Facultad de Ciencias

APPWISE: APLICACIÓN PARA LA GESTIÓN DE RIESGOS DE SEGURIDAD PERSONALES ASOCIADOS AL USO DE APLICACIONES MÓVILES

(AppWise: Application for the management of personal security risks associated with the use of mobile applications)

Trabajo de Fin de Grado
para acceder al

GRADO EN Ingeniería Informática

Autor: Alina Solonaru Botnari

Director: Carlos Blanco Bueno

Co-Director: Juan María Rivas Concepción

Julio – 2023

Resumen

El auge y la proliferación de las aplicaciones móviles han transformado nuestra forma de vida, ofreciéndonos acceso instantáneo a una amplia variedad de servicios, información y entretenimiento desde la palma de nuestra mano. Sin embargo, este crecimiento también ha dado lugar a preocupaciones cada vez mayores en términos de seguridad y privacidad. Con millones de usuarios de *smartphones* en todo el mundo, la necesidad de protegerse de amenazas como el *phising*, el *malware* y el ciberacoso se ha vuelto más apremiante que nunca.

Es en este contexto donde se plantea el desarrollo de *AppWise*, una aplicación diseñada para abordar estos desafíos y brindar a los usuarios una solución integral y efectiva para gestionar los riesgos asociados al uso de aplicaciones móviles. La visión de *AppWise* es proporcionar a los usuarios una experiencia segura y protegida en el entorno digital, ayudándoles a comprender y mitigar los riesgos potenciales que enfrentan al utilizar aplicaciones móviles.

AppWise se compone de dos elementos principales: un servicio REST y una aplicación móvil para la plataforma *Android*. El servicio REST, desarrollado utilizando *Spring* y *Java*, actúa como una fuente centralizada de datos, almacenando información detallada sobre catálogos de aplicaciones, riesgos y controles. Esto permite a la aplicación móvil acceder a la información actualizada y brindar a los usuarios una visión completa de los riesgos asociados a las aplicaciones que utilizan.

La aplicación móvil está implementada en *Android Studio*, proporciona una interfaz intuitiva y fácil de usar para que los usuarios puedan registrar y gestionar las aplicaciones que utilizan. Además de ofrecer información detallada sobre los riesgos asociados a cada aplicación, la aplicación también brinda recomendaciones personalizadas de controles para mitigar estos riesgos. Esto permite a los usuarios tomar medidas proactivas y adoptar prácticas de seguridad adecuadas mientras utilizan aplicaciones móviles.

Palabras clave: Aplicación móvil, Android, Servicio REST, Spring, Riesgos personales, Seguridad.

Abstract

The rise and proliferation of mobile applications have transformed our way of life, offering us instant access to a wide range of services, information, and entertainment at our fingertips. However, this growth has also led to growing concerns in terms of security and privacy. With millions of smartphone users worldwide, the need to protect oneself from threats such as phishing, malware, and cyberbullying has become more pressing than ever.

In this context, the development of *AppWise* is proposed, an application designed to address these challenges and provide users with a comprehensive and effective solution for managing the risks associated with mobile app usage. The vision of *AppWise* is to provide users with a secure and protected experience in the digital environment, helping them understand and mitigate the potential risks they face when using mobile applications.

AppWise consists of two main components: a REST service and a mobile application for the Android platform. The REST service, developed using *Spring* and *Java*, acts as a centralized data source, storing detailed information about app catalogs, risks, and controls. This allows the mobile application to access up-to-date information and provide users with a comprehensive view of the risks associated with the apps they use.

The mobile application, implemented in *Android Studio*, provides an intuitive and user-friendly interface for users to register and manage the apps they use. In addition to offering detailed information about the risks associated with each app, the application also provides personalized control recommendations to mitigate these risks. This enables users to take proactive measures and adopt proper security practices while using mobile applications.

Keywords: Mobile application, Android, REST service, Spring, Personal risks, Security.

Índice

Índice de ilustraciones

Índice de tablas

1. Introducción

La creciente adopción de dispositivos inteligentes y aplicaciones móviles ha transformado nuestra forma de vida, facilitándonos el acceso a información, servicios y entretenimiento en cualquier momento y lugar. Se estima que la cantidad de usuarios de smartphones en el mundo supere los 6.800 millones en 2023, lo que supone más del 80% de la población mundial, y ya en 2021 los usuarios de móviles descargaban más de 435.000 aplicaciones por minuto (Radicati Team, 2021).

Sin embargo, este rápido crecimiento en el uso de aplicaciones móviles también ha dado lugar a una serie de desafíos en términos de seguridad y privacidad. La exposición de datos personales, la proliferación de amenazas como el *phising* y el *malware*, y el aumento de los casos de *ciberacoso* y fraudes son solo algunos ejemplos de los riesgos a los que nos enfrentamos al utilizar aplicaciones móviles.

El número de ataques cibernéticos contra organizaciones aumentó en un 13%, con un alza notable en los ataques dirigidos a dispositivos móviles (Orange Business, 2021). *Promon*, una tecnología de protección de aplicaciones, probó recientemente 357 juegos móviles Android de alto rendimiento para realizar ingeniería inversa o manipular aplicaciones, el 81% (289) de las aplicaciones mostraron cero defensa contra estos ataques y no pudieron detectar un dispositivo comprometido (Vázquez, 2023).

Estas estadísticas son preocupantes y demuestran la necesidad urgente de abordar los riesgos asociados al uso de aplicaciones móviles. Los usuarios necesitan herramientas efectivas que les permitan comprender y mitigar estos riesgos de manera proactiva, protegiendo así su privacidad y seguridad en el mundo digital.

Es en este contexto donde se plantea el desarrollo de la aplicación AppWise. La motivación detrás de este trabajo surge de la necesidad de concienciar a los usuarios sobre los riesgos a los que se exponen y proporcionarles una solución práctica para reducir su nivel de riesgo personal. Con AppWise, los usuarios podrán crear un inventario de las aplicaciones que utilizan y obtener información detallada sobre los riesgos asociados a cada una de ellas, así como recomendaciones y controles personalizados para mitigar dichos riesgos.

Por lo tanto, este proyecto plantea como objetivo: brindar a los usuarios una herramienta completa y fácil de usar para gestionar los riesgos personales asociados al uso de aplicaciones móviles, mejorando así su seguridad y protección en el entorno digital.

Para la consecución de dicho objetivo se plantean los siguientes objetivos secundarios:

- Desarrollar un servicio REST que almacene la información de catálogo de aplicaciones, riesgos y controles. Este servicio actuará como una fuente centralizada de datos para la aplicación móvil.
- Crear una aplicación móvil para la plataforma Android que permita a los usuarios registrar y gestionar las aplicaciones que utilizan, así como proporcionar información detallada sobre los riesgos asociados a cada una de ellas.
- Extender la aplicación móvil para proporcionar información sobre los controles aplicados actualmente y qué controles poder aplicar para mitigar los riesgos identificados. Esto permitirá a los usuarios tomar medidas proactivas para proteger su privacidad y seguridad mientras utilizan aplicaciones móviles.

2. Materiales y Métodos utilizados

En este capítulo se describen varios aspectos relacionados con la realización del presente proyecto: la metodología de desarrollo utilizada, la planificación de tareas y las herramientas y tecnologías utilizadas.

2.1. Metodología

Se ha empleado la metodología iterativa incremental como enfoque principal para el desarrollo del proyecto, la cual se basa en la idea de dividir el trabajo en pequeñas iteraciones o ciclos, donde se planifican, diseñan, implementan y evalúan diferentes componentes o funcionalidades del producto en cada una de ellas. A medida que se avanza en el proyecto, se incorporan nuevas funcionalidades y se realizan mejoras iterativamente.

Esta metodología ha permitido una mayor flexibilidad y adaptabilidad durante el desarrollo del proyecto ya que se iban identificando posibles mejoras y ajustes necesarios en cada iteración, asegurando así la calidad y el cumplimiento de los objetivos establecidos.

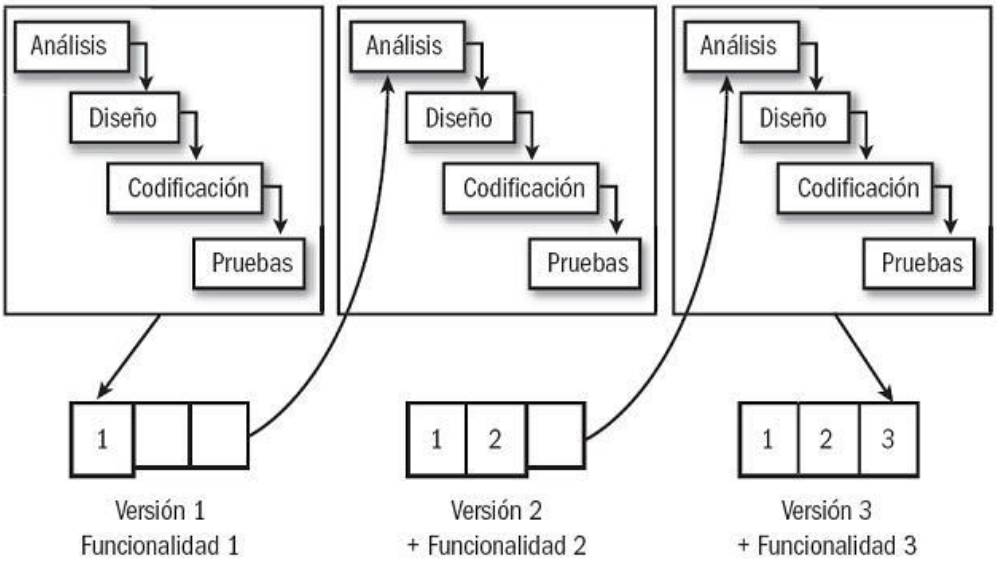


Ilustración 1. Representación gráfica del desarrollo iterativo incremental

2.2. Planificación

La realización del proyecto se divide en 3 etapas: planificación, desarrollo y redacción de la memoria.

2.2.1. Planificación inicial

Se siguieron varios pasos clave para definir el alcance y las funcionalidades de la aplicación. A continuación, se detalla el proceso seguido:

1. Reunión inicial con el tutor: Se llevó a cabo una reunión con el tutor para discutir las ideas iniciales y establecer el tema general de la aplicación. Durante esta reunión, se compartieron diferentes propuestas y se evaluaron sus viabilidades y relevancias.

2. Estudio de opciones y funcionalidades: Una vez se definió el tema de la aplicación, se procedió a investigar y explorar las diferentes opciones disponibles para implementar las funcionalidades deseadas. Dado que había diversas ideas, se realizó un análisis exhaustivo para seleccionar las funcionalidades más relevantes y factibles de implementar, teniendo en cuenta mis conocimientos y recursos disponibles.

3. Desarrollo de una *app* de prueba: Para adquirir experiencia y familiarizarse con los conceptos y tecnologías necesarios, se decidió desarrollar una aplicación de prueba preliminar. Esta *app* de prueba sirvió como un prototipo funcional para comprender mejor el enfoque que se debía tomar en el proyecto final. Durante esta fase, se implementaron elementos clave como la barra inferior de navegación y un buscador, lo que permitió aprender sobre el uso de *Fragments* y cómo implementar estas funcionalidades en la aplicación.

Ilustración 2. App de prueba

4. Investigación sobre servicios REST: Dado que se requería la implementación de un servicio REST para el intercambio de datos, se llevó a cabo una investigación exhaustiva sobre esta tecnología y cómo integrarla en aplicaciones *Android*. Se estudiaron conceptos como *endpoints*, métodos HTTP y formatos de datos para comprender su funcionamiento y asegurar una correcta implementación en el proyecto.

5. Reunión de replanificación con el tutor: Una vez se tuvieron claras las ideas y las funcionalidades principales de la aplicación, se programó otra reunión con

el tutor. Durante este encuentro, se replanteó el alcance del proyecto y se definieron de manera clara y concisa las tareas y funcionalidades que se incluirían en la aplicación final. También se establecieron los objetivos específicos a alcanzar y se estableció una hoja de ruta detallada para el desarrollo del proyecto.

2.2.2. Desarrollo

Como se ha mencionado en el apartado 2.1, se ha seguido una metodología iterativa, la cual ha permitido avanzar en el proyecto de manera planificada y eficiente. Cada iteración ha constado de un análisis de requisitos en el que se han establecido los objetivos que el software debe cumplir, una etapa de diseño en la que se ha planificado cómo satisfacer dichos objetivos, la implementación de las funcionalidades o requisitos planteados y las pruebas para verificar el correcto funcionamiento del software.

2.2.3. Memoria

Durante el proceso de desarrollo, se ha llevado a cabo la redacción de la memoria siguiendo un enfoque paralelo donde se ha ido documentando y describiendo el progreso del proyecto a medida que se avanzaba en las diferentes etapas de desarrollo. Esto ha permitido registrar con mayor precisión y detalle los pasos dados, incluyendo las decisiones tomadas, los desafíos encontrados y las soluciones implementadas. Además, esta metodología ha evitado la necesidad de realizar una redacción retrospectiva, asegurando una mayor coherencia y fluidez en la exposición de los resultados y conclusiones alcanzados.

2.2.4. Diagrama de Gantt

La planificación temporal de las principales etapas del proyecto se expone a continuación en un diagrama de Gantt.

Nombre de tarea	Duración	Comienzo	Fin
Planificación	**17 días**	**lun 06/02/23**	**mar 28/02/23**
Reunión inicial	1 día	lun 06/02/23	lun 06/02/23
Estudio de opciones	10 días	mar 07/02/23	lun 20/02/23
Realización de proyecto de prueba	15 días	mar 07/02/23	lun 27/02/23
Reunión de replanificación	1 día	mar 28/02/23	mar 28/02/23
Desarrollo	**76 días**	**mié 01/03/23**	**mié 14/06/23**
Iteración 1. Servicio REST	14 días	mié 01/03/23	lun 20/03/23
Iteración 2. Persistencia y conexión con el servicio	7 días	mar 21/03/23	mié 29/03/23
Iteración 3. Pestaña APPS	7 días	jue 30/03/23	vie 07/04/23
Iteración 4. Vista detalle App	5 días	lun 10/04/23	vie 14/04/23
Iteración 5. Pestaña RIESGOS	5 días	lun 17/04/23	vie 21/04/23
Iteración 6. Vista detalle Riesgo	5 días	lun 24/04/23	vie 28/04/23
Iteración 7. Pestaña CONTROLES	5 días	lun 01/05/23	vie 05/05/23
Iteración 8. Vista detalle Control	5 días	lun 08/05/23	vie 12/05/23
Iteración 9. Pestaña PERFIL	10 días	lun 15/05/23	vie 26/05/23
Iteración 10. Buscador de aplicaciones	7 días	lun 29/05/23	mar 06/06/23
Iteración 11. Retoques finales	5 días	mié 07/06/23	mar 13/06/23
Reunión de control	1 día	mié 14/06/23	mié 14/06/23
Memoria	90 días	mié 01/03/23	mar 04/07/23

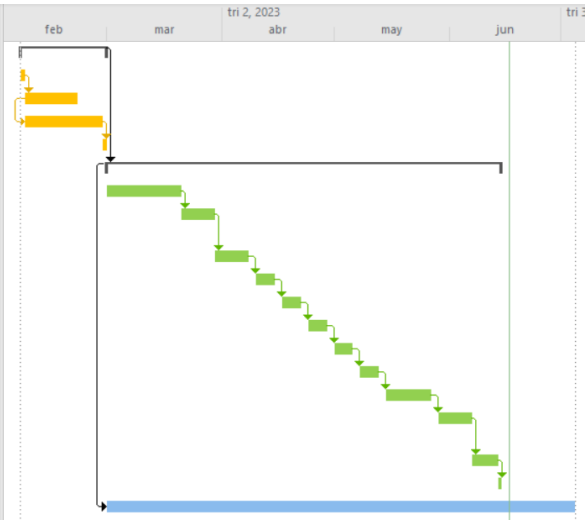


Ilustración 3. Diagrama de Gantt (etapas del proyecto)

2.3. *Tecnologías y Herramientas*

En el desarrollo del proyecto, se evaluaron varias opciones de herramientas y tecnologías con el objetivo de seleccionar las más adecuadas para cada etapa. A continuación, se detallan cada una de las herramientas y tecnologías escogidas.

Tanto para el desarrollo del servicio REST como la aplicación *Android* se ha usado *Git* como sistema de control de versiones, junto con la herramienta *Sourcetree* para una interfaz visual más amigable. *Git* es ampliamente utilizado en la industria del desarrollo de software y proporciona un mecanismo eficiente para el seguimiento de cambios, la colaboración y el control de versiones del código fuente. Se proporciona el enlace al repositorio de *GitHub* donde se encuentra el código completo del proyecto.

2.3.1. Servicio REST

Para el desarrollo del servicio REST se han utilizado las siguientes herramientas:

1. Entorno de programación: *Eclipse* es un entorno de desarrollo integrado (IDE) ampliamente utilizado en la industria del desarrollo software. Ofrece un conjunto de herramientas poderosas y una interfaz intuitiva que facilita la creación, depuración y despliegue de aplicaciones. Se ha elegido debido a la familiaridad y experiencia previa con la plataforma, lo que ha permitido aprovechar las habilidades existentes y agilizar el proceso de desarrollo del servicio REST.

2. Lenguaje de programación: *Java* es un lenguaje de programación robusto y versátil que se utiliza ampliamente en el desarrollo de aplicaciones empresariales. Tiene una sintaxis clara y una amplia gama de bibliotecas y *frameworks*, ofrece un entorno confiable para construir aplicaciones escalables y seguras. Al elegirlo se ha podido aprovechar su amplia adopción en la comunidad de desarrollo, su gran cantidad de recursos de aprendizaje y su compatibilidad con numerosas tecnologías, incluido el desarrollo de servicios REST. Además, la familiaridad con *Java* también ha permitido aprovechar la experiencia previa y acelerar el desarrollo del servicio.

3. *Framework* para el servicio REST: *Spring* es un *framework* de desarrollo de aplicaciones *Java*, se basa en el principio de inversión de control (IoC) y la programación orientada a aspectos (AOP), lo que permite una mayor modularidad y flexibilidad en el desarrollo de aplicaciones. Ofrece diversos módulos que abarcan desde la gestión de dependencias, la integración con bases de datos y servicios web, hasta la implementación de seguridad y pruebas unitarias.

   Inicialmente, se consideró el uso de JAX-RS (*Java API for RESTful Web Services*) junto con JAXB (*Java Architecture for XML* Binding) y su despliegue en servidores como *Tomcat*, *TomcatEE* y *Glassfish*, basado en la experiencia previa con estas herramientas. Sin embargo, se encontraron dificultades para que llegara a funcionar, lo que llevó a la elección de *Spring*, específicamente *Spring Boot*, que proporciona una integración sin problemas con el servidor incorporado y las tecnologías de serialización, lo que ha permitido un desarrollo más eficiente y sin obstáculos en el servicio REST.

4. Base de datos: *MySQL* es un sistema de gestión de bases de datos relacional, es ampliamente utilizado, cuenta con una buena documentación y es

compatible con *Java*, lo que facilita la integración con el servicio REST desarrollado.

5. Gestión del ORM: JPA (*Java Persistence API*) se ha utilizado para gestionar el mapeo objeto-relacional (ORM). Como una especificación de *Java*, JPA proporciona un conjunto de interfaces y anotaciones que permiten mapear objetos *Java* a tablas en una base de datos relacional. Al utilizar JPA, se ha simplificado la interacción con la base de datos, ya que se han definido entidades persistentes y se han establecido relaciones entre ellas utilizando las anotaciones convenientes. JPA se encarga de realizar las operaciones de persistencia, como insertar, actualizar y eliminar registros, de manera transparente, abstrayendo gran parte de la complejidad asociada con el acceso a la base de datos. Esto ha permitido un manejo eficiente y estructurado de los datos en el servicio REST.

6. Ayuda a la gestión del proyecto: *Maven* es una poderosa herramienta de gestión de proyectos y construcción de software que simplifica la gestión de dependencias, la compilación, el empaquetado y la implementación de aplicaciones.

2.3.2. Aplicación móvil

Para la implementación de la aplicación, las herramientas más destacables han sido las siguientes:

1. Entorno de desarrollo: *Android Studio* es un entorno de desarrollo integrado (IDE) específicamente diseñado para el desarrollo de aplicaciones *Android*. Proporciona un conjunto completo de herramientas y funcionalidades que facilitan la creación, depuración y prueba de aplicaciones móviles. La elección de *Android Studio* se basa en la experiencia previa y familiaridad con la plataforma, ya que además ofrece una interfaz intuitiva, un amplio soporte para *Android SDK* y una integración fluida con otras herramientas y servicios de desarrollo de *Android*.

2. Persistencia de datos: *GreenDAO* es un ORM (*Object-Realtional Mapping*) ligero y eficiente para *Android*, utilizado para el manejo de la capa de persistencia en la aplicación. Proporciona una forma sencilla de trabajar con la base de datos y realizar operaciones CRUD (Crear, Leer, Actualizar, Eliminar) en los objetos de la aplicación. *GreenDAO* se eligió principalmente por su capacidad para establecer relaciones entre objetos de una manera sencilla, a diferencia de otras bibliotecas como *Room*, que se consideró inicialmente, pero se descartó debido a la complejidad que implicaba para manejar las relaciones entre objetos.

3. Acceso a la API REST: *Retrofit* es una biblioteca de Android que simplifica la comunicación con servicios web RESTful. Se ha utilizado para realizar las solicitudes HTTP al servicio REST desarrollado. *Retrofit* ofrece una interfaz intuitiva y potente para definir las llamadas a la API y manejar las respuestas de manera eficiente.

4. Deserialización de los datos: *Gson* es una biblioteca de *Java* desarrollada por *Google* que se utiliza para convertir objetos *Java* en su representación JSON y viceversa. En el contexto de la aplicación, *Gson* se utiliza para deserializar los

datos recibidos de la API REST en formato JSON y convertirlos en objetos *Java* que se pueden utilizar en la lógica de la aplicación. La elección de *Gson* se basa en su simplicidad de uso, rendimiento y amplia compatibilidad con los estándares de JSON.

3. Análisis de Requisitos

En esta sección se presenta una visión general de la aplicación, brindando una descripción completa de los requisitos funcionales y no funcionales que la misma debe cumplir.

3.1. Visión general de AppWise

AppWise será una aplicación móvil diseñada para gestionar los riesgos asociados al uso de aplicaciones móviles. Proporcionará a los usuarios un inventario de las aplicaciones que utilizan y ofrecerá información detallada sobre los riesgos de seguridad asociados a cada una de ellas. La aplicación hará uso de un servicio REST que deberá proporcionar un catálogo de aplicaciones con su respectiva categoría, riesgos y controles.

Para ponerse en contexto, un riesgo es una probabilidad de que, ante una situación determinada (en este caso, la descarga de una aplicación concreta), una amenaza informática se convierta en un evento real que resulte en un daño, que en este caso afecta personalmente al usuario. Por ejemplo, la exposición de datos personales, el spam, el ciberacoso o la publicidad engañosa.

Por otro lado, un control se refiere a una posible acción que puede tomar el usuario para evitar o reducir los daños ocasionados por dichos riesgos. Por ejemplo, la descarga de aplicaciones sólo de fuentes fiables, la configuración de contraseñas seguras, la instalación de antivirus o la realización de copias de seguridad periódicas.

Dicho esto, la interfaz de la aplicación se divide en diferentes secciones para facilitar la navegación y la visualización de la información. En la sección de “Aplicaciones”, los usuarios podrán explorar el catálogo de aplicaciones, realizar búsquedas y acceder a detalles específicos de cada aplicación, incluyendo sus riesgos asociados. Además, los usuarios podrán agregar o eliminar aplicaciones de su perfil personal.

La sección de “Riesgos” permitirá a los usuarios acceder a una lista de riesgos de seguridad y obtener información detallada sobre cada uno de ellos, así como los controles recomendados para mitigarlos. Los usuarios también podrán explorar la sección de “Controles”, donde se proporcionará una lista de controles de seguridad y su relación con los riesgos correspondientes que mitigan.

La aplicación permitirá la interacción entre secciones, lo que significa que los usuarios podrán acceder rápidamente por ejemplo a la vista de detalle de un riesgo desde la vista de detalle de las aplicaciones. Además, se incluirá una sección personal donde los usuarios podrán ver su nivel de riesgo calculado en función de las aplicaciones asociadas y los controles aplicados. Aquí, también podrán visualizar y administrar las aplicaciones y controles agregados a su perfil personal.

En resumen, *AppWise* pretenderá brindar a los usuarios una visión general de las aplicaciones que utilizan, los riesgos asociados a ellas y los controles recomendados para mitigar esos riesgos, tal y como se muestra en la Ilustración 4. Esta aplicación

proporcionará una herramienta práctica y útil para gestionar los riesgos de seguridad en el entorno móvil de manera efectiva y personalizada.

Ilustración 4. Mockups de *AppWise*

3.2. Requisitos funcionales

A continuación se presentan los requisitos funcionales considerados, organizados por aquellos relacionados con el servicio REST y con la aplicación móvil.

3.2.1. Servicio REST

ID	Descripción
RF1	El servicio debe proporcionar un punto de acceso para obtener todas las aplicaciones móviles con la posibilidad de filtrar por categoría.
RF2	El servicio debe proporcionar un punto de acceso para obtener una aplicación concreta identificándola por su nombre.
RF3	El servicio debe proporcionar un punto de acceso para obtener todos los riesgos.
RF4	El servicio debe proporcionar un punto de acceso para obtener un riesgo concreto identificándolo por su id.
RF5	El servicio debe proporcionar un punto de acceso para obtener todos los controles.
RF6	El servicio debe proporcionar un punto de acceso para obtener un control concreto identificándolo por su id.
RF7	El servicio debe proporcionar un punto de acceso para obtener todas las categorías de aplicaciones móviles.
RF8	El servicio debe proporcionar el id, nombre y descripción de cada control.
RF9	El servicio debe proporcionar el id, nombre, descripción y lista de controles de cada riesgo.
RF10	El servicio debe proporcionar el nombre y la lista de riesgos de cada categoría de aplicaciones móviles.
RF11	El servicio debe proporcionar el nombre, *url* del icono y categoría de cada aplicación.

Tabla 1. Requisitos funcionales del Servicio REST

3.2.2. Aplicación móvil

ID	Descripción
RF1	La aplicación debe mostrar el catálogo completo de aplicaciones móviles ordenadas por categoría.
RF2	La aplicación debe proporcionar información detallada sobre una aplicación seleccionada (icono, nombre, categoría y riesgos asociados).
RF3	La aplicación debe permitir a los usuarios buscar aplicaciones dentro del catálogo utilizando su nombre o su categoría.
RF4	La aplicación debe permitir a los usuarios poder agregar o eliminar aplicaciones a su perfil personal.
RF5	La aplicación debe permitir a los usuarios acceder a la lista completa de los riesgos de seguridad asociados a aplicaciones móviles.
RF6	La aplicación debe proporcionar información detallada sobre un riesgo seleccionado (nombre, descripción y controles recomendados para mitigarlo).
RF7	La aplicación debe permitir a los usuarios acceder a la lista completa de controles de seguridad asociados a riesgos en aplicaciones móviles.
RF8	La aplicación debe proporcionar información detallada sobre un control seleccionado (nombre, descripción y riesgos que mitiga).
RF9	La aplicación debe permitir a los usuarios poder agregar o eliminar controles a su perfil personal.
RF10	La aplicación debe permitir a los usuarios acceder a la lista de aplicaciones agregadas a su perfil personal.
RF11	La aplicación debe permitir a los usuarios acceder a la lista de controles agregados a su perfil personal.
RF12	La aplicación debe mostrar el nivel de riesgo personal del usuario en base a las aplicaciones y controles agregados.
RF13	La aplicación debe proporcionar un indicador visual junto a las aplicaciones o controles agregados en las listas correspondientes.

Tabla 2. Requisitos funcionales de la Aplicación móvil

3.3. Requisitos no funcionales

A continuación, se presentan los requisitos no funcionales o atributos de calidad que ha de satisfacer la aplicación desarrollada.

ID	Tipo	Descripción
RNF1	Portabilidad	La aplicación debe ser compatible y ejecutarse en diferentes versiones de *Android*, desde *Android 8.1 Oreo* hasta las versiones más recientes, garantizando así su portabilidad en diferentes dispositivos *Android*.
RNF2	Rendimiento	La aplicación debe tener un tiempo de carga rápida (inferior a 3 segundos), y una respuesta fluida (inferior a 400 milisegundos), asegurando una experiencia de usuario ágil y sin retrasos perceptibles.
RNF3	Usabilidad	La aplicación debe adaptarse al modo oscuro de manera consistente, asegurando que todos los elementos visuales sean legibles y de forma que se reduzca la fatiga visual en entornos con poca iluminación.
RNF4	Escalabilidad	La arquitectura de la aplicación debe ser escalable, permitiendo gestionar un aumento en la cantidad de datos sin comprometer el rendimiento y la funcionalidad de la aplicación.
RNF5	Usabilidad	La interfaz de usuario de la aplicación debe ser intuitiva,

		fácil de usar y seguir las directrices de diseño de Android, brindando una experiencia de usuario agradable y facilitando la interacción con la aplicación.

Tabla 3. Requisitos no funcionales

4. Diseño e implementación del Servicio REST

En esta sección se aborda el diseño y la implementación del servicio REST, explicando brevemente la arquitectura utilizada y profundizando en cada una de las capas.

4.1. Arquitectura

Para desarrollar el servicio REST se ha seguido la arquitectura en capas típica de las aplicaciones *Spring Boot*, que sigue un enfoque de separación de responsabilidades y promueve la modularidad y la reutilización del código. A continuación, se describen las capas principales de esta arquitectura:

- *Controller Layer*: Esta capa se encarga de recibir las solicitudes HTTP y gestionar la interacción con los clientes. Los controladores son responsables de manejar las peticiones, llamar a los servicios correspondientes y devolver las respuestas adecuadas al cliente. Aquí se definen las rutas y los puntos de entrada de la aplicación.

- *Service Layer*: La capa de servicios contiene la lógica de negocio de la aplicación. Los servicios se encargan de procesar y transformar los datos, aplicar reglas de negocio y coordinar las operaciones entre las capas superiores e inferiores. Aquí se implementan las funcionalidades principales de la aplicación.

- *Repository Layer*: Esta capa se encarga de interactuar con la capa de persistencia, que puede ser una base de datos relacional, una base de datos NoSQL u otro sistema de almacenamiento. Los repositorios proporcionan métodos para realizar operaciones CRUD (Crear, Leer, Actualizar, Eliminar) en las entidades de la aplicación y abstraen los detalles específicos del almacenamiento de datos.

- Entidades: Representan los objetos del dominio de la aplicación y encapsulan los datos y comportamientos asociados. Estas entidades mapean directamente al sistema de almacenamiento.

- Base de Datos: Almacena los datos de la aplicación de manera persistente.

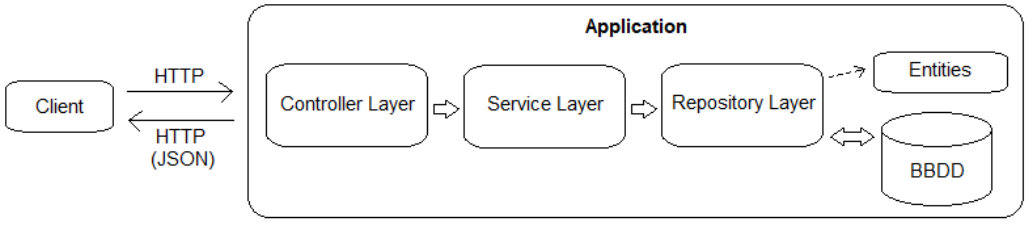


Ilustración 5. Esquema arquitectura Spring Boot

Concretamente, el diagrama de componentes que representa la arquitectura del servicio es el que se muestra a continuación, donde pueden observarse de manera organizada los diferentes componentes distribuidos en cada capa, los cuales se conectan mediante interfaces proporcionadas por la capa inferior hacia la superior.

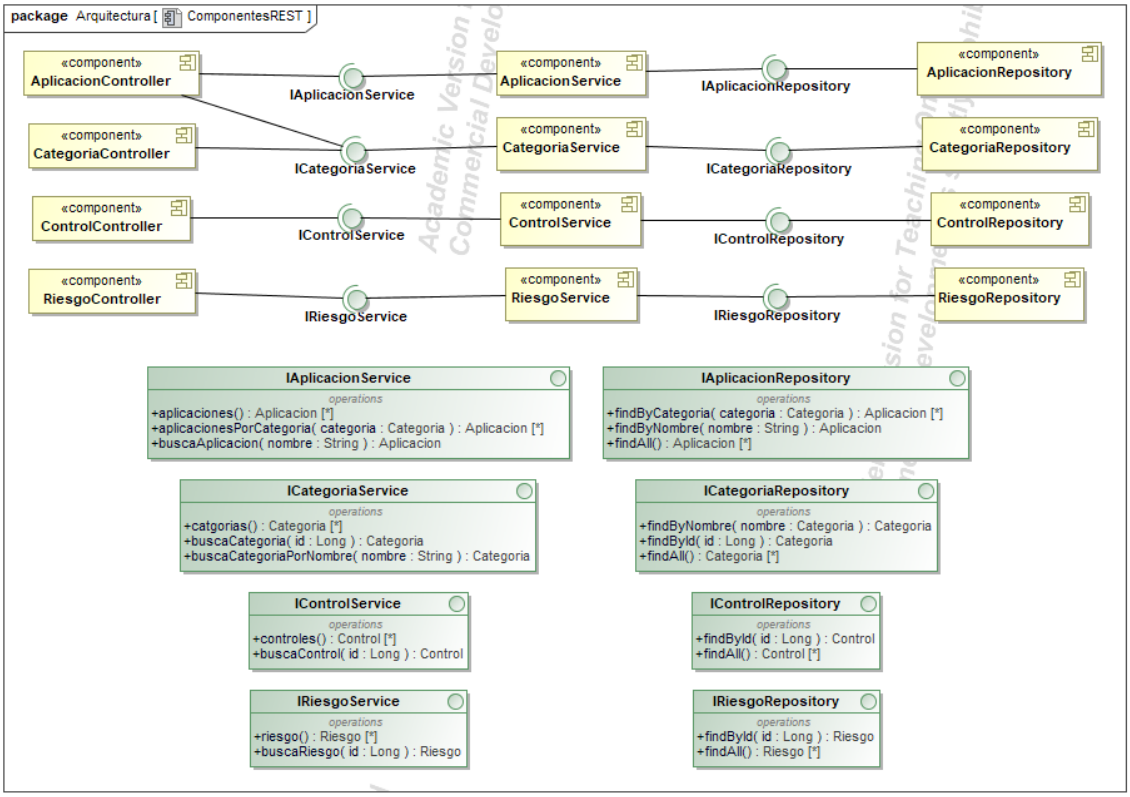


Ilustración 6. Diagrama de componentes del Servicio REST

También es importante destacar que, en el desarrollo de este proyecto, el despliegue del servicio se ha llevado a cabo en un entorno local.

4.2. Controller Layer

El primer paso en el desarrollo del servicio es definir los recursos que lo componen, las plantillas URI para cada recurso, los métodos de la interfaz uniforme soportados por cada recurso, incluyendo los códigos de respuesta HTTP que cada uno retorna, así como las representaciones de cada recurso en el lado del cliente y el servidor.

A continuación, se presenta el diseño del servicio, por cada elemento de información (*app*, riesgo, control, categoría) hay un recurso mediante el que se puede obtener la lista completa y otro con el que se puede obtener el elemento individual accediendo a él mediante su id o nombre. En el caso de las aplicaciones, también se puede filtrar la lista por categoría.

Recurso	URI	Query Params	Interfaz uniforme	Códigos HTTP
Aplicaciones móviles	/apps	categoría	GET	200 (OK)
Aplicación móvil	/apps/{nombre}		GET	200 (OK) 404 (Not Found)
Riesgos de seguridad	/riesgos		GET	200 (OK)
Riesgo de seguridad	/riesgos/{id}		GET	200 (OK) 404 (Not Found)
Controles	/controles		GET	200 (OK)

Control	/controles/{id}		GET	200 (OK) 404 (Not Found)
Categorías	/categorías		GET	200 (OK)

Tabla 4. Diseño Servicio REST

Para definir la estructura y contenido de los datos intercambiados entre el cliente y el servidor, se establecen las representaciones de cada recurso. Esto permite tener claridad sobre la información que se incluirá en las solicitudes y las respuestas. En este caso particular, dado que todos los métodos son de tipo GET, se enfoca en proporcionar la representación del lado del servidor.

En la siguiente tabla se presenta un ejemplo de la posible respuesta JSON del servidor al realizar una solicitud GET al recurso "Aplicaciones móviles", utilizando la URI */apps*. La respuesta incluye una lista con una sola aplicación llamada "*CleanMaster*" y los datos asociados a dicha aplicación.

Representación JSON


```
[
    {
        "nombre": "CleanMaster",
        "icono": "https://i.imgur.com/bbtqAl0.jpg",
        "categoria": {
            "nombre": "Herramientas de Sistema",
            "riesgos": [
                {
                    "id": 1,
                    "nombre": "Exposición de datos personales",
                    "descripcion": "Este riesgo se refiere ...",
                    "controles": [
                        {
                            "id": 1,
                            "nombre": "Fuentes confiables",
                            "descripcion": "Descarga aplicaciones ..."
                        },
                        {
                            "id": 3,
                            "nombre": "Política de privacidad",
                            "descripcion": "Lee la política de ..."
                        },
                        {
                            "id": 4,
                            "nombre": "Contraseña segura",
                            "descripcion": "Configura una contraseña ..."
                        }
                    ]
                }
            ]
        }
    }
]
```


Tabla 5. Ejemplo representación JSON del recurso Aplicaciones móviles (GET)

La implementación de las clases *Controller* se realiza mediante clases POJO (*Plain Old Java Object*) anotadas con *@RestController* y otras anotaciones específicas que definen los puntos de acceso diseñados.

A continuación, se muestra un ejemplo de la implementación de la clase *AplicationController*, la cual se encarga de gestionar los recursos “Aplicaciones móviles” y “Aplicación móvil”. La URI base *(/apps*) se especifica al comienzo de la clase y cada método se anota con la extensión correspondiente, por ejemplo, para obtener una aplicación concreta se necesita añadir su nombre. También se puede observar cómo se hace uso de los servicios *AplicacionService* y *CategoriaService* para implementar la lógica requerida según la URI y los parámetros proporcionados.

```java
@RestController
@RequestMapping("/apps")
public class AplicacionController {

    @Autowired
    private AplicacionService aplicacionService;

    @Autowired
    private CategoriaService categoriaService;

    @GetMapping
    public List<Aplicacion> getAplicaciones(@RequestParam(value="categoria", required = false) String categoria) {

        List<Aplicacion> apps = new LinkedList<Aplicacion>();

        if (categoria != null) {
            apps = aplicacionService.aplicacionesPorCategoria(categoriaService.buscaCategoriaPorNombre(categoria));
        } else {
            apps = aplicacionService.aplicaciones();
        }

        return apps;
    }

    @GetMapping("/{nombre}")
    public ResponseEntity<Aplicacion> getAplicacion(@PathVariable String nombre) {
        ResponseEntity<Aplicacion> result;
        Aplicacion app = aplicacionService.buscaAplicacion(nombre);

        if (app == null) {
            result = ResponseEntity.notFound().build();
        } else {
            result = ResponseEntity.ok(app);
        }
        return result;
    }

}
```

Ilustración 7. Clase AplicacionController

4.3. Service Layer

La capa de servicio se encarga de implementar la lógica de negocio de la aplicación. Si la lógica es simple, como en el caso de operaciones CRUD (Crear, Leer, Actualizar, Eliminar), se puede optar por omitir esta capa. Sin embargo, en este proyecto se ha decidido mantener la capa de servicio por motivos de consistencia y para seguir las mejores prácticas de diseño de software, pues mantenerla permite agregar lógica adicional en el futuro sin afectar directamente a los controladores ni a la capa de persistencia. Así, incluso en casos donde la lógica de negocio inicial sea simple, mantener la capa de servicio brinda una estructura clara y coherente en la arquitectura de la aplicación, lo cual facilita su comprensión y futuras modificaciones o mejoras.

La implementación de las clases *Service* se basa también en clases POJO anotadas con *@Service*.

NOTA: Aunque en el Diagrama de componentes del Servicio REST se muestran explícitamente las interfaces entre la capa *controller* y la capa *service*, así como entre la capa *service* y la capa *repository*, en la implementación se accede directamente a los componentes inyectándolos mediante la anotación *@Autowired*.

A continuación, se muestra la implementación de la clase *AplicacionService*, encargada de implementar las principales funcionalidades relacionadas con las aplicaciones móviles, basadas en simples operaciones de lectura haciendo uso del repositorio de aplicaciones *AplicacionRepository*.

```java
@Service
public class AplicacionService {

    @Autowired
    private AplicacionRepository repository;

    public List<Aplicacion> aplicaciones() {
        return repository.findAll();
    }

    public List<Aplicacion> aplicacionesPorCategoria(Categoria categoria) {
        return repository.findByCategoria(categoria);
    }

    public Aplicacion buscaAplicacion(String nombre) {
        return repository.findByNombre(nombre);
    }
}
```

Ilustración 8. Clase AplicacionService

4.4. *Repository Layer*

La capa *repository* es responsable de la interacción con la persistencia de datos y se basa en el *framework Spring Data JPA*, que es una implementación de la especificación JPA (*Java Persistence API*).

La persistencia de datos elegida en este proyecto es una base de datos MySQL. *Spring Data JPA* se integra de manera transparente con MySQL, lo que permite realizar operaciones de persistencia y consultas utilizando las capacidades y características propias de este motor de base de datos.

Al utilizar *Spring Data JPA* con MySQL, las entidades de dominio se mapean a tablas en la base de datos MySQL. Las anotaciones JPA, como *@Entity*, *@Id*, *@OneToOne* entre otros, se utilizan para definir el mapeo entre las entidades y las tablas, así como para especificar las columnas y restricciones de la base de datos.

Es importante mencionar que, de forma predeterminada, *Spring* utiliza JSON como formato de intercambio de datos, y para el mapeo entre objetos *Java* y JSON se utiliza la biblioteca *Jackson*.

La Ilustración 9 presenta el modelo de dominio, en el que puede observarse cómo una aplicación está vinculada a una categoría específica. Además, cada categoría está asociada a una serie de riesgos que pueden ser mitigados mediante diversos controles. Y cada control puede mitigar múltiples riesgos a la vez.

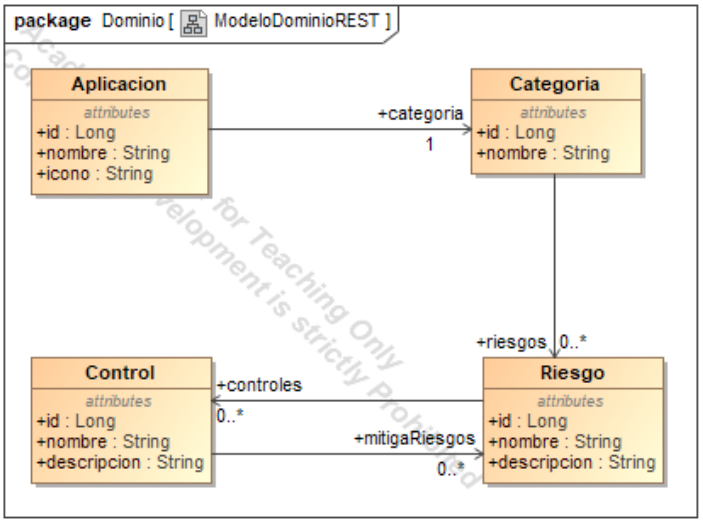


Ilustración 9. Modelo de Dominio de AppWise

El siguiente esquema muestra el modelo de la base de datos que corresponde al modelo de dominio presentado.

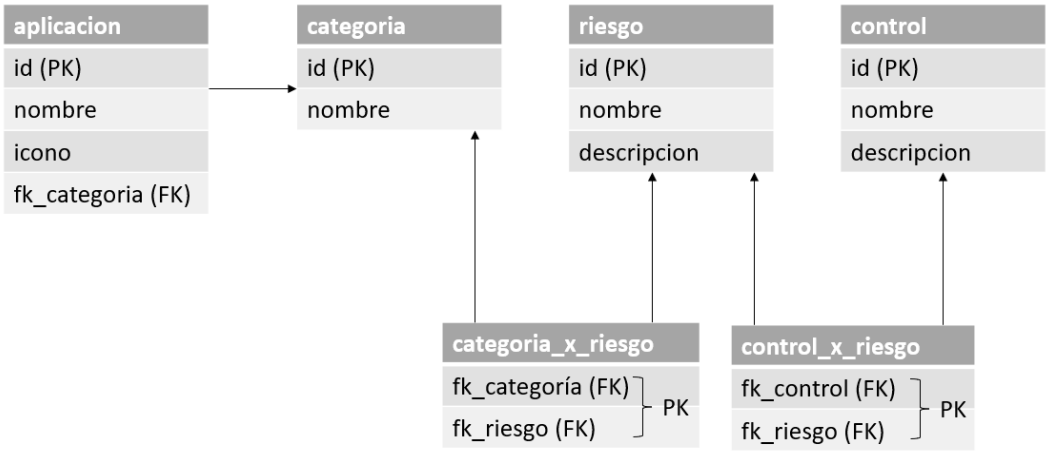


Ilustración 10. Modelo de la base de datos

En la Ilustración 11 se presenta los atributos de la clase *Aplicacion* para mostrar un ejemplo de cómo se anotan las entidades y conseguir el esquema de la base de datos mostrado anteriormente. Se anota mediante *@Entity* para denotar que es una entidad y a través de *@Id*, *@OneToOne* y *@JoinColumn* se especifican la *primary key* y *foreign key* respectivamente.

```
@Entity
public class Aplicacion {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    private String nombre;

    private String icono;

    @OneToOne
    @JoinColumn(name="fk_categoria")
    private Categoria categoria;
```

Ilustración 11. Clase Aplicación

Los repositorios se generan en tiempo de ejecución, solo es necesario definir su interfaz como extensión de las interfaces proporcionadas por *Spring Data JPA*, como *JpaRepository*. Estas interfaces heredan por defecto los métodos CRUD y permiten añadir métodos personalizados (buscar, leer, consultar, contar y obtener) sobre las entidades de forma declarativa, sin necesidad de escribir implementaciones de repositorio manualmente.

A continuación, se muestra un ejemplo a través de la clase *AplicacionRepository*. Se han añadido dos consultas personalizadas, simplemente declarando el método como "*find*" (la acción) y seguido de "*byCategoria*" (el criterio sobre la entidad).

```
public interface AplicacionRepository extends JpaRepository<Aplicacion, Long> {

    public List<Aplicacion> findByCategoria(Categoria categoria);

    public Aplicacion findByNombre(String nombre);
}
```

Ilustración 12. Clase AplicacionRepository

5. Diseño e implementación de la Aplicación Android

En esta sección se aborda el diseño y la implementación del proyecto *Android*, siguiendo el patrón Modelo Vista-Presentador (MVP). Se presenta una visión general de la arquitectura utilizada y se detalla el diseño y la implementación de cada una de las capas: modelo, vista y presentador.

5.1. Arquitectura

La arquitectura del proyecto se basa en el patrón MVP, que proporciona una separación clara de responsabilidades entre la lógica de negocio, la interfaz de usuario y la comunicación entre ambas.

En el patrón MVP, el modelo representa los datos y la lógica de negocio de la aplicación. Esta capa se encarga de acceder y gestionar los datos, así como de realizar operaciones relacionadas con la lógica de negocio. El modelo se comunica con las otras capas a través de interfaces definidas.

La vista es la capa encargada de la interfaz de usuario y la presentación de los datos al usuario. Se encarga de mostrar la información y recoger las interacciones del usuario. La vista no realiza ninguna lógica de negocio directamente, sino que se comunica con el presentador para solicitar los datos necesarios y enviar las acciones del usuario.

El presentador actúa como intermediario entre el modelo y la vista. Recibe las peticiones de la vista y se encarga de procesarlas, acceder al modelo si es necesario y devolver los resultados a la vista. El presentador también puede contener la lógica de negocio adicional que no corresponde al modelo ni a la vista.

El patrón MVP facilita la separación de responsabilidades, lo que permite una mejor organización del código y una mayor facilidad para realizar pruebas unitarias. Además, al separar la lógica de negocio de la interfaz de usuario, se mejora la mantenibilidad y la escalabilidad del proyecto.

A continuación, se muestra el diagrama de componentes que representa la arquitectura a alto nivel, incluyendo las interfaces clave. Hay 7 parejas vista-presentador:

- *Main*: es la única *Activity* de la aplicación, se encarga de gestionar la barra de navegación inferior y mostrar el *Fragment* correspondiente en función de la pestaña que se seleccione, así como configurar detalles de la barra de herramientas superior.
- *Apps*: gestiona la pestaña *Apps*, la cual muestra la lista completa de categorías y aplicaciones.
- *AppDetail*: gestiona la vista de detalle de una aplicación.
- *SearchResult*: gestiona la vista en la que se muestra el resultado de una búsqueda de aplicaciones.
- *Controles*: gestiona la pestaña Controles, la cual muestra la lista completa de controles.
- *ControlDetail*: gestiona la vista de detalle de un control.
- *Riesgos*: gestiona la pestaña Riesgos, la cual muestra la lista completa de riesgos.
- *RiesgoDetail*: gestiona la vista de detalla de un riesgo.
- *Perfil*: gestiona la pestaña Perfil, la cual muestra el nivel de riesgo del usuario junto con la lista de aplicaciones y controles que tiene agregados.

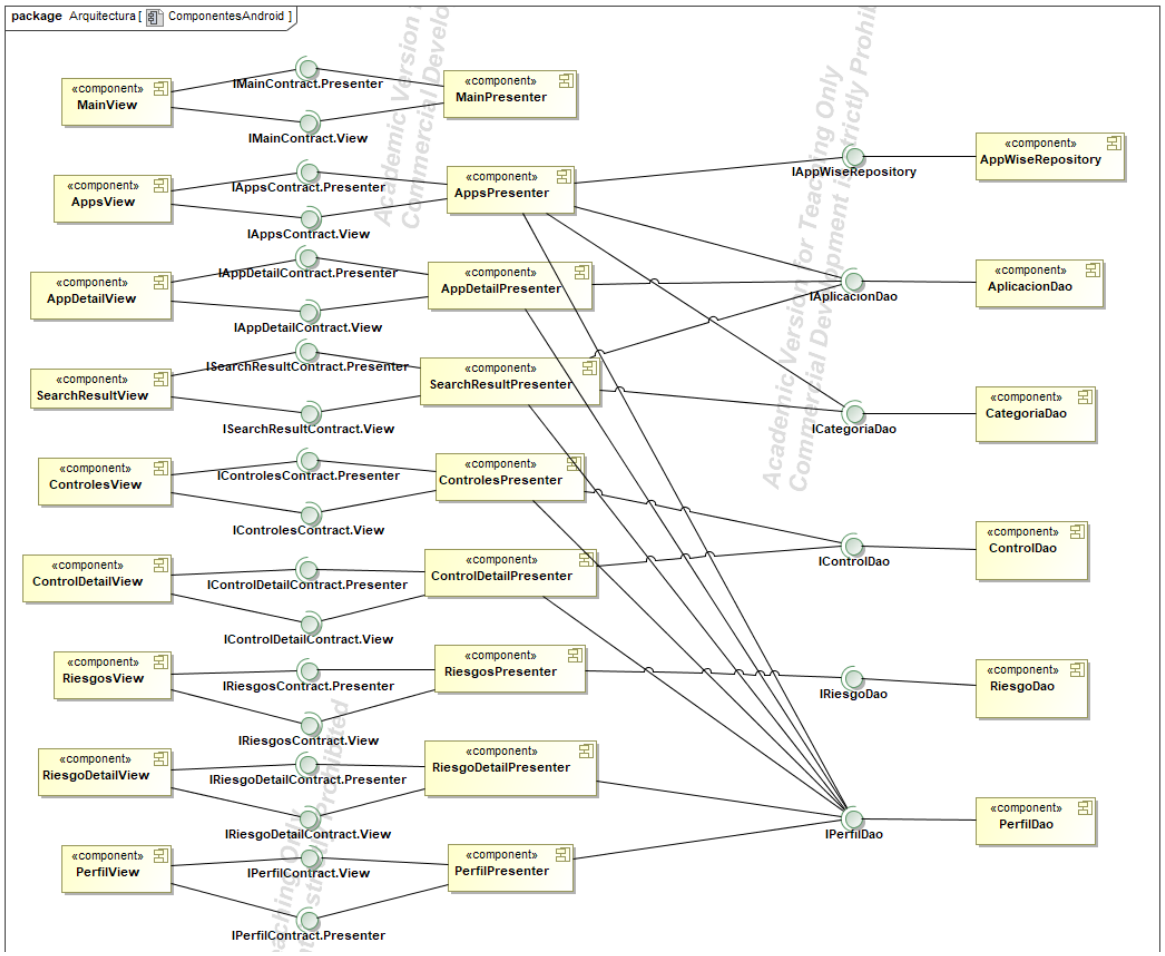
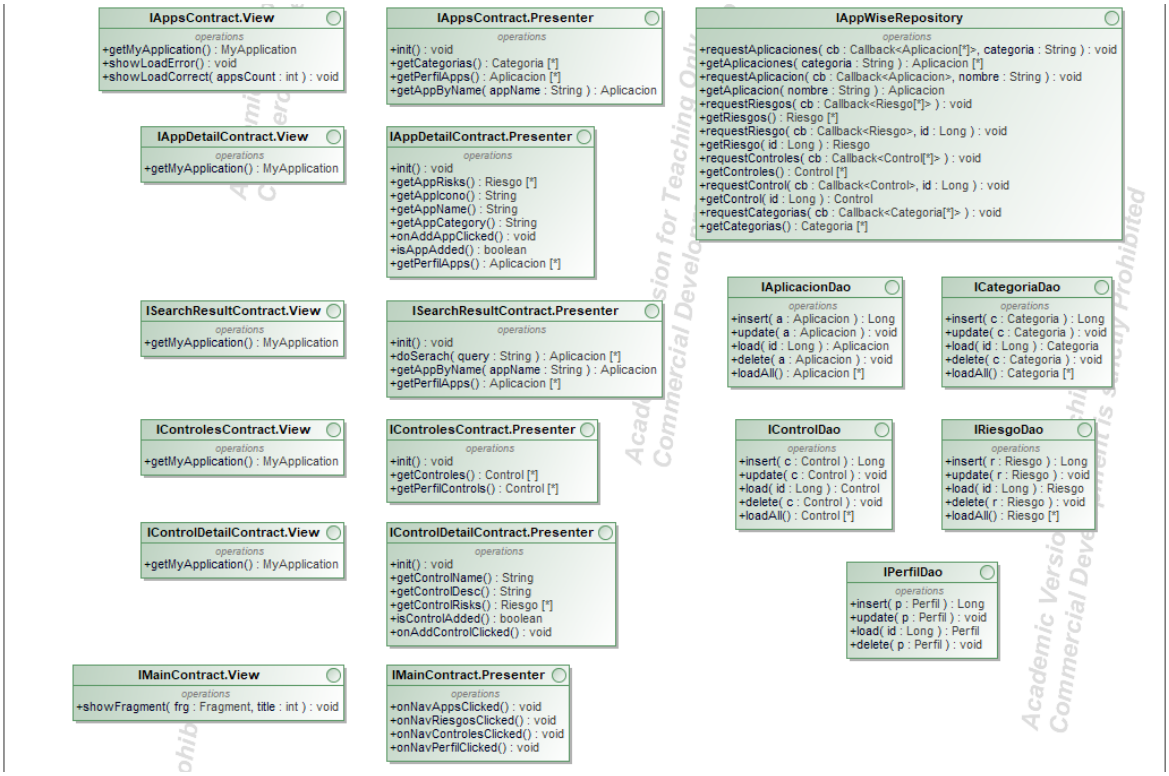


Ilustración 13. Diagrama de componentes de AppWise



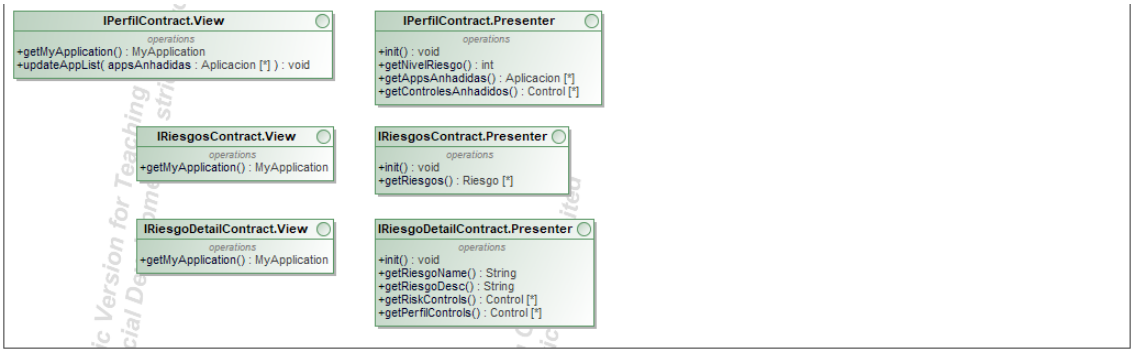


Ilustración 14. Descripción de interfaces de AppWise

5.2. Diseño e implementación del modelo

En esta sección se detalla el diseño y la implementación de la capa de modelo, que incluye la gestión de la base de datos local y el acceso a los datos del servicio REST.

5.2.1. Base de datos local

Como se ha mencionado anteriormente en la sección 6, se ha empleado *greenDAO* como una solución de mapeo objeto-relacional (ORM). *GreenDAO* proporciona una interfaz orientada a objetos para interactuar con la base de datos relacional *SQLite*. Al utilizar herramientas ORM como *greenDAO*, se simplifica significativamente el proceso de realizar tareas repetitivas relacionadas con la base de datos y nos brinda una forma más intuitiva y sencilla de trabajar con nuestros datos (GreenRobot, s.f.).

Ilustración 15. Esquema SQLite - GreenDAO (ORM)

Las siguientes clases principales son la interfaz esencial de *greenDAO*:

- *DaoMaster*: El punto de entrada para usar *greenDAO*. *DaoMaster* contiene el objeto de la base de datos (*SQLiteDatabase*) y administra las clases DAO (no los objetos) para un esquema específico. Tiene métodos estáticos para crear las tablas o eliminarlas. Sus clases internas *OpenHelper* y *DevOpenHelper* son implementaciones de *SQLiteOpenHelper* que crean el esquema en la base de datos *SQLite*.

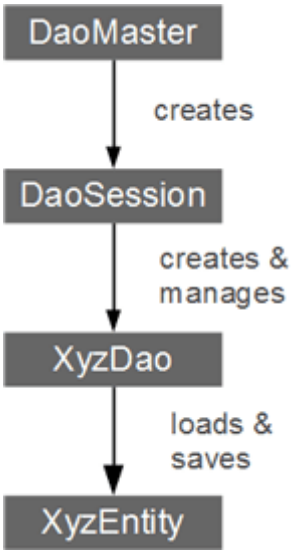


Ilustración 16. Clases principales greenDAO

- *DaoSession*: Gestiona todos los objetos DAO disponibles para un esquema específico, que se pueden adquirir utilizando uno de los métodos *getter*. *DaoSession* también proporciona algunos métodos de persistencia genéricos como *insert*, *load*, *update*, *refresh* y *delete* para las entidades. Por último, un objeto *DaoSession* también realiza un seguimiento de un ámbito de identidad.
- DAOs: Objetos de acceso a datos que persisten y realizan consultas para las entidades. Para cada entidad, *greenDAO* genera

una DAO. Tiene más métodos de persistencia que *DaoSession*, por ejemplo: *count*, *loadAll* e *insertInTx*. Estas son las clases con las que se comunican las clases *presenter*.

- Entidades: Objetos persistentes. Por lo general, las entidades son objetos que representan una fila de la base de datos utilizando propiedades estándar de Java (como un POJO o un JavaBean).

Una vez que se definen las entidades con las anotaciones correspondientes, al construir el proyecto se autogeneran las clases descritas y se habilita el acceso a los datos de la base de datos local.

En la Ilustración 17 se muestra el modelo de dominio de la aplicación, es idéntico al dominio del servicio REST, pero añadiendo un *Perfil* para poder almacenar las aplicaciones y controles que este tiene añadidos, de esta manera también se puede añadir en un futuro distintos perfiles como se explica en la sección Trabajo futuro.

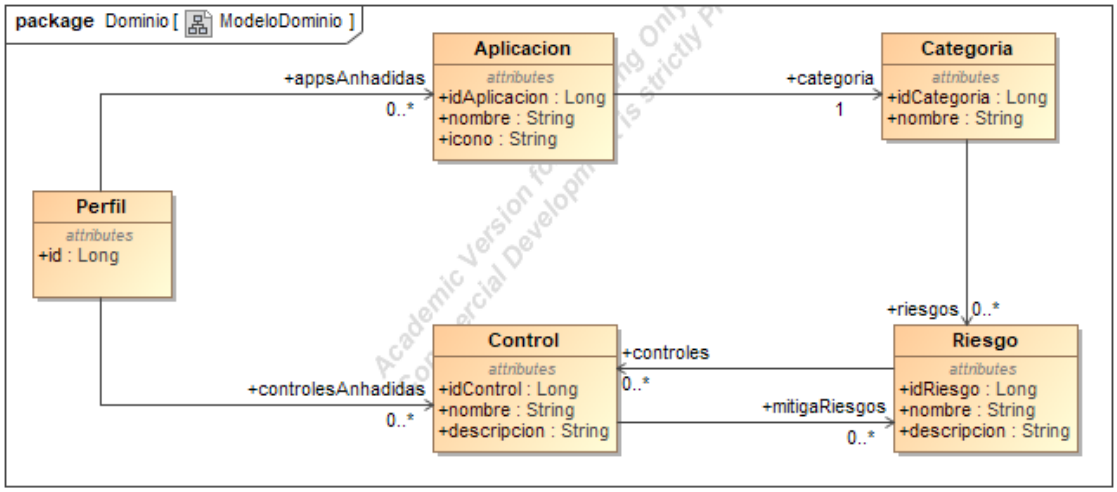


Ilustración 17. Modelo de Dominio de AppWise

En la siguiente ilustración se muestra un ejemplo a través de la clase *Riesgo*, de cómo se anotan las entidades. La anotación principal es *@Entity* para denotar que se trata de una entidad, *@Id* indica la *primary key* y *@ToMany* junto con *@JoinEntity* configuran la *foreign key*.

```
@Entity
public class Riesgo implements Parcelable {
    @NonNull
    @Id
    private Long idRiesgo;

    private String nombre;

    private String descripcion;

    @ToMany
    @JoinEntity(
            entity = JoinRiesgosWithControles.class,
            sourceProperty = "idRiesgo",
            targetProperty = "idControl"
    )
    private List<Control> controles;
```

Ilustración 18. Clase Riesgo

A continuación, en la Ilustración 19 se puede observar con el método *init()* de la clase *AppsPresenter* un ejemplo de cómo acceder a los datos de la base de datos local usando las clases descritas. A través de un objeto *MyApplication* (cuya implementación se muestra en la Ilustración 20), se obtiene la *DaoSession* de la que se pueden extraer las distintas DAO.

NOTA: Aunque en el Diagrama de componentes se muestran explícitamente las interfaces entre el presentador y las DAO, en la implementación el presentador accede directamente a las DAO autogeneradas por *greenDAO*.

```
public void init() {
    DaoSession daoSession = view.getMyApplication().getDaoSession();
    aplicacionDao = daoSession.getAplicacionDao();
    categoriaDao = daoSession.getCategoriaDao();
    perfilDao = daoSession.getPerfilDao();

    if (repository == null) {
        repository = new AppWiseRepository(view.getMyApplication());
        doSyncInit();
    }
}
```

Ilustración 19. Código para el acceso a los datos

```
public class MyApplication  extends Application {
    private DaoSession daoSession;

    @Override
    public void  onCreate() {
        super.onCreate();
        CustomOpenHelper helper = new CustomOpenHelper( context: this,  name: "appwise-db");
        Database db = helper.getWritableDb();
        daoSession = new DaoMaster(db).newSession();
    }
    public DaoSession getDaoSession() { return daoSession; }

    public static class CustomOpenHelper extends DaoMaster.OpenHelper {
        public CustomOpenHelper(Context context, String name) { super(context, name); }

        @Override
        public void onCreate(Database db) { super.onCreate(db); }
    }
}
```

Ilustración 20. Clase MyApplication

5.2.2. Acceso al Servicio REST

Para hacer las solicitudes al servicio REST se ha usado la librería *Retrofit*, tal y como se mencionaba en el apartado 6. A continuación, se muestra la estructura de clases que se ha empleado, que incluye la clase *AppWiseRepository* para añadir una capa intermedia entre el servicio y los presentadores, *AppWiseService* para el acceso a datos desde el servicio y las DAO para el acceso a la base de datos local.

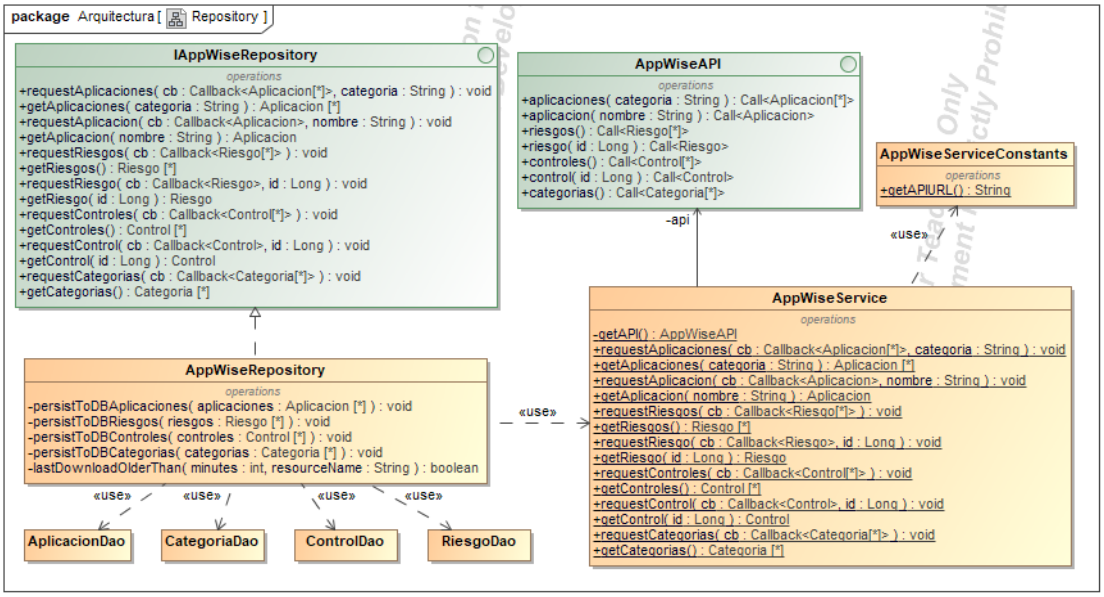


Ilustración 21. Diagrama de clases del acceso a los datos

Concretamente, estas clases desempeñan las siguientes funciones:

- *AppWiseAPI*: Esta interfaz define los métodos que se utilizarán para ejecutar las solicitudes HTTP. Cada método de esta interfaz está anotado con una anotación HTTP que proporciona el método de solicitud (en este caso solo GET) y la dirección URL relativa para la solicitud. Además, se especifica la clase que se encargará de procesar la respuesta obtenida del servicio. Esta interfaz actúa como un contrato que define las operaciones disponibles en el servicio REST.

```
public interface AppWiseAPI {
    @GET("apps")
    Call<Aplicacion[]> aplicaciones(@Query("categoria") String categoria);

    @GET("apps/{nombre}")
    Call<Aplicacion> aplicacion(@Path("nombre") String nombre);

    @GET("riesgos")
    Call<Riesgo[]> riesgos();

    @GET("riesgos/{id}")
    Call<Riesgo> riesgo(@Path("id") Long id);

    @GET("controles")
    Call<Control[]> controles();

    @GET("controles/{id}")
    Call<Control> control(@Path("id") Long id);

    @GET("categorias")
    Call<Categoria[]> categorias();
}
```

Ilustración 22. Interfaz AppWiseAPI

- *AppWiseService*: Esta clase es responsable de instanciar un objeto *Retrofit*, que actúa como el cliente HTTP para realizar las peticiones al servicio REST. Siguiendo el patrón de diseño *Singleton*, se garantiza que solo exista una instancia de la API en toda la aplicación. El objeto *Retrofit* se configura con la URL base del servicio y otros parámetros relevantes, como el convertidor *Gson*

para la serialización y deserialización de datos en formato JSON. Esta clase proporciona métodos para realizar las solicitudes de manera asíncrona a través de *callbacks*, así como de manera síncrona, lo que permite elegir la opción más adecuada según las necesidades de la aplicación.

```
public class AppWiseService {

    private static AppWiseAPI api;

    private static AppWiseAPI getAPI() {
        if (api == null) {
            Retrofit retrofit = new Retrofit.Builder()
                    .baseUrl(AppWiseServiceConstants.getAPIURL())
                    .addConverterFactory(GsonConverterFactory.create())
                    .build();

            api = retrofit.create(AppWiseAPI.class);
        }
        return api;
    }
```

Ilustración 23. Clase AppWiseService

- *AppWiseRepository*: Esta clase es la responsable de realizar las solicitudes al servicio REST utilizando el objeto *AppWiseService*. Actúa como una capa intermedia entre la capa de presentador y el servicio REST. Además de realizar las solicitudes, también se encarga de persistir los datos en la base de datos local, utilizando la capa de persistencia implementada con *GreenDAO*. Esto permite mantener una copia local de los datos para su acceso rápido y sin conexión a Internet.

5.3. Diseño e implementación de la vista

La vista en el patrón MVP tiene la responsabilidad de mostrar la interfaz de usuario de la aplicación y responder a las interacciones del usuario.

A continuación, se muestra el diagrama de navegación de la aplicación, que representa las diferentes pantallas y las transiciones entre ellas. Este diagrama permite tener una visión general de la estructura de la aplicación y cómo se relacionan las distintas vistas.

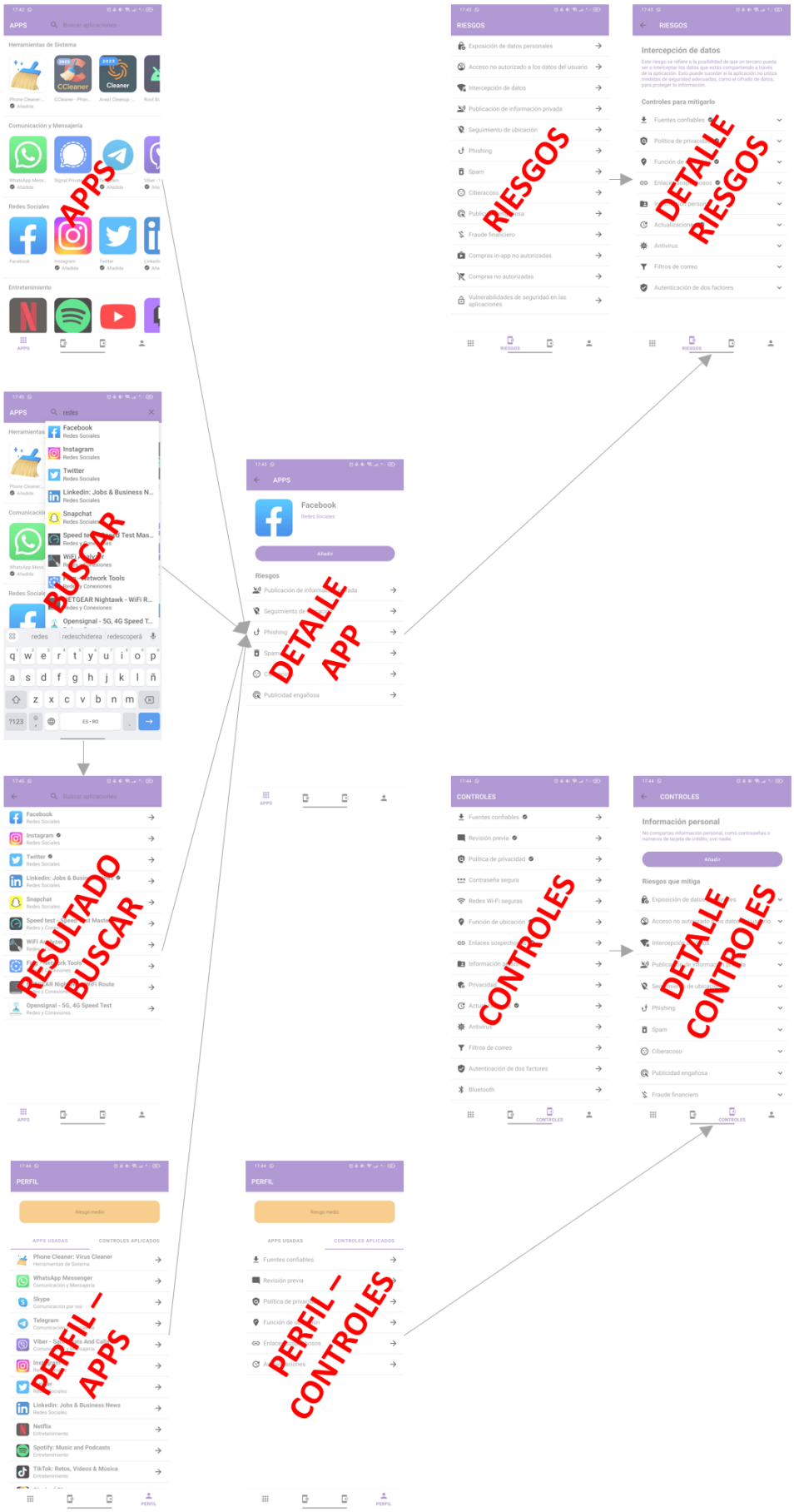


Ilustración 24. Diagrama de navegación de AppWise

En todas las vistas de la aplicación, se utiliza el componente *RecyclerView*, el cual facilita que se muestren de manera eficiente grandes conjuntos de datos. Al proporcionar los datos y definir la apariencia de cada elemento, la biblioteca *RecyclerView* genera los elementos de forma dinámica cuando se requieren.

Como su nombre lo indica, el *RecyclerView* recicla los elementos individuales. Cuando un elemento se desplaza fuera de la pantalla, en lugar de destruir su vista, el *RecyclerView* la reutiliza para mostrar nuevos elementos que se han desplazado y ahora están en pantalla. Esto mejora significativamente el rendimiento, la capacidad de respuesta de la aplicación y reduce el consumo de energía (Android Developers, 2021).

Varias clases trabajan en conjunto para construir la lista dinámica:

- *RecyclerView*: es el *ViewGroup* que contiene las vistas correspondientes a los datos. Es una vista en sí misma y se agrega al diseño de la interfaz de usuario de manera similar a otros elementos.

```
<FrameLayout xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    tools:context="es.unican.appriegospersonales.activities.riesgos.RiesgosView">

    <androidx.recyclerview.widget.RecyclerView
        android:id="@+id/riesgos_rv"
        android:layout_width="match_parent"
        android:layout_height="match_parent"/>

</FrameLayout>
```

Ilustración 25. Layout de la vista "Riesgos" con un RecyclerView

- *ViewHolder*: define un contenedor de vistas para cada elemento individual de la lista. Inicialmente, este contenedor no tiene datos asociados, pero el *RecyclerView* lo vincula a los datos correspondientes. Para definir el *ViewHolder*, se extiende la clase *RecyclerView.ViewHolder*.

```
public class RiesgoViewHolder extends RecyclerView.ViewHolder implements View.OnClickListener {

    private final TextView riesgoName_tv;
    private final ImageView riesgoIcon_iv;
    private Riesgo riesgo;

    public RiesgoViewHolder(@NonNull View itemView) {
        super(itemView);
        riesgoName_tv = itemView.findViewById(R.id.riesgoName_tv);
        riesgoIcon_iv = itemView.findViewById(R.id.riesgoIcon_iv);
        itemView.setOnClickListener(this);
    }

    @Override
    public void onClick(View view) {
        FragmentManager fragmentManager = ((AppCompatActivity) context).getSupportFragmentManager();
        fragmentManager.beginTransaction()
                .hide(Objects.requireNonNull(fragmentManager.findFragmentById(R.id.container)))
                .add(R.id.container, RiesgoDetailView.newInstance(riesgo))
                .setReorderingAllowed(true)
                .addToBackStack("riesgos")
                .commit();
    }
}
```

Ilustración 26. Clase RiesgoViewHolder

– *Adapter*: el *RecyclerView* solicita las vistas y las vincula a los datos a través de llamadas a métodos en el adaptador. Para definir el adaptador, se extiende la clase *RecyclerView.Adapter*.

```
public class RVRiesgosAdapter extends RecyclerView.Adapter<RVRiesgosAdapter.RiesgoViewHolder> {

    private final Context context;
    private final List<Riesgo> riesgos;
    private final LayoutInflater inflater;

    public RVRiesgosAdapter(Context context, List<Riesgo> riesgos) {
        this.riesgos = riesgos;
        this.context = context;
        this.inflater = LayoutInflater.from(context);
    }

    @NonNull
    @Override
    public RiesgoViewHolder onCreateViewHolder(@NonNull ViewGroup parent, int viewType) {
        View view = inflater.inflate(R.layout.rv_riesgos_riesgo, parent,  attachToRoot: false);
        return new RiesgoViewHolder(view);
    }

    @Override
    public void onBindViewHolder(@NonNull RiesgoViewHolder holder, int position) {
        Riesgo riesgo = riesgos.get(position);
        holder.riesgo = riesgo;
        holder.riesgoName_tv.setText(riesgo.getNombre());
        int iconoId = context.getResources().getIdentifier(
                name: "risk" + riesgo.getIdRiesgo(),
                defType: "drawable",
                context.getPackageName());
        if (iconoId != 0) {
            holder.riesgoIcon_iv.setImageResource(iconoId);
        }
    }

    @Override
    public int getItemCount() { return riesgos.size(); }
```

Ilustración 27. Clase RVRiesgosAdapter

– *LayoutManager*: es el administrador de diseño que organiza los elementos individuales de la lista. Se pueden utilizar los administradores de diseño proporcionados por la biblioteca *RecyclerView* o se puede definir uno personalizado. Todos los administradores de diseño se basan en la clase abstracta *LayoutManager* de la biblioteca.

En la siguiente ilustración se puede ver cómo se configuran todos los aspectos del *RecyclerView* correspondiente a la lista de riesgos: el *layout*, el *LayoutManager*, el *Adapter* y una línea a modo de división y decoración entre los elementos.

```
public View onCreateView(LayoutInflater inflater, @Nullable ViewGroup container, @Nullable
        Bundle savedInstanceState) {
    View layout = inflater.inflate(R.layout.fragment_riesgos, container,  attachToRoot: false);
    riesgos_rv = layout.findViewById(R.id.riesgos_rv);
    riesgos_rv.setLayoutManager(new LinearLayoutManager(getContext()));
    riesgos_rv.setAdapter(new RVRiesgosAdapter(getContext(), presenter.getRiesgos()));

    DividerItemDecoration dividerItemDecoration = new DividerItemDecoration(
            riesgos_rv.getContext(),
            DividerItemDecoration.VERTICAL);
    riesgos_rv.addItemDecoration(dividerItemDecoration);

    return layout;
}
```

Ilustración 28. Método onCreateView de la clase RiesgosView

5.4. Diseño e implementación del presentador

El presentador en el patrón MVP actúa como una capa intermedia entre el modelo y la vista. Su responsabilidad principal es recibir las peticiones provenientes de la vista, procesarlas y, si es necesario, acceder al modelo para obtener los datos requeridos. Posteriormente, devuelve los resultados obtenidos a la vista para que sean presentados al usuario.

En el contexto de esta aplicación, los presentadores están implementados como clases *Java* POJO (*Plain Old Java Object*) que cumplen con las interfaces definidas en los contratos establecidos con las vistas. Estas interfaces se encuentran definidas en el diagrama de componentes y definición de interfaces, como se muestra en la Ilustración 14.

6. Pruebas

Las pruebas son una parte fundamental en el desarrollo de software, ya que nos permiten verificar y validar el correcto funcionamiento del proyecto. En este apartado, se detallarán las pruebas llevadas a cabo para asegurar que tanto el servicio REST como la aplicación *Android* funcionan correctamente. Los distintos tipos de pruebas que se han abordado son los siguientes:

- Pruebas unitarias: se centran en verificar el comportamiento individual y aislado de componentes o unidades de código, cómo métodos o funciones. Estas pruebas se realizan de manera independiente, aislando la unidad de código bajo prueba de sus dependencias externas, utilizando *mocks* o *stubs* para simular el comportamiento de esas dependencias.

  Tanto para el servicio como para la aplicación, se ha seguido la técnica de prueba de métodos utilizando enfoques de caja negra, como son la partición equivalente y el AVL, para diseñar casos de prueba exhaustivos para cada método en las clases correspondientes. Además, se ha garantizado la cobertura condición/decisión. Para llevar a cabo estas pruebas, se ha hecho uso de las bibliotecas *JUnit* y *Mockito*, que proporcionan herramientas y funcionalidades para realizar pruebas unitarias y crear objetos simulados.

- Pruebas de integración: se realizan para verificar la interacción y el funcionamiento conjunto de diferentes componentes o módulos del sistema.

  Para estas pruebas se han utilizado los datos conocidos que se almacenan en la base de datos del servicio.

- Pruebas de sistema: se realizan para verificar el funcionamiento completo y global del sistema, se enfocan en validar que el sistema cumpla con los requisitos no funcionales establecidos.

- Pruebas de aceptación: se realizan para validar si el sistema cumple con las expectativas del usuario, validando su usabilidad, funcionalidad y satisfacción del usuario.

6.1. Pruebas unitarias del Servicio REST

Las clases *Service* y *Controller* son las que se han probado para garantizar su correcto funcionamiento. Sin embargo, dado que los repositorios se generan dinámicamente a partir de las interfaces definidas, se ha considerado que no era necesario realizar pruebas específicas para ellos.

A continuación, se presentan detalladamente los casos de prueba diseñados para las clases *AplicacionService* y *AplicacionController*, donde se describen las entradas utilizadas y las salidas esperadas.

Y más adelante, se puede visualizar la implementación de las pruebas UAS.1x y UAC.1x, las cuales proporcionan un vistazo general de cómo se han codificado y ejecutado las pruebas unitarias.

+aplicaciones() : List<Aplicacion>

Identificador	Entrada	Salida esperada
UAS.1a	(existen aplicaciones en el sistema)	Listado con aplicaciones
UAS.1b	(no existen aplicaciones en el sistema)	Listado vacío

Tabla 6. Casos de prueba método aplicaciones(), clase AplicacionService

+aplicacionesPorCategoria(categoria : Categoria) : List<Aplicacion>

Identificador	Entrada	Salida esperada
UAS.2a	Redes Sociales (categoría existente)	Listado con aplicaciones
UAS.2b	*null*	Listado vacío

Tabla 7. Casos de prueba método aplicacionesPorCategoria(), clase AplicacionService

+buscaAplicacion(nombre : String) : Aplicacion

Identificador	Entrada	Salida esperada
UAS.3a	Facebook (aplicación existente)	Aplicación con nombre “Facebook”
UAS.3b	XXX (aplicación no existente)	*null*

Tabla 8. Casos de prueba método buscaAplicacion(), clase AplicacionService

+getAplicaciones(categoría: String) : List<Aplicacion>

Identificador	Entrada	Salida esperada
UAC.1a	Redes Sociales (categoría existente)	200 OK Listado con aplicaciones
UAC.1b	XXX (categoría no existente)	200 OK Listado vacío
UAC.1c	*null*	200 OK Listado con todas las aplicaciones

Tabla 9. Casos de prueba método getAplicaciones(), clase AplicacionController

+getAplicacion(nombre: String) : ResponseEntity<Aplicación>

Identificador	Entrada	Salida esperada
UAC.2a	Facebook (aplicación existente)	200 OK Aplicación con nombre “Facebook”
UAC.2b	XXX (aplicación no existente)	404 Not Found

Tabla 10. Casos de prueba método getAplicacion(), clase AplicacionController

```
public class AplicacionServiceTest {

    @Mock
    private AplicacionRepository repository;

    @InjectMocks
    private AplicacionService sut;

    @BeforeEach
    public void setUp() {
        MockitoAnnotations.openMocks(this);
    }

    @Test
    public void testAplicacionesExistentes() {
        // UAS.1a
        Aplicacion a1 = new Aplicacion();
        Aplicacion a2 = new Aplicacion();
        List<Aplicacion> apps = new ArrayList<Aplicacion>();
        apps.add(a1); apps.add(a2);

        when(repository.findAll()).thenReturn(apps);

        assertEquals(apps, sut.aplicaciones());
    }

    @Test
    public void testAplicacionesNoExistentes() {
        // UAS.1b
        when(repository.findAll()).thenReturn(Collections.emptyList());

        assertEquals(Collections.emptyList(), sut.aplicaciones());
    }
```

Ilustración 29. Pruebas UAS.1x de AplicacionService

```
public class AplicacionControllerTest {

    @Mock
    private AplicacionService service;

    @Mock
    private CategoriaService catService;

    @InjectMocks
    private AplicacionController sut;

    @BeforeEach
    public void setUp() {
        MockitoAnnotations.openMocks(this);
    }

    @Test
    public void testGetAplicaciones() {
        // UAC.1a
        String nombreCategoria = "Redes sociales";
        Categoria c = new Categoria();
        c.setNombre(nombreCategoria);
        Aplicacion a1 = new Aplicacion();
        Aplicacion a2 = new Aplicacion();
        List<Aplicacion> appsEsperadas = new ArrayList<Aplicacion>();
        appsEsperadas.add(a1); appsEsperadas.add(a2);

        when(catService.buscaCategoriaPorNombre(nombreCategoria)).thenReturn(c);
        when(service.aplicacionesPorCategoria(c)).thenReturn(appsEsperadas);

        assertEquals(appsEsperadas, sut.getAplicaciones(nombreCategoria));
```

```
    // UAC.1b
    String nombreCategoriaNoExist = "XXX";

    when(catService.buscaCategoriaPorNombre(nombreCategoriaNoExist)).thenReturn(null);
    when(service.aplicacionesPorCategoria(null)).thenReturn(Collections.emptyList());

    assertEquals(Collections.emptyList(), sut.getAplicaciones(nombreCategoriaNoExist));

    // UAC.1c
    when(service.aplicaciones()).thenReturn(appsEsperadas);

    assertEquals(appsEsperadas, sut.getAplicaciones(null));
}
```

Ilustración 30. Pruebas UAC.1x de AplicacionController

6.2. *Pruebas de integración del Servicio REST*

La estrategia para la definición del orden de las pruebas de integración ha sido incremental. Se ha probado:

- La integración entre la capa *service* y la capa *repository*.
- La integración entre la capa *controller* y las anteriores.

De igual manera, se ha seguido la técnica de prueba de métodos utilizando enfoques de caja negra (partición equivalente AVL) para la definición de los casos de prueba. Para llevar a cabo las pruebas se ha usado *JUnit*.

Se han usado los mismos casos de prueba definidos para las pruebas unitarias. Por ejemplo, las pruebas definidas como UAS.x y UAC.x para las clases *AplicacionService* y *AplicacionController* en la Tabla 6 hasta la Tabla 10, aquí se han renombrado como IAS.x y IAC.x.

A continuación, se muestra la implementación de las pruebas IAS.1x y IAC.1x las cuales proporcionan un vistazo general de cómo se han codificado y ejecutado las pruebas de integración.

```
@SpringBootTest
@Transactional
public class AplicacionServiceTestIT {
    @Autowired
    private AplicacionService sut;

    @Autowired
    private AplicacionRepository repository;

    @Autowired
    private CategoriaService catService;

    @Test
    public void testAplicacionesExistentes() {
        // IAS.1a
        List<Aplicacion> apps = sut.aplicaciones();
        assertTrue(apps.size() >= 50);

        // IAS.1b
        repository.deleteAll();
        assertTrue(sut.aplicaciones().isEmpty());
    }
```

Ilustración 31. Pruebas IAS.1x de AplicacionService

```java
@SpringBootTest
@AutoConfigureMockMvc
public class AplicacionControllerIT {

    @Autowired
    private MockMvc mockMvc;

    @Test
    public void testGetAplicaciones() throws Exception {
        // IAC.1a
        mockMvc.perform(get("/apps?categoria=Redes Sociales")
                .contentType(MediaType.APPLICATION_JSON))
                .andExpect(status().isOk())
                .andExpect(jsonPath("$", hasSize(greaterThanOrEqualTo(5))));

        // IAC.1b
        mockMvc.perform(get("/apps?categoria=XXX")
                .contentType(MediaType.APPLICATION_JSON))
                .andExpect(status().isOk())
                .andExpect(jsonPath("$", hasSize(0)));

        // IAC.1c
        mockMvc.perform(get("/apps")
                .contentType(MediaType.APPLICATION_JSON))
                .andExpect(status().isOk())
                .andExpect(jsonPath("$", hasSize(greaterThanOrEqualTo(50))));
    }
```

Ilustración 32. Pruebas IAC.1x de AplicacionController

6.3. *Pruebas de aceptación del Servicio REST*

Se ha realizado una evaluación manual de la funcionalidad del servicio implementado utilizando la herramienta *Postman*.

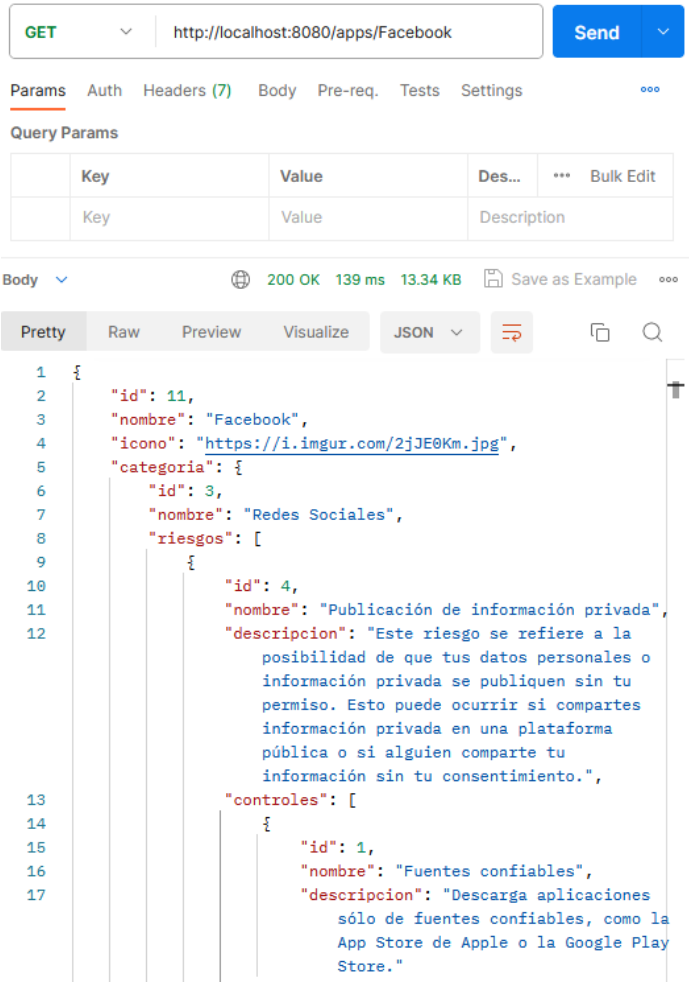


Ilustración 33. Uso de Postman para verificar la funcionalidad del servicio

6.4. Pruebas unitarias de la Aplicación Android

Las pruebas unitarias que se han realizado en la aplicación son principalmente de las clases *Presenter* ya que son las que contienen la lógica de negocio de la aplicación, para ello ha sido necesario el uso de *mocks* del resto de componentes. A continuación, se presentan detalladamente los casos de prueba diseñados para la clase *AppsPresenter*, donde se describen las entradas utilizadas y las salidas esperadas.

Y más adelante, se puede visualizar la implementación de las pruebas UAP.2x las cuales proporcionan un vistazo general de cómo se han codificado y ejecutado las pruebas unitarias.

+init() : void

Identificador	Contexto/Entrada	Resultado esperado
UAP.1a	No se producen errores	Se llama a *view.showLoadCorrect()*
UAP.1b	Al llamar a *repository.getControles()* retorna *null* porque ocurre un error en el acceso al servicio REST	Se llama a *view.showLoadError()*
UAP.1c	Al llamar a *repository.getRiesgos()* retorna *null* porque ocurre un error en el acceso al servicio REST	Se llama a *view.showLoadError()*
UAP.1d	Al llamar a *repository.getCategorias()* retorna *null* porque ocurre un error en el acceso al servicio REST	Se llama a *view.showLoadError()*
UAP.1e	Al llamar a *repository.getAplicaciones()* retorna *null* porque ocurre un error en el acceso al servicio REST	Se llama a *view.showLoadError()*

Tabla 11. Casos de prueba método init(), clase AppsPresenter

+getCategorias() : List<Categoria>

Identificador	Contexto/Entrada	Resultado esperado
UAP.2a	Existen categorías en la base de datos local	Se retorna una lista de categorías
UAP.2b	No existen categorías en la base de datos local	Se retorna una lista vacía
UAP.2c	Al llamar a *categoriaDao.loadAll()* se produce una excepción *SQLite* (error en la carga de datos para recoger las categorías)	Se llama a *view.showLoadError()* Se retorna *null*

Tabla 12. Casos de prueba método getCategorias(), clase AppsPresenter

+getPerfilApps() : List<Aplicacion>

Identificador	Contexto/Entrada	Resultado esperado
UAP.3a	El perfil tiene aplicaciones añadidas	Se retorna una lista de aplicaciones
UAP.3b	El perfil no tiene aplicaciones añadidas	Se retorna una lista vacía
UAP.3c	Al llamar a *Perfil.getInstance(perfilDao)* se produce una excepción *SQLite* (error en la carga del perfil)	Se llama a *view.showLoadError()* Se retorna *null*

Tabla 13. Casos de prueba método getPerfilApps(), clase AppsPresenter

+getAppByName(appName : String) : Aplicacion

Identificador	Contexto/Entrada	Resultado esperado
UAP.4a	“Facebook” (La aplicación existe)	Aplicación con nombre Facebook
UAP.4b	“XXX” (La aplicación no existe)	*null*

Tabla 14. Casos de prueba método getAppByName(), clase AppsPresenter

```
@Test
public void testGetCategoriasExisten() {
    // UAP.2a
    configInitCorrecto();
    List<Categoria> categorias = new ArrayList<>();
    categorias.add(new Categoria());

    when(daoSessionMock.getCategoriaDao()).thenReturn(categoriaDaoMock);
    when(categoriaDaoMock.loadAll()).thenReturn(categorias);

    sut.init();
    List<Categoria> categoriasSut = sut.getCategorias();
    assert(categorias.size() == categoriasSut.size());
    assert(categoriasSut.containsAll(categorias));
    verify(categoriaDaoMock).loadAll();
}

@Test
public void testGetCategoriasNoExisten() {
    // UAP.2b
    configInitCorrecto();
    List<Categoria> categorias = new ArrayList<>();

    when(daoSessionMock.getCategoriaDao()).thenReturn(categoriaDaoMock);
    when(categoriaDaoMock.loadAll()).thenReturn(categorias);

    sut.init();
    assert(sut.getCategorias().isEmpty());
    verify(categoriaDaoMock).loadAll();
}

@Test
public void testGetCategoriasError() {
    // UAP.2c
    configInitCorrecto();

    when(daoSessionMock.getCategoriaDao()).thenReturn(categoriaDaoMock);
    when(categoriaDaoMock.loadAll()).thenThrow(new SQLiteException());

    sut.init();
    assert(sut.getCategorias() == null);
    verify(categoriaDaoMock).loadAll();
    verify(viewMock).showLoadError();
}
```

Ilustración 34. Pruebas UAP.2x de AppsPresenter

6.5. *Pruebas de integración de la Aplicación Android*

Las pruebas de integración de la aplicación se han llevado a cabo de forma incremental y manual. El objetivo ha sido verificar que las nuevas funcionalidades agregadas cumplan con su propósito de manera correcta y eficiente, sin impactar negativamente los procesos previamente verificados.

En un principio, la intención era comprobar la integración entre capas usando *Roboelectric* para las pruebas del presentador con la capa de acceso a datos, y *Espresso* para las pruebas de la vista con las capas anteriores. Sin embargo, debido a la cantidad de tiempo y problemas que estaba llevando se decidió hacer comprobaciones mediante ejecuciones manuales tras cada iteración comprobando las nuevas funcionalidades.

Para lograr esto, al finalizar cada iteración, se ha ejecutado la aplicación tanto en el emulador de *Android Studio* con la versión *Oreo*, como en un dispositivo físico con la versión R en modo depuración comprobando las características añadidas y garantizando que funcionen según lo esperado. Además, se ha prestado especial atención a la interacción y compatibilidad con las funciones ya existentes, asegurando que no se produzcan conflictos ni deterioros en el rendimiento.

Este enfoque incremental ha permitido detectar y solucionar rápidamente cualquier problema, evitando que se propaguen y se conviertan en un obstáculo para el correcto funcionamiento de la aplicación. Al realizar las pruebas en un entorno de producción simulado, se ha obtenido una visión más realista de la experiencia del usuario y se ha garantizado la calidad del software en su conjunto.

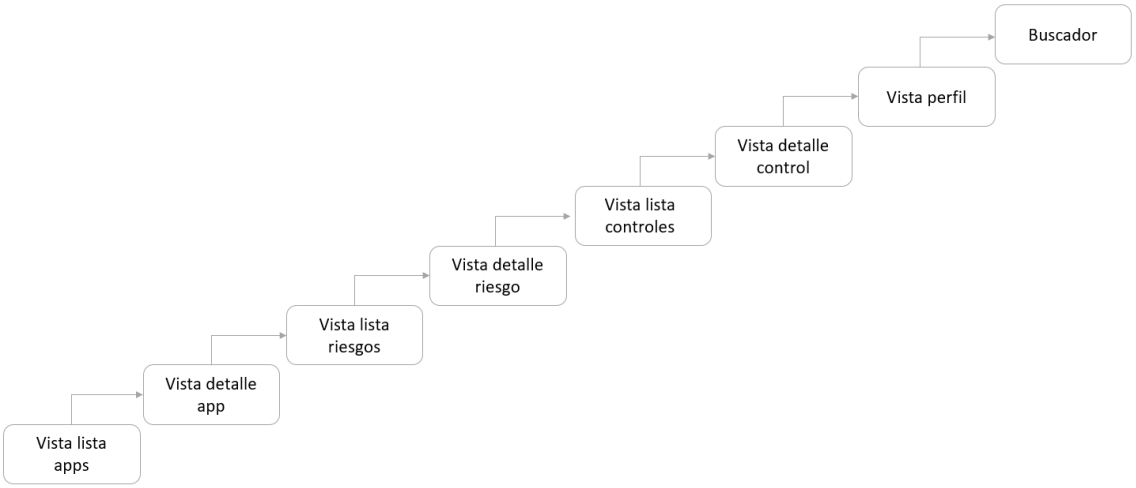


Ilustración 35. Esquema pruebas de integración de la aplicación

6.6. *Pruebas de sistema de la Aplicación Android*

En este apartado se detallarán las pruebas realizadas para cumplir los diversos requisitos no funcionales establecidos inicialmente para la aplicación.

6.6.1. Pruebas de portabilidad

El requisito de portabilidad que debía cumplir la aplicación es ser compatible y ejecutarse en diferentes versiones de Android, desde Android 8.1 Oreo hasta las versiones más recientes.

Durante las pruebas, se utilizó un emulador con Android 8.1, otro con Android 12 y un dispositivo físico con Android 11. Se verificó que la aplicación funcionara

correctamente en ambos entornos, sin presentar ningún fallo o problema significativo. Con esto se consideró que la aplicación cumple con el requisito de portabilidad establecido.

6.6.2. Pruebas de rendimiento

Dentro de los requisitos de rendimiento establecidos para la aplicación, se consideraron dos aspectos clave: el tiempo de carga inicial y el tiempo de respuesta de la interfaz de usuario. El objetivo es lograr una experiencia ágil y sin demoras perceptibles para el usuario, estableciendo un límite de 3 segundos para la carga inicial y 400 milisegundos para las respuestas a los eventos, como toques o desplazamientos.

Para evaluar este requisito, se ha escogido la pestaña “Apps” ya que además de ser la pantalla principal de la aplicación, es la que requiere acceder al servicio REST para descargar los datos, lo que influye significativamente en el tiempo de carga. Con el fin de medir con precisión este tiempo, se han realizado 5 ejecuciones en un emulador con API 27 y en un dispositivo físico con API 30.

Para llevar a cabo las mediciones se ha utilizado la herramienta *Android Profiler*, que ha permitido visualizar y registrar diferentes métricas relacionadas con el rendimiento, como el tiempo de carga, el uso de la CPU, el consumo de memoria y otros recursos. En particular, para evaluar el tiempo de carga, se ha utilizado la función de registro de eventos.

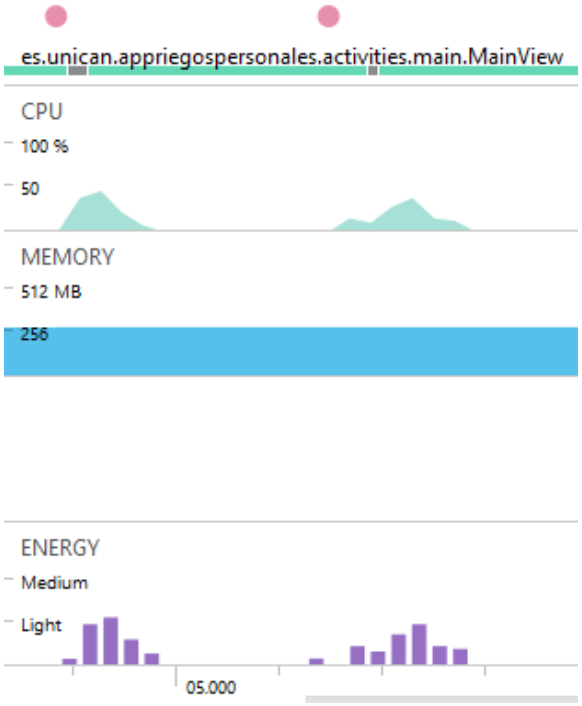


Ilustración 36. Android Profiler

Se han evaluado dos escenarios diferentes: uno con el servicio disponible, donde la aplicación se conecta a él, y otro sin el servicio disponible, lo que provoca un error y la recuperación de datos desde la base de datos local. Los resultados obtenidos mostraban una diferencia significativa: con el servicio disponible el tiempo de carga promedio es de 0,26 segundos, mientras que, sin el servicio disponible, el tiempo de carga promedio alcanza los 10,93 segundos, superando ampliamente el requisito establecido. En la siguiente tabla se observan las mediciones realizadas.

	Emulador API 27		Dispositivo API 30	
	Servicio disponible	Servicio no disponible	Servicio disponible	Servicio no disponible
1	0,3435	11,07	0,1528	11,38
2	0,4258	10,92	0,1427	10,75
3	0,3756	10,98	0,1465	10,8
4	0,356	10,97	0,1415	10,75
5	0,3828	10,83	0,1289	10,85
Media	0,37674	10,954	0,14248	10,906

Tabla 15. Mediciones tiempo de carga pestaña Apps (acceso sync)

En base a estos resultados, se tomó la decisión de realizar el acceso al servicio de forma asíncrona en lugar de síncrona, como se estaba haciendo. Esta modificación ha mejorado considerablemente el tiempo de carga en situaciones donde el servicio no está disponible, reduciéndolo a 0,12 segundos, mientras que, el acceso con el servicio disponible se mantiene en una media de 0,27 segundos.

	Emulador API 27		Dispositivo API 30	
	Servicio disponible	Servicio no disponible	Servicio disponible	Servicio no disponible
1	0,4669	0,1617	0,0895	0,1168
2	0,4276	0,1369	0,0841	0,0741
3	0,5349	0,1409	0,0805	0,0898
4	0,4436	0,1504	0,0734	0,0808
5	0,4007	0,1415	0,0769	0,0763
Media	0,45474	0,14628	0,08088	0,08756

Tabla 16. Mediciones tiempo de carga pestaña Apps (acceso async)

Esta optimización ha permitido cumplir con el requisito establecido y brindar una experiencia más fluida al usuario.

6.6.3. Pruebas de usabilidad

El primer requisito de usabilidad que debe cumplir la aplicación es adaptarse al modo oscuro de manera consistente, garantizando la legibilidad de todos los elementos visuales. Desde el inicio del desarrollo, se ha tenido presente este requisito, realizando verificaciones exhaustivas con cada nueva funcionalidad añadida para asegurar la consistencia visual en el modo oscuro. En caso de detectar alguna discrepancia, se han realizado los ajustes necesarios para corregirlo. A continuación, se presentan ejemplos de la interfaz en modo oscuro para ilustrar este aspecto.

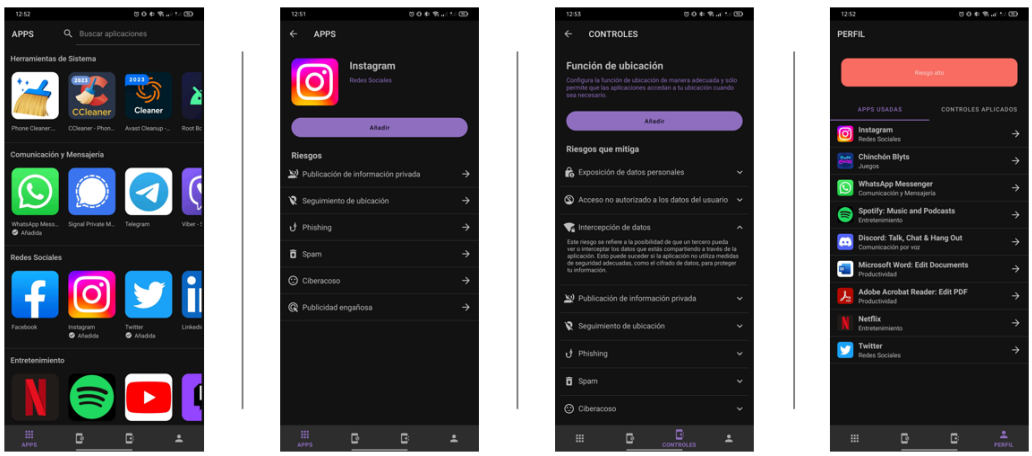


Ilustración 37. Interfaz de usuario en modo oscuro

Otro de los requisitos de usabilidad para la aplicación es buscar que la aplicación se intuitiva, fácil de usar y siga las directrices de diseño de *Android*. El objetivo es brindar una experiencia de usuario agradable y facilitar la interacción con la aplicación.

Para ello, durante el proceso de desarrollo, se ha tomado inspiración en la exitosa aplicación *Play Store*, que destaca por adherirse a las directrices de diseño de *Android*. Se han tenido en cuenta los enfoques utilizados en *Play Store* y se han aplicado principios similares para garantizar que la aplicación cumpla con los estándares de usabilidad.

Además, para garantizar la calidad de la usabilidad, las posteriores pruebas de aceptación en las que se recibe retroalimentación de diferentes usuarios han confirmado que la aplicación cumple con el requisito de facilidad de uso y ha sido bien recibida.

6.6.4. Pruebas de escalabilidad

La elección de la arquitectura MVP (Modelo-Vista-Presentador) para desarrollar la aplicación se basa en el requisito de garantizar la escalabilidad y la gestión eficiente de grandes cantidades de datos sin comprometer el rendimiento y la funcionalidad.

Al utilizar el patrón MVP, se logra una estructura modular y bien organizada, lo que facilita la escalabilidad de la aplicación. Por ejemplo, actualmente el volumen de datos que se maneja es 50 aplicaciones, 13 categorías, 13 riesgos y 15 controles, pero si se requiere manejar una mayor cantidad de datos, se puede ajustar la capa del modelo para manejar eficientemente la carga de datos adicionales sin afectar la interfaz de usuario o el flujo de la aplicación. Esto evita posibles problemas de rendimiento y asegura que la aplicación siga siendo ágil y responsiva incluso con grandes volúmenes de datos.

Además, el MVP permite una mayor reutilización de código, lo que facilita la implementación de nuevas funcionalidades y la introducción de cambios en la aplicación. Al tener una separación clara entre la lógica de negocio y la interfaz de usuario, se pueden realizar modificaciones en una capa sin afectar a las demás, lo que simplifica el mantenimiento y la evolución de la aplicación a medida que se van agregando nuevas características o se realizan mejoras.

6.7. Pruebas de aceptación de la Aplicación Android

Se ha llevado a cabo un proceso de validación a través de pruebas de aceptación con un grupo de usuarios cercanos, incluyendo familiares y amigos, con el objetivo de obtener retroalimentación directa y realista sobre la aplicación.

Dado que el servicio está desplegado en local, ha sido necesaria una ejecución previa conectando cada dispositivo al PC donde se encuentra desplegado el servicio para la descarga de datos. Durante una semana, estos usuarios han tenido la oportunidad de utilizar la aplicación en su día a día y compartir sus opiniones y sugerencias.

En este proceso de validación, se ha identificado un punto de mejora, muchos de los participantes han expresado el deseo de recibir recomendaciones más específicas y personalizadas sobre qué acciones tomar para reducir sus riesgos personales. Por ejemplo, han destacado la utilidad de contar con recomendaciones directas y personalizadas, como "realizar un control específico para maximizar los beneficios y reducir los riesgos identificados".

Sin embargo, en términos generales, la aplicación ha sido considerada aceptada por los usuarios participantes. Han valorado positivamente la funcionalidad general, destacando la capacidad de visualizar y gestionar los riesgos personales de manera intuitiva y eficiente. Además, han destacado la utilidad y relevancia de la información proporcionada, así como la facilidad de uso de la interfaz de usuario.

La retroalimentación recopilada durante las pruebas de aceptación ha sido analizada cuidadosamente y se han tomado en cuenta las sugerencias y comentarios para futuras iteraciones del producto. Este proceso de validación con usuarios reales ha permitido obtener una visión más completa de las necesidades y expectativas de los usuarios, así como identificar oportunidades de mejora para garantizar que la aplicación brinde una experiencia satisfactoria y personalizada.

7. Conclusiones

En este último apartado, se detallan las conclusiones obtenidas tras finalizar el proyecto evaluando si se han cumplido los objetivos planteados inicialmente. Además de explicar qué funcionalidades se desean añadir en un futuro.

7.1. Objetivos

Este proyecto planteaba tres objetivos secundarios para lograr crear una herramienta completa y amigable para gestionar los riesgos personales asociados al uso de aplicaciones móviles, mejorando la seguridad y protección de los usuarios en el entorno digital, los cuales considero que han sido alcanzados.

En primer lugar, se ha desarrollado con éxito un servicio REST que sirve como fuente centralizada de datos para la aplicación móvil. Este servicio almacena de manera eficiente la información sobre el catálogo de aplicaciones, los riesgos y los controles, brindando a los usuarios una base sólida para abordar los riesgos vinculados a las aplicaciones que emplean.

En segundo lugar, se ha creado una aplicación móvil para la plataforma *Android* que permite a los usuarios registrar y administrar las aplicaciones que utilizan. Además, proporciona información detallada y comprensible sobre los riesgos asociados a cada aplicación, ofreciendo una visión clara de los posibles peligros en el entorno móvil.

En último lugar, la aplicación móvil incluye información sobre los controles aplicados actualmente y aquellos que pueden aplicarse para mitigar los riesgos identificados.

En conclusión, los objetivos secundarios se consideran alcanzados y, por tanto, el objetivo principal habiéndose logrado desarrollar una herramienta funcional y útil.

7.2. Trabajo futuro

A pesar de los logros alcanzados, hay diversas funcionalidades que se pretenden añadir en futuras versiones de la aplicación con el objetivo de mejorar aún más la experiencia del usuario y proporcionar una gestión de riesgos personalizada y efectiva. Algunas de las ideas para el desarrollo futuro son las siguientes:

1. Realizar recomendaciones sobre qué controles aplicar: Se buscará desarrollar un algoritmo de recomendaciones que en base a la situación actual del usuario (riesgos actuales y controles aplicados), le sugiera qué controles serían los mejores a aplicar para bajar su nivel de riesgo. Esto permitirá una mayor satisfacción de los usuarios que quieran soluciones rápidas y razonables sin tener que pararse a mirar todos los riesgos y controles.

2. Cuestionario inicial: Se planea implementar un cuestionario amigable al inicio de la aplicación, en el cual se solicitará información relevante como la edad del usuario, las aplicaciones que utiliza con mayor frecuencia, entre otros datos pertinentes. Esta forma de recoger los datos mejorará la experiencia de usuario.

3. Conciencia del tipo de persona: Se explorará la posibilidad de incluir en el perfil del usuario opciones para indicar su grupo demográfico, como persona mayor, niño, etc. Esta información se utilizará para ampliar el análisis de riesgos, teniendo en cuenta las particularidades de cada grupo demográfico. Por ejemplo, una persona mayor puede tener un mayor riesgo de ser víctima de ataques de phishing, por lo que se brindarán recomendaciones específicas para mitigar ese riesgo.

4. Cargar aplicaciones automáticamente: Se pretende desarrollar una funcionalidad que permita a la aplicación conectarse al dispositivo móvil del usuario y cargar automáticamente las aplicaciones instaladas. Esto facilitará la gestión de aplicaciones y evitará la necesidad de ingresarlas manualmente.

5. Reporte de incidencias por parte de los usuarios: Se implementará una funcionalidad que permita a los usuarios reportar incidencias o riesgos asociados a una aplicación. Esta comunicación usuario-servidor contribuirá a una inteligencia social, donde los usuarios podrán compartir información valiosa sobre posibles riesgos. Estos reportes se subirán al servidor y los usuarios podrán ver qué aplicaciones han sido reportadas por otros usuarios. Esta función se mostrará en una pestaña separada, como "alerta social", para distinguirla del nivel de riesgo actual y fomentar una mayor conciencia de los riesgos informados por la comunidad.

Estas ideas de desarrollo futuro son solo algunas de las muchas posibilidades que se consideran para seguir mejorando la aplicación y brindar una experiencia de gestión de riesgos más completa y personalizada. El objetivo es seguir adaptándose a las necesidades y demandas de los usuarios, proporcionando herramientas efectivas y actualizadas para garantizar la seguridad y protección en el entorno digital.

Bibliografía

Radicati Team. (19 de Enero de 2021). *The Radicati Group, Inc. A Technology Market Research Firm*. Obtenido de https://www.radicati.com

Android Developers. (02 de 08 de 2021). *Android for Developers*. Obtenido de https://developer.android.com/guide/topics/ui/layout/recyclerview?hl=es-419

GreenRobot. (s.f.). *GreenRobot*. Obtenido de https://greenrobot.org/greendao/documentation/introduction/

Orange Business. (9 de Diciembre de 2021). *Orange Business*. Obtenido de https://www.orange-business.com/es/prensa/el-numero-ataques-ciberneticos-contra-organizaciones-aumenta-en-13-con-alza-notable-en-los

Vázquez, I. (22 de Marzo de 2023). *App Marketing News*. Obtenido de https://appmarketingnews.io/el-81-de-las-apps-pueden-ser-vulnerables-a-ciberataques/