

```
            var phone : TextView? = null
        )

        override fun getView(position: Int,
                convertView: View?, parent: ViewGroup): View {
            var view: View?
            if (convertView == null) {
                view = layoutInflater.inflate(R.layout.item_patient, null)
                val viewHolder = ViewHolder()
                viewHolder.sex = view!!
                        .findViewById(R.id.patient_icon_sex) as ImageView
                viewHolder.name = view
                        .findViewById(R.id.patient_name) as TextView
                viewHolder.email = view
                        .findViewById(R.id.patient_date) as TextView
                viewHolder.phone = view
                        .findViewById(R.id.patient_init_date) as TextView
                view.tag = viewHolder
            } else {
                view = convertView
                (view.tag as ViewHolder)
            }
            val holder = view.tag as ViewHolder
            if (items[position].sex.equals("H")) {
                holder.sex!!.setImageResource(R.drawable.ic_male)
            } else {
                holder.sex!!.setImageResource(R.drawable.ic_female)
            }
            holder.email!!.text = items[position].email
            holder.phone!!.text = items[position].startDay
            holder.name!!.text = items[position].name
            return view
        }

    }
}
```

Listing C.5: *PatientsViewInteractorImpl.kt*

```
package es.bcg.tfm.migraine.ui.studiesnavigation.fragmet.patients

import es.bcg.tfm.migraine.domain.core.exception.FailureType
import es.bcg.tfm.migraine.domain.model.patient.PatientsModel
```

```
import es.bcg.tfm.migraine.presentation.features
                                    .patients.PatientsPresenter
import es.bcg.tfm.migraine.presentation.features
                                    .patients.PatientsViewInteractor
import es.bcg.tfm.migraine.ui.studiesnavigation
                        .fragmet.patients.viewmodel.PatientsViewModel

class PatientsViewInteractorImpl(
    private val view: PatientsContractView,
    private val presenter: PatientsPresenter
) : PatientsViewInteractor {

    override fun initialize() {
    }

    override fun init() {
        presenter.attachInteractor(this)
        presenter.initialize()
    }


    override fun destroy() {
        presenter.destroy()
    }

    override fun showError(error: FailureType) {}

    override fun logOut() {}

    override fun patients() {
        presenter.patients()
    }

    override fun showPatientsSuccess(patientsModel: PatientsModel) {
        view.showPatientsOk(PatientsViewModel.transform(patientsModel))
    }

    override fun showPatientsError(error: FailureType) {
        view.showPatientsError()
    }

}
```