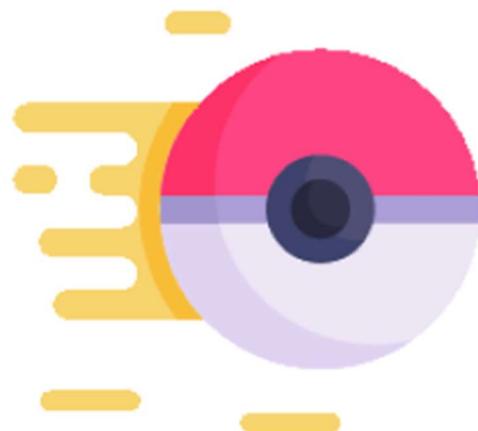


CICLO FORMATIVO DE GRADO SUPERIOR

DESARROLLO DE APLICACIONES MULTIPLATAFORMA



Pokelytics

NEREA RAMOS ESCOBAR

JOAQUÍN AYLLÓN GARCÍA

CARLOS PÉREZ MORENO

Licencia

Esta obra está bajo una licencia Reconocimiento-Compartir bajo la misma licencia 3.0 España de Creative Commons. Para ver una copia de esta licencia, visite <http://creativecommons.org/licenses/by-sa/3.0/es/> o envie una carta a Creative Commons, 171 Second Street, Suite 300, San Francisco, California 94105, USA.

ÍNDICE

ÍNDICE DE ILUSTRACIONES	1
1. RESUMEN	2
2. ABSTRACT	3
3. INTRODUCCIÓN	4
4. CONTEXTO FUNCIONAL Y TECNOLÓGICO	5
4.1 CONTEXTO FUNCIONAL.....	5
4.2 CONTEXTO TECNOLÓGICO	6
5. PLANIFICACIÓN DEL PROYECTO	7
5.1 PLANIFICACIÓN	7
5.1.1 Decisión del proyecto.	7
5.1.2 Diseño del proyecto.	7
5.1.3 Desarrollo	9
5.1.4 Pruebas	9
5.2 CONTENIDO DE LA API	10
5.3 GESTIÓN DE LOS DATOS CON FIREBASE.....	13
5.4 ESTIMACIÓN DE RECURSOS Y PLANIFICACIÓN:.....	16
6. DESCRIPCIÓN DE LA SOLUCIÓN: ANÁLISIS Y DISEÑO	17
6.1. ESPECIFICACIÓN DE REQUISITOS	23
6.1.1 Objetivos y alcance:	23
6.1.2 Requisitos funcionales:	24
6.1.3 Requisitos no funcionales:	25
6.1.4 Interfaz de usuario:.....	26
6.1.5 Requisitos de rendimiento y escalabilidad:	27
6.1.6 Requisitos de seguridad y privacidad:	28
6.1.7 Requisitos de mantenimiento y soporte:	30
6.2 SELECCIÓN DE LA PLATAFORMA TECNOLÓGICA	31
6.3 DESCRIPCIÓN DEL DISEÑO DE LA SOLUCIÓN	32
7. DESCRIPCIÓN DE LA SOLUCIÓN: CONSTRUCCIÓN	34
7.1 DOCUMENTACIÓN DESCRIPTIVA	34
7.1.1 ActivityBase:	34
7.1.2 LoginActivity	36
7.1.3 MainActivity.....	37
7.1.4 SplashActivity.....	38

7.2 ¿POR QUÉ HEMOS ELEGIDO LENGUAJES BASADOS EN C?.....	40
7.3 USO DE LA CLASE ENCRYPTEDSHARED PREFERENCES	42
7.3.1 Seguridad avanzada.....	42
7.3.2 Integración sencilla.....	42
7.3.3 Compatibilidad con versiones anteriores:	42
7.3.4 Cifrado de valores específicos:	42
7.3.5 Rendimiento aceptable:.....	43
7.3.6 Cómo guardar datos simples con SharedPreferences	43
7.3.7 Cómo controlar las preferencias compartidas.....	44
7.3.8 Cómo escribir en las preferencias compartidas.....	45
7.3.9 Cómo leer desde las preferencias compartidas.....	45
7.4 PROBLEMAS ENCONTRADOS Y SOLUCIONES	46
8. VALIDACIÓN DE LA SOLUCIÓN.....	48
8.1. DOCUMENTACIÓN DESCRIPTIVA	48
8.1.1 METODOLOGÍA TDD	48
8.1.2 MODELO PARA POKELYTICS.....	50
8.1.3 INYECCIÓN DE DEPENDENCIAS	55
8.1.4 Hilt	59
8.1.5 ¿Koin vs Dagger?.....	60
8.1.6 RETROFIT.....	62
8.1.7 CLEAN CODE: DRY, SOLID.....	68
8.1.8 Cómo guardar contenido en una base de datos local con Room	71
8.1.9 JUNIT	76
8.1.10 Empleando Mockito.....	90
8.2 PROBLEMAS ENCONTRADOS Y JUSTIFICACIÓN.....	91
9. POKELYTICS COMO PROYECTO EMPRENDEDOR	93
9.1 ANÁLISIS DE LA SITUACIÓN ACTUAL	93
9.2 NECESIDADES DEL CLIENTE Y OPORTUNIDAD DE NEGOCIO	96
10. PREVENCIÓN DE RIESGOS	99
10.1 Fatiga visual.....	99
10.2 Fatiga muscular	99
10.3 Caídas de personal o golpe contra objetos	99
10.4 Contacto eléctrico	100
10.5 Carga mental	100
10.6 Factores en la organización.....	100
Bibliografía.....	101

ÍNDICE DE ILUSTRACIONES

<i>Imagen 1 Vista general del prototipo</i>	7
<i>Imagen 2 Vista detallada del prototipo.....</i>	8
<i>Imagen 3 Vista general del tablero de Trello.....</i>	8
<i>Imagen 4 Vista MainActivity</i>	9
<i>Imagen 5 PokéApi.....</i>	11
<i>Imagen 6 Firebase Analytics.....</i>	14
<i>Imagen 7 Firebase</i>	15
<i>Imagen 8 Diagrama de Gantt.....</i>	16
<i>Imagen 9 Tipografías.....</i>	17
<i>Imagen 10 Login de Pokelytics</i>	19
<i>Imagen 11 Login final de Pokelytics</i>	20
<i>Imagen 12 Perfil de usuario.....</i>	20
<i>Imagen 13 Lista de Pokemons</i>	21
<i>Imagen 14 Fragments con información del pokemon</i>	21
<i>Imagen 15 Ejemplos de iconos</i>	22
<i>Imagen 16 Icono de Pokelytics</i>	23
<i>Imagen 17 Estructura del proyecto</i>	32
<i>Imagen 18 ActivityBase</i>	35
<i>Imagen 19 LoginActivity</i>	37
<i>Imagen 20 MainActivity</i>	38
<i>Imagen 21 SplashActivity</i>	39
<i>Imagen 22 Diagrama MVVM.....</i>	51
<i>Imagen 23 Esquema de costes de Kent Beck</i>	68
<i>Imagen 24 Funciones de Clean Code</i>	70
<i>Imagen 25 Imports</i>	70
<i>Imagen 26 Árbol de directorios</i>	70
<i>Imagen 27 Store</i>	71
<i>Imagen 28 Diagrama de la arquitectura de la biblioteca de Room.....</i>	73
<i>Imagen 29 Tests.....</i>	92
<i>Imagen 30 Estadísticas jugadores de Pokemon</i>	94
<i>Imagen 31 Estadísticas Edades Jugadores</i>	95

1. RESUMEN

Pokelytics es una aplicación móvil que se enfoca en ser una herramienta completa para todos los fans de la franquicia Pokémon. La idea principal es ofrecer una experiencia completa de una Pokédex digital, en la que se puedan buscar todos los Pokémon, además de objetos y otros elementos importantes de la franquicia.

La aplicación se ha diseñado para ofrecer una experiencia de usuario cómoda e intuitiva, que permita encontrar la información que se busca de forma rápida y sencilla. Se han implementado diferentes filtros para buscar por tipo, región, habilidades y más, lo que hace que sea fácil encontrar los Pokémon que se necesitan para completar la Pokédex.

Además de la información de los Pokémon, Pokelytics también ofrece una amplia variedad de información sobre objetos, habilidades y otros elementos de la franquicia Pokémon. Por ejemplo, se pueden buscar objetos como Poke Balls, bayas y piedras evolutivas, y se pueden conocer las habilidades de los diferentes Pokémon y cómo evolucionan.

La elección de este proyecto se debe a la gran popularidad y éxito de la franquicia Pokémon en todo el mundo. Desde su lanzamiento en 1996, la franquicia ha sido un éxito de ventas en todo el mundo y ha creado una gran comunidad de fans. La franquicia ha evolucionado con el tiempo y ha añadido nuevas generaciones de Pokémon, nuevas regiones y nuevas mecánicas de juego. Por lo tanto, los fans siempre están buscando nuevas formas de descubrir más sobre la franquicia y Pokelytics se presenta como una herramienta útil para aquellos que deseen conocer más acerca de la misma.

Además, hay una gran cantidad de información sobre la franquicia que puede ser difícil de encontrar en un solo lugar. Por ejemplo, para encontrar información sobre objetos o habilidades específicas, los fans tienen que buscar en diferentes fuentes o en diferentes juegos. Pokelytics busca simplificar todo esto y ofrecer una herramienta completa y fácil de usar para que los fans encuentren toda la información que necesitan en un solo lugar.

La aplicación también es útil para aquellos que juegan juegos de la franquicia como Pokémon GO y Pokémon Masters, ya que ofrece información sobre los Pokémon y objetos que se utilizan en estos juegos. De esta manera, los usuarios pueden estar mejor informados y tomar decisiones más acertadas a la hora de jugar.

Pokelytics en esencia, es una aplicación móvil diseñada para ofrecer una experiencia completa de una Pokédex digital, en la que se puedan buscar todos los Pokémon, objetos y otros elementos importantes de la franquicia Pokémon. La elección de este proyecto se debe a la gran popularidad y éxito de la franquicia Pokémon en todo el mundo y la necesidad de una herramienta completa y fácil de usar para que los fans encuentren toda la información que necesitan en un solo lugar. La aplicación también es útil para aquellos que juegan juegos de la franquicia, como Pokémon GO y Pokémon Masters, ya que ofrece información sobre los Pokémon y objetos que se utilizan en estos juegos. En general, Pokelytics es una herramienta útil y valiosa para cualquier fan de la franquicia Pokémon.

2. ABSTRACT

Pokelytics is a mobile application focused on being a complete tool for all fans of the Pokémon franchise. The main idea is to offer a complete experience of a digital Pokédex, in which all Pokémons, objects, and other important elements of the franchise can be searched for.

The application has been designed to offer a comfortable and intuitive user experience, which allows finding the information you are looking for quickly and easily. Different filters have been implemented to search by type, region, abilities, and more, which makes it easy to find the Pokémons needed to complete the Pokédex.

In addition to Pokémons information, Pokelytics also offers a wide variety of information about objects, abilities, and other elements of the Pokémon franchise. For example, objects such as Poke Balls, berries, and evolutionary stones can be searched for, and users can learn about the abilities of different Pokémons and how they evolve.

The choice of this project is due to the great popularity and success of the Pokémon franchise worldwide. Since its launch in 1996, the franchise has been a sales success worldwide and has created a large community of fans. The franchise has evolved over time and has added new generations of Pokémons, new regions, and new gameplay mechanics. Therefore, fans are always looking for new ways to discover more about the franchise, and Pokelytics is presented as a useful tool for those who wish to learn more about it.

In addition, there is a lot of information about the franchise that can be difficult to find in one place. For example, to find information about specific objects or abilities, fans have to search different sources or different games. Pokelytics seeks to simplify all this and offer a complete and easy-to-use tool for fans to find all the information they need in one place.

The application is also useful for those who play games from the franchise such as Pokémon GO and Pokémon Masters, as it offers information about the Pokémons and objects used in these games. This way, users can be better informed and make more informed decisions when playing.

Definitely, Pokelytics is a mobile application designed to offer a complete experience of a digital Pokédex, in which all Pokémons, objects, and other important elements of the Pokémon franchise can be searched for. The choice of this project is due to the great popularity and success of the Pokémon franchise worldwide and the need for a complete and easy-to-use tool for fans to find all the information they need in one place. The application is also useful for those who play games from the franchise such as Pokémon GO and Pokémon Masters. Overall, Pokelytics is a useful and valuable tool for any fan of the Pokémon franchise.

3. INTRODUCCIÓN

El proyecto Pokelytics App es una aplicación cuya finalidad es proporcionar información completa a cerca de Pokémon además de algunos otros elementos relacionados con ellos.

Esta franquicia posee en el mercado más de 80 videojuegos donde encontramos grandes cantidades de usuarios que continúan disfrutando de la saga y otros que están iniciándose. Éstos últimos normalmente desconocen ciertos aspectos de la lógica natural del juego tal como que un Pokémon de tipo fuego es débil enfrentándose a otro del tipo agua por cuestiones obvias. Aquí es donde entra en juego Pokelytics ya que no siempre las fortalezas y debilidades son tan evidentes entre tipos. Además, muchas veces en los videojuegos no hacemos uso de algunos elementos simplemente por desconocimiento de la utilidad de éstos. Por ello, esta app pretende hacer que los elementos más importantes que hemos de tener en cuenta en el videojuego se clarifiquen para poder exprimir todo el potencial de los Pokémon y hacer que el usuario tenga una mejor experiencia de juego gracias a la información que posee.

La información sobre Pokémon es pública y hay múltiples foros, blogs, apps, APIs, etc. Pero no siempre tenemos la facilidad, la comodidad o el atractivo que ofrece una app sencilla como Pokelytics.

Esta información la recogerá Pokelytics para hacer que los usuarios puedan consultar esta información y personalizar sus búsquedas e incluso generar una Pokédex personalizada en la que tengan almacenados los Pokémon que ya obran en su poder.

Para poder hacer un uso personalizado de la app los usuarios tendrán que registrarse con un correo electrónico y podrán personalizar algunos aspectos de su perfil que harán que la experiencia con la app se sienta más personal.

La aplicación utiliza Firebase para poder almacenar todos estos datos de carácter personal y mantenerlos seguros. El desarrollo de la aplicación se realiza con Android Studio ya que la cuota de mercado que abarca Android es mayor que la de Apple (los dos mayores del mercado).

La información general que muestra la aplicación proviene de una API bastante recurrida por la comunidad de seguidores de Pokémon como es <https://pokeapi.co/api/v2>. Aquí encontramos el archivo JSON con el que podremos implementar toda la información que queramos manejar desde las funcionalidades de nuestra aplicación.

Para realizar el control de versiones de la aplicación se ha empleado GitHub, que nos ha permitido trabajar sobre distintas versiones del proyecto para ir añadiendo cambios sin que éstos puedan comprometer en ningún momento las funcionalidades implementadas y que esto pueda desencadenar fallos en el proyecto que nos puedan perjudicar de manera drástica.

Durante la elaboración del prototipo y diseño de la aplicación hemos utilizado Figma y para la organización y gestión de tareas hemos utilizado Trello, ya que es sencillo y cómodo y permite conocer en tiempo real si alguno de los miembros del equipo está trabajando en alguna de las tareas y evitar así que realicemos el trabajo por duplicado.

4. CONTEXTO FUNCIONAL Y TECNOLÓGICO

4.1 CONTEXTO FUNCIONAL

Para comprender las funcionalidades de Pokelytics primero vamos a comenzar por comentar brevemente en qué consiste tanto Pokémon como qué es una Pokédex.

Pokémon es una franquicia de videojuegos, series de televisión, películas, juegos de cartas, juguetes y otros productos relacionados con la captura, entrenamiento y batalla de criaturas ficticias conocidas como Pokémon.

El juego se desarrolla en un mundo imaginario llamado Pokémon World, donde los entrenadores, llamados “entrenadores Pokémon”, atrapan y entrena Pokémon para competir en batallas contra otros entrenadores. Cada Pokémon tiene habilidades y características únicas, lo que los hace más efectivos en ciertas situaciones de batalla. Los entrenadores pueden evolucionar a sus Pokémon a través de la experiencia y la exposición a elementos específicos.

La Pokédex es una enciclopedia electrónica que contiene información detallada sobre cada Pokémon conocido. Fue creada por el profesor Oak, un personaje recurrente en la franquicia, para ayudar a los entrenadores a completar su colección de Pokémon. La Pokédex proporciona información sobre las habilidades, fortalezas y debilidades de cada Pokémon, así como datos sobre su evolución, comportamiento y hábitat.

En este contexto ya podemos hacer referencia a nuestro proyecto Pokelytics, basado en esa Pokédex original que contenía todos los Pokémon y que sirve como medio de consulta para sus usuarios. Sin embargo, Pokelytics va un paso más allá y pretende hacer que los usuarios puedan mejorar su entrenamiento y sus capacidades como entrenadores Pokémon ya que permite conocer y mejorar en función de los tipos de rivales que tenemos delante.

Su función principal será la de poder consultar todos los Pokémon, podremos ver a qué tipo pertenece, contra qué tipos es más fuerte y frente a qué tipos se debilita. Podremos conocer los tipos de ataques que puede aprender y su efectividad, así como algunos objetos que podremos usar sobre ellos para incrementar su potencial en ataque o en defensa. Podremos crear un área dentro de la aplicación que nos permitirá ser más selectos con nuestras consultas aplicando filtros y haciendo que nuestra búsqueda cumpla algunos requisitos.

Para ello nos ayudaremos entre otros recursos de un diseño que resulte atractivo para que la experiencia del usuario sea lo más amigable posible para que cada vez que lo necesite recurra a nuestra app para realizar sus consultas. En el diseño hemos intentado mantener una estética que sea muy similar a lo que encontramos cuando vamos a cualquiera de los juegos de la franquicia, ya que, si el usuario encuentra algo completamente distinto a lo que encuentra en cualquiera de los juegos, puede que la experiencia y la sensación de satisfacción a la hora de utilizar nuestra app no sea lo que espere y, por tanto, deje de utilizarla.

Para que todas las funcionalidades tengan una utilidad real gestionamos los datos mediante Firebase lo cual nos permite de una manera rápida, segura y eficiente realizar la gestión de los datos que almacena la aplicación de los usuarios que acceden a ella. Esto además nos abre la puerta a poder realizar integraciones con otros servicios

como por ejemplo los servicios de Google y hacer que un registro u otras tareas que se integren se puedan realizar más rápidamente, lo que hace que el usuario pueda tener una mejor experiencia de uso. De manera simultánea esto permite que se pueda personalizar la aplicación almacenando lo que el usuario desea guardar para próximas veces que acceda a la app como por ejemplo los Pokémon que tiene en favoritos.

4.2 CONTEXTO TECNOLÓGICO

Sobre el contexto tecnológico cabe destacar algunas de las herramientas que hemos utilizado para el desarrollo del proyecto:

- Figma: herramienta para el desarrollo del prototipo.
- Trello: herramienta para fijar tareas y organizar el trabajo de manera coordinada.
- GitHub: herramienta para el control de versiones.
- Android Studio: herramienta para el desarrollo de la aplicación.
- Firebase: herramienta para la gestión de los datos en los que se basa la app.
- Word: herramienta para elaborar la memoria.
- Navegadores (Chrome, Firefox): herramientas para la búsqueda de información online.
- Discord: herramienta de comunicación e intercambio de información.
- Whatsapp: herramienta directa de comunicación.

En la actualidad contamos con diversas aplicaciones en Play Store que dan un soporte similar al consumidor o cliente. Estas aplicaciones tienen múltiples descargas, algunas de ellas llegan a situarse en un volumen de más de 100 mil descargas lo que es indicativo de la alta demanda de este producto.

De manera oficial la franquicia Pokémon posee una aplicación en la AppStore de Apple, pero no en la Play Store. Además, ésta tiene un coste para todo aquel que desee descargarla. Sin embargo, prácticamente la totalidad de las aplicaciones de este tipo desarrolladas en Android no tienen ningún cargo, lo que hace que sea más accesible a los usuarios dado que no tienen que hacer un desembolso para usarla.

Estos datos hacen que nuestras opciones de descargas sean muy optimistas ya que es un producto ciertamente demandado.

Los lenguajes para el desarrollo de la aplicación empleados son en esencia Kotlin y Java. Son lenguajes compatibles entre ellos y nos permiten dar simplicidad y eficiencia al código.

Como hemos comentado en apartados anteriores, el contenido de la app emplea una API de donde obtenemos los datos de la franquicia y que nos permite tratarlos de manera masiva para mostrarlos de una forma ordenada de tal forma que el usuario no encuentre una cantidad de información sin sentido para él si no que de una forma más visual pueda gestionar la información que la API le proporciona.

El proceso de desarrollo ha requerido la realización de pruebas continuas para que la aplicación no produzca errores en su ejecución que se han realizado en Android Studio y en diversos terminales de los desarrolladores para verificar de manera continuada el correcto funcionamiento. De igual forma progresivamente se han ido realizando copias en GitHub para el control de versiones por si en algún momento se producía algún error poder volver a la última versión operativa.

5. PLANIFICACIÓN DEL PROYECTO

5.1 PLANIFICACIÓN

5.1.1 DECISIÓN DEL PROYECTO.

Para decidir el tema del proyecto dos de los integrantes del equipo (los que inicialmente lo componían), decidieron enfocar el proyecto a una industria cuyo auge cada día crece más y más como es el mundo de los videojuegos. Sin embargo, pretendíamos además que hubiera una parte nostálgica que comprendiera todo el rango evolutivo que han tenido los videojuegos desde las generaciones algo más actuales pero que además pudiera ser un proyecto en el que poder hacer uso de datos desde alguna API. En un principio una idea estaba orientada a los juegos de Super Mario, sin embargo, era probablemente bastante más extenso y no íbamos a poder abarcar todas las funcionalidades que nos hubiera gustado. Por eso finalmente la decisión fue Pokémon, que desde su nacimiento ha ido creciendo con nuevos personajes ampliando la familia sin perder la esencia, pero además englobados todos en un denominador común: son Pokémon.

Con la decisión tomada había diferentes ideas sobre como enfocar el proyecto, pero probablemente y dada la información que encontramos en internet, lo más inteligente era enfocarse en desarrollar una Pokédex que fuera útil, sencilla, práctica y ágil y que además pudiera dejar la puerta abierta a permitir futuras integraciones con otros proyectos o a ampliarse con nuevas funcionalidades en función de lo que el mercado pueda demandar.

En este punto elaboramos el anteproyecto y lo trasladamos a la tutora, que posteriormente comunicaría al equipo que tendría un miembro más el equipo.

5.1.2 DISEÑO DEL PROYECTO.

En los inicios del proyecto comenzamos buscando un nombre para nuestra app. Comenzamos con una lluvia de ideas: cada miembro propuso algunos nombres y con varios sobre la mesa conjuntamente tomamos una decisión sobre cuál era el que más nos parecía que encajaba. Pokelytics era la decisión final.

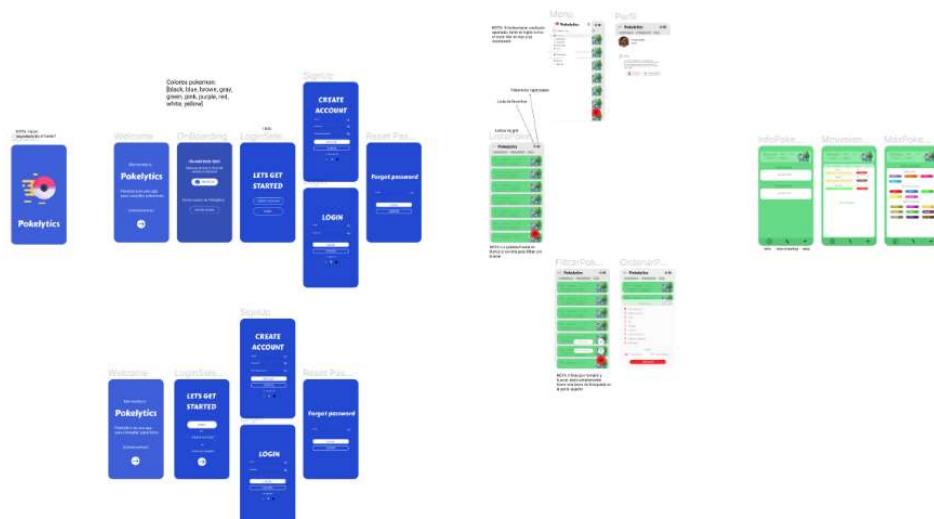


Imagen 1 Vista general del prototipo

Tras la distribución de tareas comenzamos a desarrollar el prototipo de la app con Figma. El prototipo además ha ido evolucionando también con el proyecto puesto que una vez realizadas ciertas partes y viendo cómo quedaban sobre la app el resultado no era el esperado, por ello se volvía al prototipo y se tomaban nuevas ideas, las mostrábamos en el prototipo y se llevaban a cabo las modificaciones si el resultado nos convencía a todos los integrantes.

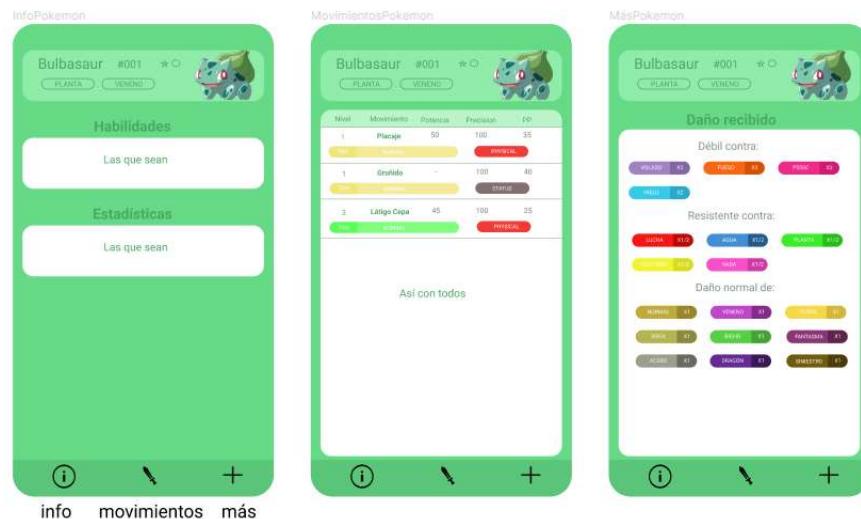


Imagen 2 Vista detallada del prototipo

Durante el prototipado se han ido anotando tareas en nuestra herramienta Trello donde podemos visualizar todo lo que está en proceso, lo que está hecho las ideas que nos surgían...

Imagen 3 Vista general del tablero de Trello

Con gran cantidad de tareas asignadas el progreso del desarrollo ha ido siendo de manera fluida en función de los tiempos que podíamos.

Para mantener esta fluidez hemos establecido diversos canales de comunicación para el equipo: un canal de Discord donde nos hemos ido comunicando de manera directa con todos los miembros vía mensaje cuando no ha sido estrictamente

necesario el tener una conversación en tiempo real y un grupo de whatsapp donde si era importante mantener la conexión en tiempo real para poder coordinarnos en caso de que fuésemos a realizar algún meeting a través de la anterior plataforma donde podíamos compartir pantalla.

5.1.3 DESARROLLO

Durante el proceso de desarrollo dado que hemos elegido cada uno qué tareas íbamos a ir desarrollando, hemos tenido la facilidad de ir realizando las tareas que mejor se nos daba a cada uno siempre apoyándonos unos a otros en el trabajo para evitar errores y en caso de dudas de igual forma nos hemos ido apoyando unos en otros.

En el desarrollo podemos destacar que el software utilizado es Android Studio y el lenguaje que hemos empleado en su mayoría es Kotlin, puesto que uno de los miembros del equipo tiene gran manejo del lenguaje y el código es más sencillo que java.

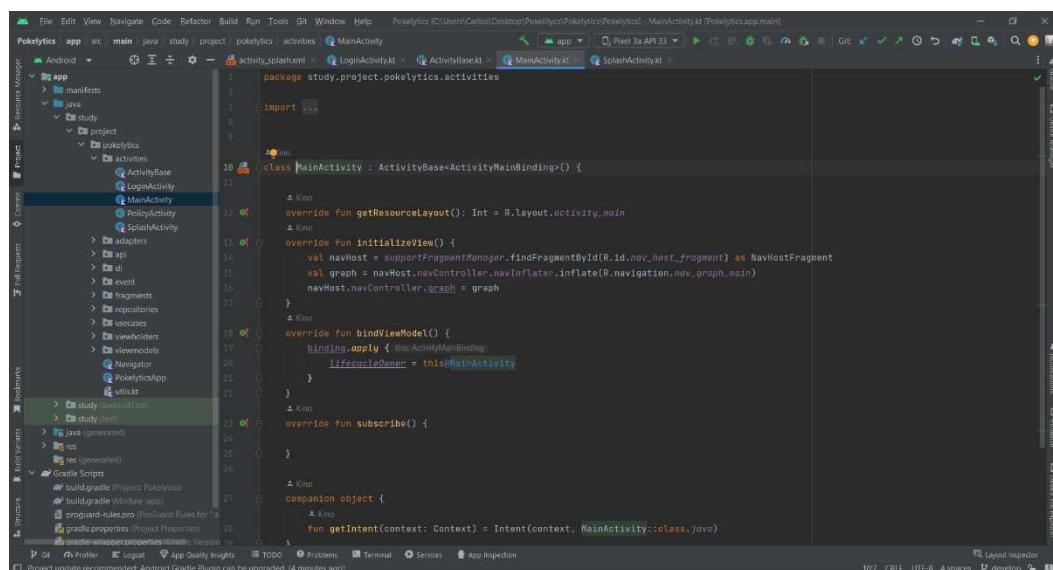
A screenshot of the Android Studio interface. The main window shows the code for 'MainActivity.kt' in the 'Activities' package. The code uses Kotlin and follows the MVVM pattern. The project structure on the left shows modules like 'app', 'study', and 'PokeLyticsApp'. The bottom status bar indicates '107 CRU, UTF-8, 4 spaces'.

Imagen 4 Vista MainActivity

Para evitar posibles problemas derivados de errores en el código gestionados de manera incorrecta hemos estado realizando de forma periódica subidas de código a la plataforma GitHub que nos ha permitido realizar desde ahí el control de versiones para, en cualquier momento donde se produjera un error, poder volver a una copia anterior en la que el código fue salvado.

5.1.4 PRUEBAS

Una vez se el desarrollo en proceso y algunas de las partes ya son funcionales hemos sometido a nuestra app a diversas pruebas. Todas estas pruebas han estado destinadas a corregir errores y mejorar el funcionamiento y la experiencia del usuario. Las pruebas las hemos realizado tanto desde los emuladores que nos ofrece Android Studio como desde los propios terminales físicos de los miembros del equipo, que además al ser todos diferentes, nos permite conocer diferentes reacciones en diferentes terminales de la app.

5.2 CONTENIDO DE LA API

La API usada en Pokelytics es una API de solo consumo, solo está disponible el método HTTP GET en los recursos.

No se requiere autenticación para acceder a esta API y todos los recursos están completamente abiertos y disponibles. Desde el cambio a un alojamiento estático en noviembre de 2018, se eliminó por completo la limitación de velocidad, pero recomiendan que se limite la frecuencia de las solicitudes para limitar los costos de alojamiento.

Política de uso justo PokéAPI es gratuita y abierta para su uso. También es muy popular. Debido a esto, piden a cada desarrollador que cumpla con su política de uso justo. Las personas que no cumplan con la política de uso justo tendrán su dirección IP permanentemente prohibida.

PokéAPI es principalmente una herramienta educativa y no toleran ataques de denegación de servicio que impidan que las personas aprendan.

REGLAS:

Cachear localmente los recursos siempre que los solicite. Sér amable y amistoso con los compañeros desarrolladores de PokéAPI. Desde la API alientan a que vayan los usuarios a Slack para contactarles por ahí antes de abrir un ticket en GitHub, para que puedan mantener las incidencias organizadas y en buen estado . También hay un sólido grupo de personas que usan la API que pueden tener respuestas o planes de experiencia.

BIBLIOTECAS DE CONTENIDO:

- Node Server-side with auto caching: Pokedex Promise v2 by Thomas Asadurian and Alessandro Pezzé
- Browser-side with auto caching: pokeapi-js-wrapper by Alessandro Pezzé
- Python 3 with auto caching: PokeBase by Greg Hilmes
- Python 2/3 with auto caching: Pokepy by Paul Hallett
- Kotlin (and Java): PokeKotlin by sargunster
- Java (Spring Boot) with auto caching: pokeapi-reactor by Benjamin Churchill
- .NET (C#, VB, etc): PokeApi.NET by PoroCYon
- .NET Standard: PokeApiNet by mtrdp642
- Swift: PokemonAPI by kinkofer
- PHP: PokePHP by Dan Rovito
- PHP: PHPokéAPI by Imerotta
- Ruby: Poke-Api-V2 by rdavid1099
- Go: pokeapi-go by mtislzr
- Crystal: pokeapi by henrikac
- Typescript with auto caching: Pokenode-ts by Gabb-c
- Rust with auto caching: Rustemon by mlemesle
- Asynchronous Python wrapper with auto caching: aiopokeapi by beastmatser
- Scala 3 with auto caching: pokeapi-scala by Juliano Alves

LISTAS DE RECURSOS/PAGINACIÓN (GRUPO)

Llamar a cualquier punto final de API sin un ID o nombre de recurso devolverá una lista paginada de recursos disponibles para esa API. Por defecto, una “página” de lista contendrá hasta 20 recursos. Si desea cambiar esto, simplemente agregue un parámetro de consulta ‘limit’ a la solicitud GET, por ejemplo, ?limit=60. Puede usar ‘offset’ para pasar a la siguiente página, por ejemplo, ?limit=60&offset=60.

IDIOMAS

La API dispone de diversos idiomas dispuestos para su selección y que hacen que la API pueda tener un mayor consumo por toda la comunidad.

La API PokeAPI es una herramienta valiosa para cualquier desarrollador interesado en crear aplicaciones o proyectos basados en el universo de Pokémon. En este texto, profundizaremos en tres aspectos clave de PokeAPI: sus características generales, sus recursos disponibles y las bibliotecas de envoltura disponibles.

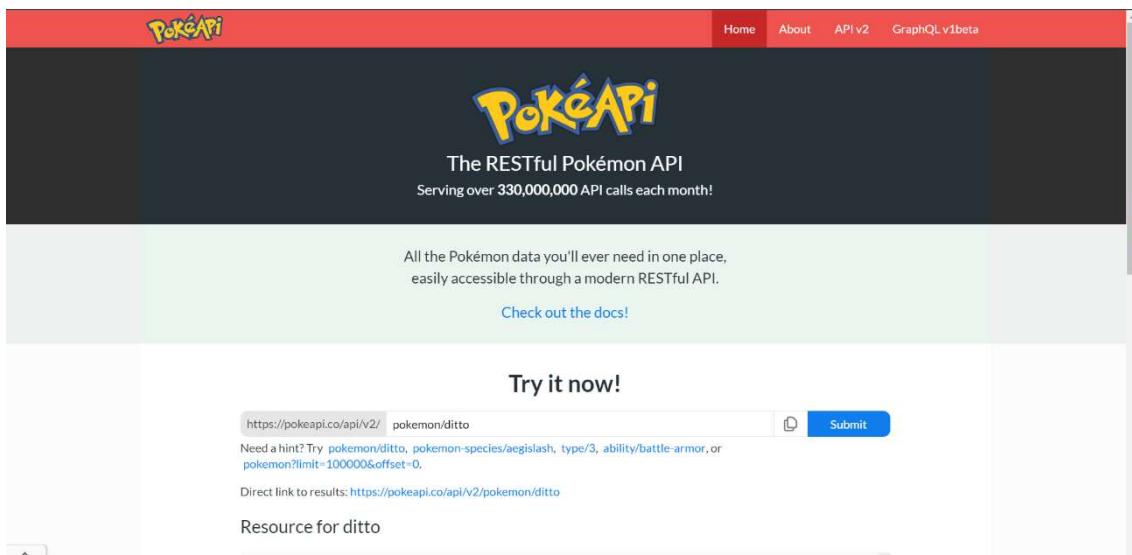


Imagen 5 PokéApi

1. Características generales

La API PokeAPI proporciona una forma fácil y eficiente de acceder a información detallada sobre los diversos aspectos del universo Pokémon. Esta API utiliza la arquitectura RESTful, lo que significa que las solicitudes se realizan utilizando los métodos HTTP GET, POST, PUT y DELETE, y los datos se devuelven en formato JSON.

Una de las características más importantes de PokeAPI es que no requiere autenticación. Esto significa que cualquier persona puede acceder a la API sin necesidad de registrarse o proporcionar credenciales. Además, PokeAPI es completamente gratuita y está abierta para su uso por cualquier desarrollador.

Sin embargo, PokeAPI tiene una política de uso justo que se aplica a todos los desarrolladores que la utilizan. Esto significa que los desarrolladores deben tener en cuenta el número de solicitudes que realizan a la API y limitar la frecuencia de las

solicitudes para evitar afectar el rendimiento de la API. PokeAPI también puede bloquear permanentemente las direcciones IP de aquellos que violen esta política.

2. Recursos disponibles

La API PokeAPI proporciona una amplia variedad de recursos que los desarrolladores pueden utilizar para crear aplicaciones y proyectos basados en el universo Pokémon. Estos recursos incluyen información detallada sobre los propios Pokémon, los movimientos, las habilidades y las estadísticas, así como los lugares y los objetos.

Para acceder a un recurso específico, los desarrolladores deben enviar una solicitud HTTP GET a la URL correspondiente. Por ejemplo, para obtener información sobre un Pokémon específico, se puede enviar una solicitud a la URL <https://pokeapi.co/api/v2/pokemon/{id or name}/>, donde “id or name” representa el ID numérico o el nombre del Pokémon deseado.

PokeAPI también ofrece recursos adicionales para ayudar a los desarrolladores a explorar y acceder a la información disponible. Por ejemplo, la API proporciona recursos para listar todos los recursos disponibles, para buscar recursos por nombre o para paginar a través de grandes conjuntos de recursos.

3. Bibliotecas de envoltura disponibles

Para hacer que el acceso a PokeAPI sea aún más fácil, muchos desarrolladores han creado bibliotecas de envoltura que se pueden utilizar con una variedad de lenguajes de programación. Estas bibliotecas se encargan de realizar solicitudes a la API, manejar los errores y devolver los datos en un formato fácilmente manejable.

Existen bibliotecas de envoltura para una variedad de lenguajes de programación, incluyendo Python, Java, Ruby, Swift y más. Algunas de estas bibliotecas son específicas de un lenguaje de programación en particular, mientras que otras son compatibles con varios lenguajes.

Las bibliotecas de envoltura son especialmente útiles para los desarrolladores que quieren acceder a PokeAPI desde aplicaciones de escritorio o móviles, o para aquellos que no quieren preocuparse por los detalles técnicos de la implementación de solicitudes HTTP.

La API PokeAPI es una herramienta valiosa para cualquier desarrollador interesado en crear aplicaciones o proyectos basados en el universo de Pokémon. Esta API ofrece características generales útiles, incluyendo el uso de arquitectura RESTful, una política de uso justo y la gratuidad y apertura para cualquier desarrollador.

Además, PokeAPI ofrece una amplia variedad de recursos que los desarrolladores pueden utilizar para acceder a información detallada sobre Pokémon, movimientos, habilidades, estadísticas, lugares y objetos. Los desarrolladores pueden acceder a estos recursos enviando solicitudes HTTP GET a la URL correspondiente.

Por último, las bibliotecas de envoltura disponibles hacen que el acceso a PokeAPI sea aún más fácil para los desarrolladores, especialmente para aquellos que quieren acceder a PokeAPI desde aplicaciones de escritorio o móviles, o para aquellos que no quieren preocuparse por los detalles técnicos de la implementación de solicitudes HTTP.

5.3 GESTIÓN DE LOS DATOS CON FIREBASE

Firebase es una plataforma en la nube propiedad de Google que se utiliza para crear aplicaciones web y móviles. Esta plataforma proporciona herramientas para el desarrollo y la implementación de aplicaciones, así como para el análisis y la mejora de la experiencia del usuario. En este texto, exploraremos el funcionamiento de Firebase y cómo gestiona los datos almacenados en su plataforma.

1. Funcionamiento de Firebase

Firebase es una plataforma que consta de varios servicios que se pueden utilizar para desarrollar aplicaciones web y móviles. Estos servicios incluyen:

- Autenticación: Firebase proporciona una solución completa para la autenticación de usuarios en aplicaciones web y móviles. Los desarrolladores pueden utilizar esta herramienta para autenticar a los usuarios mediante correo electrónico y contraseña, autenticación social con servicios como Facebook y Google, autenticación con proveedores de identidad empresarial y autenticación de teléfonos móviles.
- Almacenamiento en la nube: Firebase también ofrece una solución de almacenamiento en la nube para aplicaciones web y móviles. Los desarrolladores pueden utilizar esta herramienta para almacenar y recuperar archivos como imágenes, videos y otros datos multimedia en Firebase Storage.
- Bases de datos en tiempo real: Firebase proporciona una base de datos en tiempo real que permite a los desarrolladores almacenar y sincronizar datos en tiempo real entre los clientes y el servidor. La base de datos en tiempo real de Firebase utiliza una arquitectura de datos JSON, lo que permite una integración fácil y flexible con aplicaciones web y móviles.
- Hosting web: Firebase también ofrece un servicio de alojamiento web para aplicaciones web estáticas y dinámicas. Los desarrolladores pueden utilizar esta herramienta para alojar su sitio web y para implementar y actualizar fácilmente su contenido.
- Notificaciones push: Firebase proporciona una solución de notificaciones push que permite a los desarrolladores enviar notificaciones push a los usuarios de sus aplicaciones. Los desarrolladores pueden personalizar las notificaciones para que se ajusten a las necesidades de sus usuarios y enviarlas a usuarios específicos o a todos los usuarios de su aplicación.
- Analítica: Firebase también ofrece herramientas de análisis para que los desarrolladores puedan comprender mejor cómo los usuarios interactúan con sus aplicaciones. Con Firebase Analytics, los desarrolladores pueden obtener información sobre el uso de la aplicación, el comportamiento de los usuarios y los eventos que ocurren en la aplicación.

2. Gestión de datos en Firebase

Firebase utiliza una arquitectura de datos JSON, lo que permite una integración fácil y flexible con aplicaciones web y móviles. En Firebase, los datos se almacenan en una base de datos en tiempo real que se sincroniza en tiempo real entre los clientes y el servidor. Los datos se organizan en un árbol de datos, que consta de nodos y hojas.

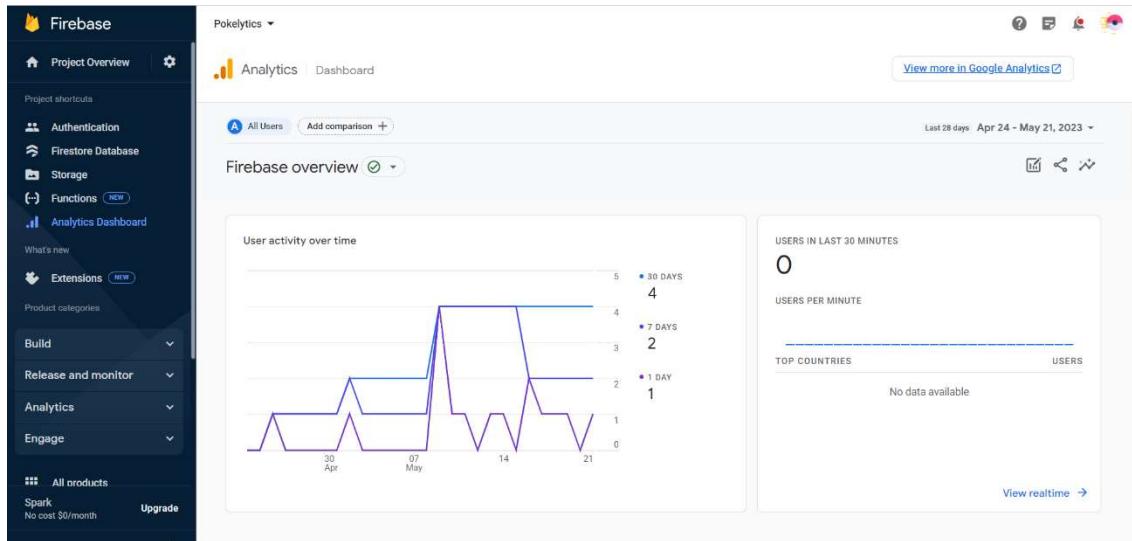


Imagen 6 Firebase Analytics

Los nodos son objetos que contienen otros objetos, y cada nodo puede tener varios hijos. Los nodos se identifican mediante una clave única y se pueden crear, leer, actualizar y eliminar mediante el uso de Firebase Database API. Las hojas son objetos que contienen datos y no tienen hijos. Las hojas se identifican mediante una clave única y se pueden leer y actualizar mediante la API de Firebase Database.

Firebase también proporciona herramientas para la autenticación de usuarios y la gestión de permisos de acceso a los datos almacenados en Firebase Database. Los desarrolladores pueden utilizar Firebase Authentication para autenticar a los usuarios y Firebase Security Rules para definir reglas de seguridad que permitan o denieguen el acceso a los datos según los roles de usuario y las acciones permitidas.

Además, Firebase ofrece herramientas para hacer copias de seguridad y restaurar los datos almacenados en Firebase Database. Los desarrolladores pueden hacer copias de seguridad de sus datos mediante la descarga de una copia de la base de datos y guardarla en un lugar seguro. Firebase también proporciona una herramienta de importación y exportación que permite a los desarrolladores mover datos entre aplicaciones y proyectos de Firebase.

Firebase también se integra con otras herramientas y servicios de Google Cloud Platform, como BigQuery y Cloud Storage, lo que permite a los desarrolladores escalar sus aplicaciones y gestionar grandes volúmenes de datos. BigQuery es una herramienta de análisis de datos que permite a los desarrolladores realizar consultas y análisis en grandes conjuntos de datos. Cloud Storage es un servicio de almacenamiento en la nube que permite a los desarrolladores almacenar y recuperar grandes volúmenes de datos.

3. Ventajas de utilizar Firebase para la gestión de datos

Firebase ofrece varias ventajas para la gestión de datos en aplicaciones web y móviles. Algunas de las ventajas más notables son:

- **Facilidad de uso:** Firebase es fácil de utilizar, lo que permite a los desarrolladores concentrarse en el desarrollo de la aplicación en lugar de en la gestión de la infraestructura. Los desarrolladores pueden utilizar las herramientas y servicios

de Firebase para desarrollar aplicaciones rápidamente y sin preocuparse por la escalabilidad y la gestión de los datos.

- **Flexibilidad:** Firebase es flexible y se adapta a diferentes necesidades de desarrollo. Los desarrolladores pueden utilizar las herramientas y servicios de Firebase según sus necesidades específicas, lo que les permite crear aplicaciones personalizadas y únicas.
- **Integración:** Firebase se integra con otras herramientas y servicios de Google Cloud Platform, lo que permite a los desarrolladores escalar sus aplicaciones y gestionar grandes volúmenes de datos. Firebase también se integra con otras herramientas de desarrollo, como React y Angular, lo que permite a los desarrolladores utilizar su herramienta de desarrollo favorita.
- **Seguridad:** Firebase ofrece herramientas para la autenticación de usuarios y la gestión de permisos de acceso a los datos almacenados en Firebase Database. Los desarrolladores pueden utilizar estas herramientas para garantizar la seguridad de los datos de sus usuarios.

Firebase es una plataforma de desarrollo de aplicaciones móviles y web propiedad de Google que ofrece una amplia gama de herramientas y servicios para ayudar a los desarrolladores a construir y escalar aplicaciones con facilidad. Firebase utiliza una arquitectura de datos JSON que permite una integración fácil y flexible con aplicaciones web y móviles. Firebase también ofrece herramientas para la autenticación de usuarios y la gestión de permisos de acceso a los datos almacenados en Firebase Database. Firebase es fácil de utilizar, flexible, se integra con otras herramientas y servicios de Google Cloud Platform y ofrece herramientas de seguridad para garantizar la seguridad de los datos de los usuarios.

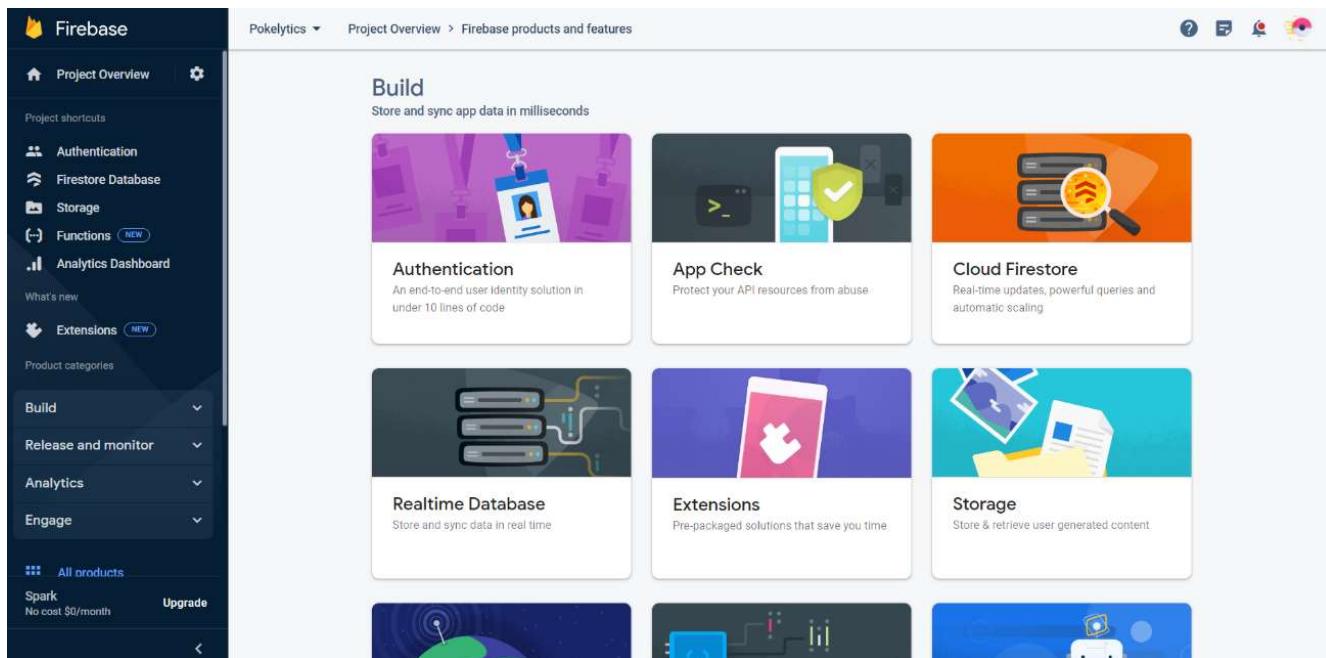


Imagen 7 Firebase

5.4 ESTIMACIÓN DE RECURSOS Y PLANIFICACIÓN:

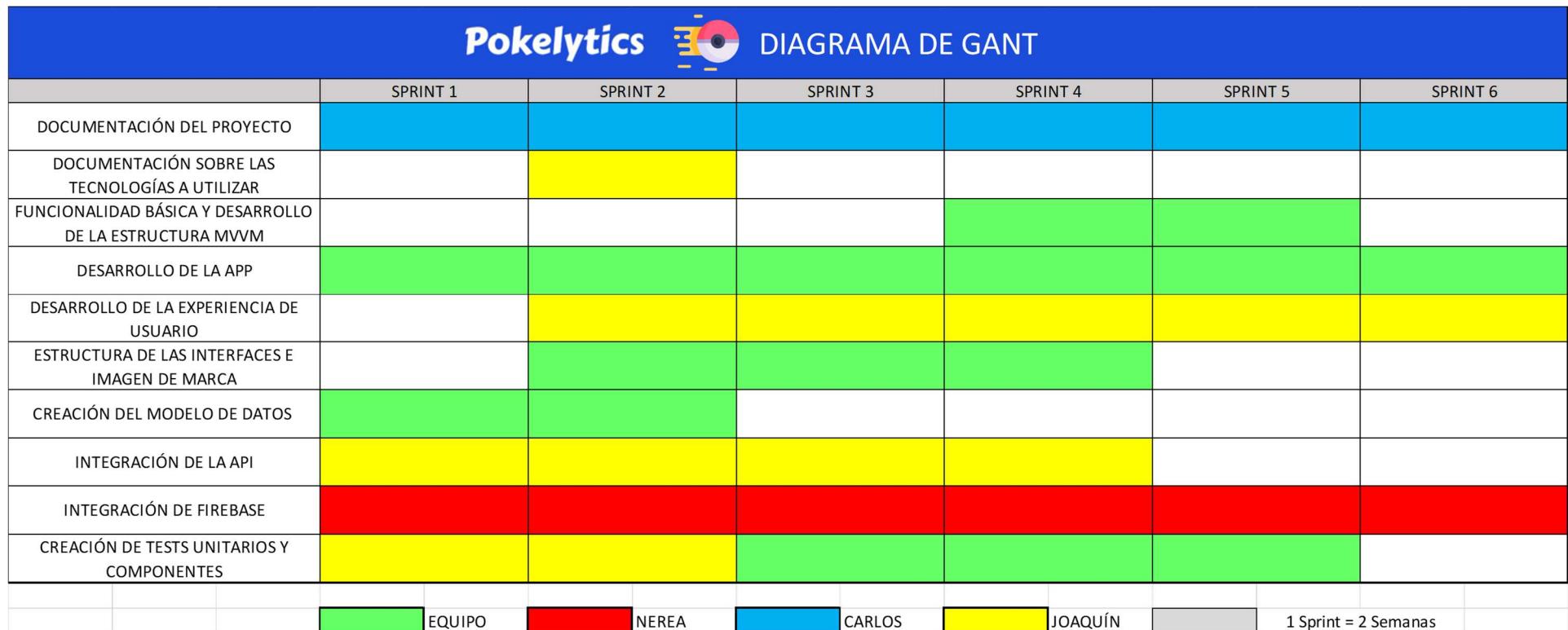


Imagen 8 Diagrama de Gantt

6. DESCRIPCIÓN DE LA SOLUCIÓN: ANÁLISIS Y DISEÑO

Antes de entrar en materia con en diseño de la app hay algunos puntos a tener en cuenta en el análisis que hemos realizado para llevar a cabo la app. Primero hemos comenzado viendo como funcionaban otras apps del mismo tipo y nos hemos fijado cuáles eran sus puntos fuertes para tomarlos como referencia y sus puntos débiles para tomarlos como referencia también y tratar de mejorarlo en la medida de lo posible.

Inspeccionamos por encima la API para ver la viabilidad a la hora de extraer datos. Vimos que estaba bastante bien estructurada y que ofrecía múltiples posibilidades a la hora de extraer información útil para la app. Por ello con toda la información y la perspectiva del proyecto prácticamente encarrilada comenzamos con el diseño.

Con respecto al diseño había varios puntos a tener en cuenta: Pokémon es una saga actual pero que no deja de ser clásica en el mundo de los videojuegos. Por lo que el logo decidimos que tuviera ese toque anticuado para poder homenajear de alguna forma al Pokémon original que no tenía una calidad HD. La pokéball era un elemento muy representativo que le iba bien a la hora de jugar con los colores, las formas y las posibles animaciones, ya que todo jugador de la saga recuerda que estas pequeñas esferas tenían colores, movimientos y acciones muy característicos que todos reconoceríamos.



Imagen 9 Tipografías

Así pues, ya teníamos el diseño de lo que sería el logo de la app que aparecería sin duda en el Splash, o pantalla de inicio de la aplicación. Teníamos que definir ciertos colores que fuesen parte de la paleta que utilizaríamos en la aplicación para los fondos, los textos, ventanas, etc. Para la paleta de colores realmente era sencillo, pues cualquiera de los colores le iría bien. Todas las ediciones de los inicios de la saga tenían asignado un color: Pokémon rojo, Pokémon amarillo, Pokémon azul... estas ediciones hacían que se asociara a un Pokémon principal por lo que cualquiera de estos colores de inicio de la saga habría sido igual de representativo. Sin embargo, nosotros empleamos un tono de azul que nos daba cierto contraste con los colores blanco y rojo de la pokéball y con los colores que elegimos para los textos (color blanco), ayudaba a que destacase.

Todas las compañías habitualmente hacen del nombre de su empresa un símbolo. Cuando pensamos en Netflix se nos viene a la cabeza su tipografía y sus colores, cuando pensamos en la 20thCentury pensamos en su tipografía sus colores e incluso su melodía. Esto es algo que hemos intentado hacer en Pokelytics, ya teníamos los colores, el logo y ahora teníamos que definir la tipografía. Y qué mejor tipografía para hacer referencia e identificar sobre qué es nuestra app, que utilizar la misma que la franquicia Pokémon.

Con todo esto ya definido podemos comenzar a trabajar en la parte del diseño de la app, es decir, como se va a ver todo ello en la aplicación final y que aspecto visual va a experimentar el usuario desde su terminal.

Para comenzar en ese diseño nos decidimos por la herramienta Figma. Figma es una herramienta de diseño colaborativo basada en la nube que ha ganado popularidad rápidamente en la comunidad de diseño. Con su enfoque en la colaboración en tiempo real y su conjunto de características robustas, Figma ha revolucionado la forma en que los equipos trabajan juntos en el diseño de interfaces de usuario y experiencias de usuario.

Uno de los puntos fuertes más destacados de Figma es su capacidad para permitir la colaboración en tiempo real. Múltiples miembros del equipo pueden trabajar simultáneamente en un proyecto, lo que facilita la comunicación y la toma de decisiones conjuntas. Con la capacidad de ver las actualizaciones en tiempo real, los diseñadores pueden ofrecer retroalimentación instantánea y realizar ajustes sobre la marcha, lo que ahorra tiempo y mejora la eficiencia del equipo.

Además, Figma se destaca por ser una herramienta basada en la nube, lo que significa que no requiere instalación y se puede acceder desde cualquier lugar con conexión a Internet. Esto elimina las barreras de compatibilidad y facilita el acceso a proyectos y archivos en cualquier momento y desde cualquier dispositivo. La capacidad de Figma para funcionar en diferentes sistemas operativos, como Windows, macOS y Linux, también lo hace altamente accesible para equipos con diversas configuraciones.

La versatilidad de Figma es otro punto fuerte que lo distingue de otras herramientas de diseño. No solo permite crear diseños de interfaces de usuario estáticos, sino que también ofrece la posibilidad de diseñar prototipos interactivos y animaciones. Esto permite a los diseñadores presentar y probar sus ideas de manera más efectiva, ya sea para revisar el flujo de usuario, demostrar interacciones o presentar animaciones de micro interacciones.

La capacidad de Figma para facilitar el diseño de sistemas y componentes reutilizables es también una de sus características más valiosas. Con la función de bibliotecas de componentes, los equipos pueden crear elementos de diseño reutilizables y mantener una consistencia visual en todos los proyectos. Esto ahorra tiempo y esfuerzo, especialmente en proyectos más grandes o en equipos con flujos de trabajo ágiles.

Otro aspecto destacado de Figma es su enfoque en la integración con otras herramientas y servicios populares. Con complementos y API personalizadas, Figma se puede conectar con aplicaciones de terceros, como Slack, Trello y Jira, lo que permite una mayor automatización y flujo de trabajo fluido entre diferentes herramientas utilizadas por los equipos de diseño.

Figma es una herramienta de diseño colaborativo basada en la nube que ofrece una amplia gama de características y beneficios para los equipos de diseño. Desde su capacidad para colaborar en tiempo real hasta su versatilidad para crear prototipos interactivos y animaciones, Figma ha demostrado ser una herramienta indispensable para diseñadores de todo el mundo. Su enfoque en la reutilización de componentes y la integración con otras herramientas también lo convierte en una opción atractiva para aquellos que buscan una solución completa para el diseño de interfaces de usuario y experiencias de usuario.



Imagen 10 Login de Pokelytics

Durante el proceso de diseño se han producido diversos cambios. Algunos los hemos conservado dentro del propio Figma y otros han sido borrados. En ésta página podemos observar el diseño original que en un principio iban a tener algunas pantallas que aparecerían tras el Splash, sin embargo observamos que quizá había demasiadas y para el usuario probablemente podría reportarle una experiencia un tanto tediosa así que eliminamos algunas de éstas. Así quedó el resultado finalmente del diseño:

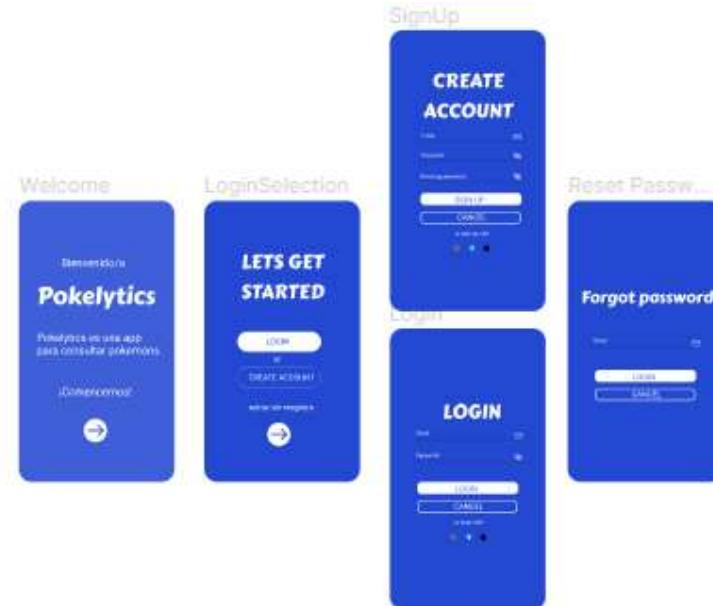


Imagen 11 Login final de Pokelytics

Eliminando algunas de las ventanas podíamos conseguir un resultado más sencillo y una experiencia más atractiva para el usuario. Además, una vez registrado el usuario ni siquiera tiene que pasar por éstas por lo que se simplifica y agiliza más aún el acceso a la aplicación.

Una vez dentro de la aplicación cuando comenzamos a obtener los datos de la api accedemos a una pantalla que tiene diferentes utilidades como se muestra en la imagen. Dentro de esta pantalla algunas utilidades quedan restringidas a aquellos usuarios que se han registrado y que, a priori, van a dejar instalada la aplicación en su terminal.

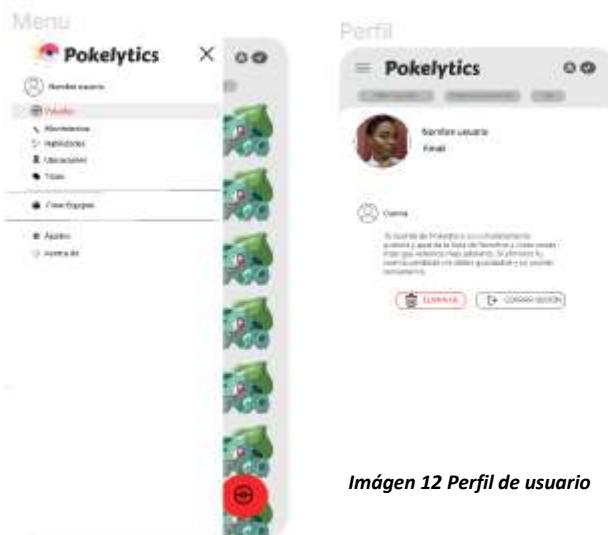


Imagen 12 Perfil de usuario



NOTA: La pokeball sería en blanco y serviría para filtrar y/o buscar

Dentro de la aplicación cada Pokémon tiene un color asignado en función del tipo de Pokémon que es, por ejemplo, los Pokémon de tipo agua, tienen un color azul, los de tipo fuego, color rojo, etc. por lo que dentro de la Pokédex tenemos una paleta de colores muy variada que refleja el tipo del Pokémon.

Esto también se aplica a la hora de acceder en detalle a los Pokémon donde viene la información detallada sobre éste. Aquí los colores también juegan un papel muy importante ya que proporcionan información visual muy rápida y a la vez hace que tenga un atractivo que llama la atención al usuario.

Imagen 13 Lista de Pokemons

En la imagen inferior podemos ver un menú inferior con algunos símbolos que nos muestran al hacer tap sobre ellos información, movimientos o más info. La imagen nos muestra cómo se verán una vez accedamos a cada una de las opciones.



Imagen 14 Fragments con información del pokémon

Dentro de estas pantallas encontramos siempre visible el menú de navegación donde tenemos los tres iconos que son representativos de su funcionalidad. Éstos los hemos obtenido de una página muy conocida como es flaticon.es. Flaticon.es es un sitio web que ofrece una amplia variedad de recursos gráficos, especialmente iconos, para su uso en proyectos de diseño. Con una extensa biblioteca de más de 10 millones de iconos de alta calidad, Flaticon.es se ha convertido en una fuente confiable y popular para diseñadores de todo el mundo.

Una de las características destacadas de Flaticon.es es su colección diversa y en constante crecimiento de iconos. Los usuarios pueden encontrar iconos en una amplia gama de estilos, temáticas y formatos, lo que les brinda la flexibilidad de elegir el recurso adecuado para sus proyectos. Además, la plataforma permite a los usuarios personalizar los iconos según sus necesidades, como cambiar el color, tamaño y dirección, lo que añade un toque de personalización a los diseños.

Otra característica importante de Flaticon.es es la posibilidad de descargar los iconos en diferentes formatos, como PNG, SVG y EPS, lo que facilita su adaptación a diversas aplicaciones y software de diseño. Los iconos descargados también están disponibles en diferentes tamaños, lo que permite una integración sin problemas en proyectos de diferentes dimensiones.

La función de búsqueda avanzada de Flaticon.es es otra característica destacada. Los usuarios pueden buscar iconos por palabras clave, filtros de estilo, categorías y colecciones, lo que agiliza el proceso de encontrar el recurso adecuado. Además, los usuarios también pueden guardar y organizar sus iconos favoritos en colecciones personalizadas para un acceso rápido y conveniente en el futuro.

Flaticon.es también ofrece una opción de suscripción premium que brinda beneficios adicionales a los usuarios, como acceso a iconos premium exclusivos, descargas ilimitadas y licencias comerciales para uso comercial. Esta opción premium es ideal para aquellos que necesitan una mayor variedad de iconos y recursos gráficos para sus proyectos profesionales.

De ésta popular web obtenemos algunos de los iconos que empleamos como recursos por ejemplo para la barra de navegación entre Fragments que contiene la app.



Imagen 15 Ejemplos de iconos

Para finalizar en lo referente a las partes más importantes del diseño de la app hemos de destacar el ícono que consiste en la pokéball que también empleamos en el splash, ya que es una parte muy representativa de la saga y los colores que emplea funcionan muy bien con los elegidos por el equipo de desarrollo.

La pokeball, con sus colores icónicos de blanco, rojo y negro, es reconocible al instante por los fanáticos de Pokémon en todo el mundo. Estos colores clásicos representan la esencia misma de la pokeball y se han convertido en un símbolo reconocido en la cultura popular.

El diseño del logo con la pokeball y la estela amarilla es una combinación efectiva de colores que representa la esencia de Pokémon. Los colores blanco, rojo y negro de la pokeball evocan diferentes aspectos de la experiencia Pokémon, mientras que el amarillo de la estela añade un toque de energía y emoción. En conjunto, este diseño captura la esencia de la franquicia y seguramente resonará con los aficionados de todas las edades.



Imagen 16 Icono de Pokelytics

6.1. ESPECIFICACIÓN DE REQUISITOS

6.1.1 OBJETIVOS Y ALCANCE:

El objetivo principal de este proyecto es desarrollar una aplicación móvil que funcione como una Pokédex, proporcionando a los usuarios una herramienta completa y fácil de usar para explorar y obtener información sobre los diferentes Pokémon de la popular franquicia.

Los principales objetivos de la aplicación Pokédex son los siguientes:

- a) Proporcionar información detallada: La aplicación debe permitir a los usuarios acceder a información completa y precisa sobre los Pokémon, incluyendo datos como nombres, tipos, descripciones, habilidades, movimientos, evoluciones y más.
- b) Funcionalidad de búsqueda y filtrado: La aplicación debe permitir a los usuarios buscar y filtrar Pokémon según criterios específicos, como nombre, tipo, habilidades, evolución, generación, región, entre otros.
- c) Visualización de detalles: La aplicación debe permitir a los usuarios ver información detallada sobre cada Pokémon seleccionado, incluyendo descripciones, movimientos, imágenes, evoluciones y otras características relevantes.

- d) Interfaz intuitiva y amigable: La aplicación debe tener una interfaz de usuario intuitiva y fácil de usar, con una navegación clara y accesible. Debe ser atractiva visualmente y proporcionar una experiencia de usuario agradable.
- e) Integración multimedia: La aplicación debe ser capaz de mostrar imágenes y otros contenidos multimedia relacionados con cada Pokémon, lo que incluye ilustraciones y/o sprites, entre otros.

El alcance del proyecto se limitará a la creación de la aplicación de la Pokédex en sí, con la funcionalidad básica mencionada anteriormente. No se incluirán características adicionales, como la capacidad de capturar Pokémon, batallas o intercambios entre usuarios.

Es importante tener en cuenta que la aplicación Pokédex estará diseñada para funcionar en dispositivos móviles Android.

La especificación de requisitos establece los cimientos para el desarrollo de la aplicación Pokelytics, definiendo claramente los objetivos y el alcance del proyecto. Esto permite al equipo de garantizar la entrega de una aplicación de calidad que cumpla con las expectativas de los usuarios.

6.1.2 REQUISITOS FUNCIONALES:

En el desarrollo de la aplicación, es importante definir los requisitos funcionales, que son las funcionalidades y características específicas que la aplicación debe incluir. A continuación, se detallan algunos requisitos funcionales clave para la aplicación:

- a) Búsqueda de Pokémon: La aplicación debe permitir a los usuarios buscar Pokémon utilizando diferentes criterios, como el nombre del Pokémon, su tipo, habilidades o número de Pokédex. La búsqueda debe ser rápida y precisa, mostrando resultados relevantes.
- b) Filtrado de Pokémon: Los usuarios deben poder filtrar los Pokémon según diferentes atributos, como su tipo, generación, región o estado de evolución. Esto permitirá a los usuarios encontrar Pokémon específicos o explorar diferentes grupos de Pokémon según sus preferencias.
- c) Visualización de detalles del Pokémon: Cuando los usuarios seleccionen un Pokémon, la aplicación debe mostrar información detallada sobre él, como su tipo, número, habilidades, movimientos aprendidos y estadísticas. Esta información debe presentarse de manera clara y fácil de leer.
- d) Favoritos y lista de capturados: La aplicación debe permitir a los usuarios marcar Pokémon como favoritos o agregarlos a una lista de Pokémon capturados. Esto les permitirá llevar un seguimiento de los Pokémon que les interesan o que han capturado en sus aventuras.

- e) Compartir Pokémon: Los usuarios deben tener la opción de compartir información sobre Pokémon específicos a través de diferentes canales, como redes sociales, correo electrónico o mensajes instantáneos. Esto les permitirá compartir descubrimientos interesantes o intercambiar información con otros jugadores. (Esta funcionalidad se implementaría para próximas versiones de la aplicación ya que es algo complejo que no ha sido posible desarrollarlo en un primer periodo de desarrollo).
- f) Actualizaciones y nuevas generaciones: La aplicación debe ser capaz de recibir actualizaciones periódicas para incluir nuevos Pokémon y características relacionadas con las nuevas generaciones de juegos de Pokémon. Esto garantizará que la aplicación esté actualizada y al día con los últimos lanzamientos de la franquicia.
- g) Idiomas y localización: La aplicación debe admitir diferentes idiomas, lo que permitirá a los usuarios elegir su idioma preferido para la interfaz y el contenido.
- h) Historial de búsqueda y navegación: La aplicación debe proporcionar un historial de búsqueda y navegación, permitiendo a los usuarios acceder rápidamente a los Pokémon que han buscado anteriormente o las páginas que han visitado recientemente. Esto mejorará la usabilidad y facilitará la navegación dentro de la aplicación. (Este punto es otra posible mejora prevista ya que de no ser posible su implementación en tiempo habría que dejarlo como una posible mejora futura).

6.1.3 REQUISITOS NO FUNCIONALES:

Además de los requisitos funcionales que describen las funcionalidades específicas de la aplicación, también es fundamental considerar los requisitos no funcionales, que son aquellos aspectos que no están directamente relacionados con las funciones, pero que son igualmente importantes para el buen desempeño y la experiencia del usuario. A continuación, se detallan algunos requisitos no funcionales relevantes para la aplicación:

- a) Rendimiento: La aplicación de la Pokédex debe ser ágil y responder de manera rápida a las interacciones del usuario. Esto implica un tiempo de carga mínimo para mostrar la información de los Pokémon, una navegación fluida entre las diferentes pantallas y una búsqueda y filtrado eficientes, incluso cuando la aplicación maneje una gran cantidad de datos.
- b) Usabilidad: La interfaz de usuario de la aplicación debe ser intuitiva y fácil de usar, incluso para aquellos usuarios que no estén familiarizados con la franquicia Pokémon. Los elementos de la interfaz deben estar organizados de manera lógica y coherente, y la navegación entre las diferentes secciones de la aplicación debe ser clara y accesible.
- c) Seguridad: La aplicación debe garantizar la seguridad y la protección de los datos de los usuarios. Esto implica la implementación de medidas de seguridad y la protección contra posibles vulnerabilidades, para prevenir cualquier acceso no autorizado o robo de información personal.

- d) Compatibilidad: La aplicación de la Pokédex debe ser compatible con diferentes dispositivos. Debe adaptarse a diferentes tamaños de pantalla, resoluciones y capacidades de hardware, asegurando que los usuarios puedan acceder y disfrutar de la aplicación sin importar el dispositivo que utilicen.
- e) Accesibilidad: La aplicación debe ser accesible para usuarios con diferentes necesidades y capacidades. Esto implica garantizar que la aplicación sea compatible con lectores de pantalla, brindar opciones de contraste y tamaño de fuente ajustables, y cumplir con los estándares de accesibilidad establecidos para permitir el acceso equitativo a todos los usuarios.
- f) Mantenibilidad y escalabilidad: La aplicación debe estar diseñada de manera que sea fácil de mantener y actualizar en el futuro. Esto implica seguir buenas prácticas de desarrollo de software, documentar adecuadamente el código y la arquitectura, y utilizar tecnologías escalables que permitan agregar nuevas funcionalidades y adaptarse a futuras demandas y cambios en el entorno tecnológico.
- g) Tiempo de respuesta: La aplicación debe proporcionar respuestas rápidas y confiables a las solicitudes de los usuarios, evitando demoras excesivas en la carga de datos o en la ejecución de acciones. Un tiempo de respuesta óptimo contribuirá a una experiencia de usuario fluida y satisfactoria.

6.1.4 INTERFAZ DE USUARIO:

El diseño de la interfaz de usuario (UI) de la aplicación es esencial para proporcionar a los usuarios una experiencia visualmente atractiva, intuitiva y fácil de usar. A continuación, se detallan aspectos clave que deben considerarse al definir los requisitos de la interfaz de usuario:

- a) Diseño visual: El diseño visual de la aplicación debe ser coherente con la temática y el estilo de la franquicia Pokémon. Se pueden utilizar colores, fuentes y elementos gráficos que reflejen el mundo Pokémon y generen una experiencia inmersiva para los usuarios.
- b) Estructura de la información: La información sobre los Pokémon debe presentarse de manera clara y organizada. La interfaz debe facilitar la identificación y comprensión de la información relevante, utilizando jerarquías visuales, agrupamiento lógico y disposición adecuada de elementos.
- c) Navegación intuitiva: La aplicación debe contar con una navegación intuitiva que permita a los usuarios moverse de manera fluida entre las diferentes secciones y funcionalidades. Se utilizan menús desplegables, barras de navegación, botones de retorno y otras técnicas de navegación reconocibles.

- d) Iconografía y elementos gráficos: Se utilizan iconos y elementos gráficos relacionados con Pokémon para mejorar la comprensión y la identificación de diferentes acciones y funciones dentro de la aplicación. Esto ayudará a los usuarios a reconocer rápidamente las opciones disponibles y facilitará la interacción.
- e) Diseño responsive: La interfaz debe adaptarse a diferentes tamaños de pantalla y dispositivos, garantizando una experiencia consistente y fácil de usar en smartphones, tabletas y otros dispositivos. Se deben tener en cuenta las mejores prácticas de diseño responsive para asegurar que la interfaz se ajuste adecuadamente a cada dispositivo.
- f) Retroalimentación visual: La aplicación debe proporcionar retroalimentación visual clara cuando los usuarios realicen acciones o interactúen con elementos de la interfaz. Esto incluye efectos de animación, cambios de color, transiciones suaves, entre otros, que indiquen visualmente que se ha realizado una acción y ayuden a los usuarios a comprender el estado de la aplicación.
- g) Experiencia del usuario: La interfaz debe diseñarse pensando en la experiencia del usuario (UX). Esto implica que la aplicación sea fácil de aprender, que las acciones sean intuitivas, que los errores sean claros y comprensibles, y que se minimicen los pasos y la complejidad para realizar tareas comunes.
- h) Feedback y ayuda contextual: La aplicación puede ofrecer feedback y ayuda contextual para guiar a los usuarios y proporcionar información adicional cuando sea necesario. Esto puede incluir mensajes informativos, sugerencias de uso, instrucciones breves y enlaces a recursos de ayuda.

Al definir los requisitos de la interfaz de usuario, es importante considerar las expectativas y preferencias de los usuarios objetivo, así como las mejores prácticas de diseño de interfaces y las directrices de la plataforma en la que se desarrollará la aplicación. Una interfaz de usuario bien diseñada mejorará la usabilidad, la satisfacción del usuario y la adopción de la app.

6.1.5 REQUISITOS DE RENDIMIENTO Y ESCALABILIDAD:

Los requisitos de rendimiento y escalabilidad son fundamentales para garantizar que Pokelytics funcione de manera eficiente y pueda manejar un número creciente de usuarios y datos. A continuación, se detallan algunos aspectos clave a considerar en este punto:

- a) Tiempo de respuesta: La aplicación debe tener un tiempo de respuesta rápido para las interacciones del usuario, como la carga de información de Pokémon, la realización de búsquedas o la navegación entre pantallas. El tiempo de respuesta debe ser lo suficientemente rápido como para proporcionar una experiencia fluida y sin retrasos perceptibles.

- b) Escalabilidad: La aplicación debe ser capaz de manejar un crecimiento en la cantidad de usuarios y datos sin degradar su rendimiento. Esto implica diseñar una arquitectura escalable que permita agregar servidores adicionales o recursos de hardware según sea necesario para satisfacer la demanda creciente.
- c) Gestión eficiente de datos: La aplicación debe ser capaz de gestionar eficientemente la base de datos de Pokémon, garantizando un acceso rápido a la información y minimizando el tiempo de consulta. Esto implica optimizar las consultas y tener en cuenta el crecimiento de la base de datos a largo plazo.
- d) Tolerancia a fallos: La aplicación debe ser robusta y tener mecanismos de recuperación ante posibles fallos. Esto puede incluir copias de seguridad regulares de los datos, detección de errores y recuperación automática, así como la capacidad de manejar situaciones inesperadas sin afectar negativamente la experiencia del usuario.
- e) Eficiencia de recursos: La aplicación debe utilizar los recursos del sistema de manera eficiente, como la memoria, el almacenamiento y el consumo de energía. Esto garantizará un rendimiento óptimo y una experiencia fluida sin agotar los recursos del dispositivo del usuario.
- f) Pruebas de rendimiento: Se deben realizar pruebas exhaustivas de rendimiento para garantizar que la aplicación cumpla con los requisitos establecidos. Esto implica simular cargas de usuarios y situaciones de alto tráfico para evaluar el rendimiento en condiciones reales y realizar ajustes y mejoras según sea necesario.
- g) Optimización de velocidad: La aplicación debe ser optimizada en términos de velocidad de carga y rendimiento general. Se deben utilizar técnicas de compresión, almacenamiento en caché y carga diferida de datos para reducir los tiempos de carga y mejorar la respuesta general de la aplicación.
- h) Monitorización y análisis: La aplicación debe contar con herramientas de monitorización y análisis para supervisar el rendimiento y la utilización de recursos en tiempo real. Esto permitirá detectar y solucionar problemas de rendimiento, así como identificar áreas de mejora y optimización continuas.

Al tener en cuenta estos requisitos de rendimiento y escalabilidad, se garantiza que Pokelytics pueda funcionar de manera eficiente y confiable, incluso en escenarios de alta demanda y crecimiento. Esto asegurará una experiencia de usuario satisfactoria y permitirá que la aplicación se adapte a futuras necesidades y expansiones.

6.1.6 REQUISITOS DE SEGURIDAD Y PRIVACIDAD:

En el desarrollo de la aplicación de la Pokédex, es fundamental considerar los requisitos de seguridad y privacidad para proteger la información de los usuarios y garantizar una experiencia confiable. A continuación, se detallan algunos aspectos clave a tener en cuenta en este punto:

- a) Autenticación y autorización: La aplicación debe contar con un sistema de autenticación seguro para garantizar que solo los usuarios autorizados tengan acceso a la información y funcionalidades. Esto implica el uso de contraseñas seguras y la gestión adecuada de permisos y roles de usuario.
- b) Protección de datos personales: La aplicación debe cumplir con las regulaciones y leyes de protección de datos personales, como el Reglamento General de Protección de Datos (RGPD) en la Unión Europea. Esto incluye la recopilación y gestión adecuada de datos personales, el consentimiento explícito del usuario y la implementación de medidas de seguridad para proteger la información sensible.
- c) Encriptación de datos: La aplicación debe utilizar técnicas de encriptación para proteger los datos en tránsito y en reposo. Esto implica el uso de protocolos seguros de transferencia de datos, como HTTPS, y el almacenamiento encriptado de datos sensibles en la base de datos.
- d) Prevención de ataques: La aplicación debe implementar medidas de seguridad para prevenir y mitigar ataques informáticos, como ataques de inyección SQL, ataques de denegación de servicio (DDoS) o intentos de acceso no autorizado. Esto puede incluir el uso de firewalls, protección contra malware y pruebas de seguridad regulares.
- e) Actualizaciones de seguridad: La aplicación debe recibir actualizaciones regulares de seguridad para abordar vulnerabilidades conocidas y mantener la protección contra nuevas amenazas. Esto implica seguir las recomendaciones de seguridad de los frameworks y librerías utilizadas, y estar al tanto de los parches y actualizaciones disponibles.
- f) Privacidad y consentimiento: La aplicación debe proporcionar a los usuarios información clara sobre las prácticas de privacidad y recopilación de datos, así como la posibilidad de dar su consentimiento para el procesamiento de sus datos. Los usuarios deben tener el control sobre qué información se recopila, cómo se utiliza y con quién se comparte.
- g) Eliminación segura de datos: La aplicación debe permitir a los usuarios eliminar de manera segura sus datos personales y cualquier otra información asociada a su cuenta. Esto implica garantizar que los datos se eliminen de forma permanente de la base de datos y cualquier otro sistema de almacenamiento.

En este sentido Firebase es el encargado de realizar y mantener todo este tipo de acciones, ya que Google respalda la seguridad de los datos contenidos en Firebase por lo que a nivel de aplicación no es necesario implementar sistemas de seguridad más allá de los habituales en lo que a datos respecta.

Al cumplir con estos requisitos de seguridad y privacidad, Pokelytics brindará a los usuarios confianza en la protección de sus datos personales y garantizará la integridad y confidencialidad de la información. Esto ayudará a construir una relación sólida con los usuarios y a cumplir con las regulaciones y estándares de seguridad aplicables.

6.1.7 REQUISITOS DE MANTENIMIENTO Y SOPORTE:

Los requisitos de mantenimiento y soporte son fundamentales para garantizar el funcionamiento continuo y la disponibilidad de la aplicación de la Pokédex. A continuación, se detallan algunos aspectos clave a considerar en este punto:

- a) Actualizaciones de software: La aplicación debe contar con un plan de actualizaciones regulares para corregir errores, agregar nuevas funcionalidades y mejorar el rendimiento. Esto implica mantenerse al tanto de las actualizaciones de los frameworks y librerías utilizadas, y aplicar los parches y mejoras pertinentes.
- b) Monitorización del rendimiento: La aplicación debe ser monitoreada de forma continua para detectar problemas de rendimiento, errores o caídas del sistema. Esto puede implicar el uso de herramientas de monitorización y registro de eventos.
- c) Soporte técnico: La aplicación debe contar con un canal de soporte técnico para que los usuarios puedan informar sobre problemas, recibir asistencia y realizar consultas relacionadas con su uso.
- d) Respaldo y recuperación de datos: La aplicación debe contar con un sistema de respaldo regular de los datos para evitar la pérdida de información en caso de fallas o incidentes. Además, se debe establecer un plan de recuperación de datos en caso de desastres, para garantizar la disponibilidad de la información crítica.
- e) Mantenimiento del servidor: Si la aplicación requiere un servidor dedicado, se deben establecer procedimientos de mantenimiento periódico para garantizar su buen funcionamiento. Esto incluye la instalación de actualizaciones de sistema operativo, la optimización de recursos y el seguimiento de los registros de errores y rendimiento.
- f) Documentación técnica: La aplicación debe contar con una documentación técnica completa que describa su funcionamiento, configuración y uso. Esto facilitará el mantenimiento continuo, el soporte técnico y la capacitación del personal encargado de la aplicación.
- g) Gestión de versiones: La aplicación debe contar con un sistema de gestión de versiones para mantener un registro de las diferentes versiones del software y las actualizaciones realizadas. Esto permitirá un seguimiento claro de los cambios, facilitará la identificación de problemas y proporcionará un historial de actualizaciones.
- h) Pruebas de calidad: Es importante realizar pruebas de calidad periódicas para garantizar el correcto funcionamiento de la aplicación y la detección temprana de posibles problemas. Esto incluye pruebas de regresión, pruebas de rendimiento y pruebas de seguridad para mantener la calidad y estabilidad del sistema.

Al tener en cuenta estos requisitos de mantenimiento y soporte, se asegura que la aplicación pueda recibir las actualizaciones necesarias, contar con un soporte técnico eficiente y mantener un alto nivel de disponibilidad y rendimiento a lo largo del tiempo.

6.2 SELECCIÓN DE LA PLATAFORMA TECNOLÓGICA

El proyecto de desarrollo de Pokelytics requiere una cuidadosa selección de la plataforma tecnológica adecuada para su implementación. La elección de la plataforma correcta es fundamental para garantizar el rendimiento, la usabilidad y la escalabilidad de la aplicación. Algunos aspectos clave a considerar al seleccionar la plataforma tecnológica para la Pokédex:

1. Plataforma móvil: Dado que Pokelytics es una aplicación dirigida a dispositivos móviles, se debe considerar la selección de una plataforma móvil popular y ampliamente utilizada, como iOS (para dispositivos iPhone y iPad) o Android (para dispositivos con sistema operativo Android). La elección de la plataforma debe basarse en la investigación del mercado objetivo y las preferencias de los usuarios. Como hemos abordado en puntos anteriores nuestro desarrollo se fundamentará en Android por el amplio mercado que tiene y puesto que el desarrollo de ésta se va a realizar con Android Studio es lo más viable.
2. Lenguaje de programación: Dependiendo de la plataforma seleccionada, se deben evaluar los lenguajes de programación disponibles para el desarrollo de aplicaciones móviles. Por ejemplo, para iOS se puede utilizar Swift o Objective-C, mientras que para Android se puede utilizar Java o Kotlin. La elección del lenguaje debe considerar la experiencia y habilidades del equipo de desarrollo, así como las ventajas y características ofrecidas por cada lenguaje. Pokelytics cuenta con un equipo de desarrollo que tiene conocimientos para desarrollar la app con Java y Kotlin por lo que éstos son los lenguajes empleados en el desarrollo de la app.
3. Frameworks y herramientas de desarrollo: Existen varios frameworks y herramientas disponibles que facilitan el desarrollo de aplicaciones móviles. Por ejemplo, para el desarrollo de aplicaciones iOS, se puede considerar el uso de frameworks como SwiftUI o UIKit, y para Android, se puede evaluar el uso de Android SDK como es nuestro caso, junto con frameworks como React Native o Flutter. La selección de frameworks y herramientas debe considerar la productividad del desarrollo, la facilidad de mantenimiento y el rendimiento de la aplicación.
4. Integraciones y APIs: Pokelytics se beneficia de integraciones con servicios externos, como APIs de Pokémon para obtener datos actualizados, servicios de autenticación, almacenamiento en la nube y otros servicios relacionados. Al seleccionar la plataforma tecnológica, se deben considerar las facilidades que ofrece para integraciones y la disponibilidad de APIs relevantes. Como hemos comentado en puntos anteriores la API a la que tiene conexión Pokelytics es PokeApi.
5. Experiencia del equipo de desarrollo: Es importante evaluar la experiencia y las habilidades del equipo de desarrollo en la plataforma tecnológica seleccionada. Un equipo con experiencia previa en el desarrollo de aplicaciones móviles en la plataforma elegida puede acelerar el proceso de desarrollo, reducir la curva de aprendizaje y garantizar la calidad del código y la implementación. Dado que es el proyecto de final del curso, la experiencia como desarrolladores aun no es muy elevada. Sin embargo, alguno de los miembros tiene grandes conocimientos en las herramientas que hemos empleado por lo que el proyecto se ha desarrollado de una forma muy efectiva. Además gracias a internet y a toda la

información que podemos encontrar en la red, cualquiera de los problemas surgidos ha sido solucionado en un lapso de tiempo bastante breve.

6. Consideraciones de rendimiento y escalabilidad: La plataforma tecnológica seleccionada debe ser capaz de manejar eficientemente el rendimiento y la escalabilidad de la aplicación. Esto implica evaluar la capacidad de la plataforma para gestionar grandes cantidades de datos, realizar consultas rápidas a la base de datos y manejar un alto número de usuarios simultáneos sin degradar el rendimiento.
7. Compatibilidad con actualizaciones futuras: La elección de la plataforma tecnológica debe considerar su capacidad para soportar actualizaciones futuras y mantenerse al día con las últimas tendencias y avances tecnológicos. Esto asegurará que la app pueda adaptarse a cambios en los requisitos del negocio, nuevas funcionalidades y mejoras en la experiencia del usuario.

6.3 DESCRIPCIÓN DEL DISEÑO DE LA SOLUCIÓN

Respecto al diseño de la solución cabe destacar que el proyecto en cuanto a la arquitectura se ha estructurado en directorios para que todo se constituya de una forma comprensible y accesible. Cada directorio aloja diferentes funcionalidades que hacen que, a la hora de leer, añadir, interpretar o modificar funcionalidades sea sencillo para aquel que lo realiza.

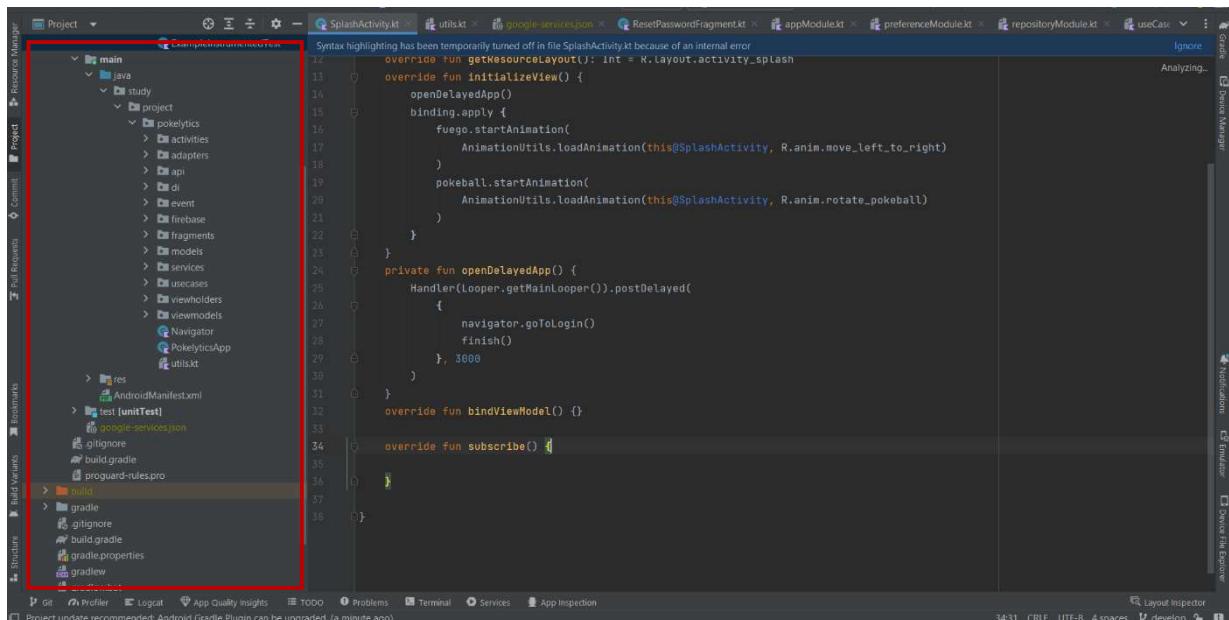


Imagen 17 Estructura del proyecto

Sobre la interfaz del usuario, se ha buscado que el usuario pueda manejar la app de manera intuitiva y eficaz. La aplicación sería fácilmente utilizable por un niño puesto que las funcionalidades son sencillas, rápidas y no hay un exceso de posibilidades que hagan que la aplicación se ralentice ni que el usuario pueda perderse en el contenido o a la hora de buscar alguna de las funcionalidades de las que dispone.

Puesto que los datos se almacenan en Firebase, el diseño de almacenamiento de estos datos no ha sido necesario hacerlo más complejo. Simplemente hemos aprovechado la posibilidad que nos brinda Google de utilizar esta herramienta y la hemos aplicado de la manera más sencilla sin que ello implique ningún tipo de riesgo para la información aquí almacenada.

En el ámbito de la seguridad de la aplicación, dado que empleamos herramientas con un gran respaldo a nivel de seguridad como son Android Studio, Firebase y la propia API, éstas son las encargadas de hacer una gestión segura tanto del código, los datos y la información contenida en la aplicación.

Sobre el diseño de las pruebas que realizamos a la aplicación se ha pensado en las habituales que se hacen en el desarrollo de cualquier otra aplicación, empleando los métodos que podríamos emplear en otra app de mayor envergadura. Éstas pruebas son pruebas unitarias se han realizado con Android Studio, Mockito, JUnit... Todo ello además acompañado de metodologías TDD que permiten que el desarrollo de la app sea más óptimo aun si cabe.

7. DESCRIPCIÓN DE LA SOLUCIÓN: CONSTRUCCIÓN

7.1 DOCUMENTACIÓN DESCRIPTIVA

7.1.1 ACTIVITYBASE:

El código proporcionado muestra una clase abstracta llamada ActivityBase. Esta clase está diseñada para ser utilizada como base para otras actividades en una aplicación de Android. La clase utiliza el enlace de vista de datos (ViewDataBinding) para facilitar la vinculación de vistas y datos en las actividades derivadas. Esta clase abstracta proporciona una estructura común para las actividades en una aplicación de Android que utilizan enlace de vista de datos. Las clases derivadas deben implementar los métodos abstractos para personalizar el comportamiento específico de cada actividad.

A continuación, se explica qué hace cada parte del código:

1. La línea abstract class ActivityBase<T : ViewDataBinding>: AppCompatActivity() define la clase abstracta ActivityBase que extiende AppCompatActivity. La clase toma un parámetro de tipo genérico T que debe ser una subclase de ViewDataBinding. Esto permite que las clases derivadas especifiquen el tipo de enlace de vista de datos que desean utilizar.
2. La línea lateinit var binding: T declara una variable llamada binding que será utilizada para almacenar la instancia de enlace de vista de datos. La anotación lateinit indica que la inicialización de esta variable se realizará más adelante en el código.
3. La línea val navigator: Navigator by inject { parametersOf(this) } declara una propiedad llamada navigator que se inicializa utilizando la biblioteca de inyección de dependencias Koin. La propiedad navigator se inyecta con una instancia de la clase Navigator, y se le pasa this (la instancia actual de la actividad) como argumento.
4. El método onCreate(savedInstanceState: Bundle?) es el método de ciclo de vida de la actividad que se ejecuta cuando la actividad se crea. En este método, se realiza lo siguiente:
 - Se llama al método super.onCreate(savedInstanceState) para asegurarse de que el comportamiento predeterminado de AppCompatActivity se ejecute correctamente.
 - Se utiliza DataBindingUtil.setContentView(this, getResourceLayout()) para inflar y establecer el contenido de la vista de la actividad mediante enlace de vista de datos. La función DataBindingUtil.setContentView() toma como argumento el objeto this (la instancia actual de la actividad) y el resultado del método getResouceLayout() que debe ser implementado por las clases derivadas para proporcionar el diseño de recursos correspondiente a la actividad.
 - Se llaman a los métodos bindViewModel(), initializeView(), subscribe() y setTopBar() que deben ser implementados por las clases derivadas para realizar la vinculación de vistas y datos, inicializar la vista, suscribirse a eventos y configurar la barra superior respectivamente.

5. El método `setTopBar()` es un método vacío que puede ser implementado por las clases derivadas para configurar la barra superior de la actividad.
6. Los métodos `getResourceLayout()`, `initializeView()`, `bindViewModel()` y `subscribe()` son métodos abstractos que deben ser implementados por las clases derivadas de `ActivityBase`. Estos métodos permiten personalizar el diseño de recursos de la actividad, inicializar la vista, vincular la vista con el modelo de vista correspondiente y suscribirse a eventos específicos.

```
1 package study.project.pokelytics.activities
2
3 import android.os.Bundle
4 import androidx.appcompat.app.AppCompatActivity
5 import androidx.databinding.DataBindingUtil
6 import androidx.databinding.ViewDataBinding
7 import org.koin.android.ext.android.inject
8 import org.koin.core.parameter.parametersOf
9 import study.project.pokelytics.Navigator
10
11 abstract class ActivityBase<T : ViewDataBinding> : AppCompatActivity() {
12
13     lateinit var binding: T
14     val navigator: Navigator by inject { parametersOf(...parameters: this) }
15
16     override fun onCreate(savedInstanceState: Bundle?) {
17         super.onCreate(savedInstanceState)
18         binding = DataBindingUtil.setContentView(activity, getResourceLayout())
19         bindViewModel()
20         initializeView()
21         subscribe()
22         setTopBar()
23     }
24
25     private fun setTopBar() {}
26
27     protected abstract fun getResourceLayout(): Int
28
29     protected abstract fun initializeView()
30
31     protected abstract fun bindViewModel()
32
33     protected abstract fun subscribe()
34
35 }
```

Imagen 18 ActivityBase

7.1.2 LOGINACTIVITY

La clase LoginActivity que extiende la clase ActivityBase<ActivityLoginBinding>. Esta clase representa una actividad en una aplicación de Android que se utiliza para iniciar sesión. Proporciona implementaciones personalizadas para los métodos abstractos heredados. Estos métodos se utilizan para configurar la vista, vincular la vista con el modelo de vista y proporcionar el diseño de recursos correspondiente. La clase también define un método estático para obtener un intento para iniciar esta actividad.

La línea `class LoginActivity : ActivityBase<ActivityLoginBinding>()` define la clase LoginActivity que hereda de la clase ActivityBase parametrizada con ActivityLoginBinding. Esto significa que LoginActivity utilizará la clase ActivityLoginBinding para realizar el enlace de vistas y datos.

1. El método getResourceLayout(): Int se sobrescribe para proporcionar el diseño de recursos correspondiente a la actividad de inicio de sesión. En este caso, devuelve R.layout.activity_login, que es el identificador del archivo de diseño XML activity_login.xml.
2. El método initializeView() se sobrescribe para inicializar la vista de la actividad. En este caso, se encuentra el fragmento de navegación (NavHostFragment) en la vista y se infla un gráfico de navegación (nav_graph_login) en el controlador de navegación asociado. Esto configura la navegación de la actividad de inicio de sesión.
3. El método bindViewModel() se sobrescribe para realizar la vinculación entre la vista y el modelo de vista. Aquí, se establece el lifecycleOwner del enlace de vista de datos (binding) para que sea esta instancia de LoginActivity. Esto permite que la vista observe los cambios en los datos del modelo de vista.
4. El método subscribe() se sobrescribe, pero en este caso, no contiene ninguna implementación. Este método se puede utilizar para suscribirse a eventos u observar cambios en los datos.
5. El bloque companion object define un método estático getIntent(context: Context) que devuelve un intento para iniciar LoginActivity. Esto permite que otras partes de la aplicación obtengan un intento para iniciar esta actividad específica.

```
1 package study.project.pokelytics.activities
2
3 import android.content.Context
4 import android.content.Intent
5 import androidx.navigation.fragment.NavHostFragment
6 import study.project.pokelytics.R
7 import study.project.pokelytics.databinding.ActivityLoginBinding
8
9 class LoginActivity : ActivityBase<ActivityLoginBinding>() {
10
11     override fun getResourceLayout(): Int = R.layout.activity_login
12
13     override fun initializeView() {
14         val navHost = supportFragmentManager.findFragmentById(R.id.nav_host_fragment) as NavHostFragment
15         val graph = navHost.navController.navInflater.inflate(R.navigation.nav_graph_login)
16         navHost.navController.graph = graph
17     }
18
19     override fun bindViewModel() {
20         binding.apply { this@ActivityLoginBinding
21             lifecycleOwner = this@LoginActivity
22         }
23     }
24
25     override fun subscribe() {
26     }
27
28     companion object {
29         fun getIntent(context: Context) = Intent(context, LoginActivity::class.java)
30     }
31 }
```

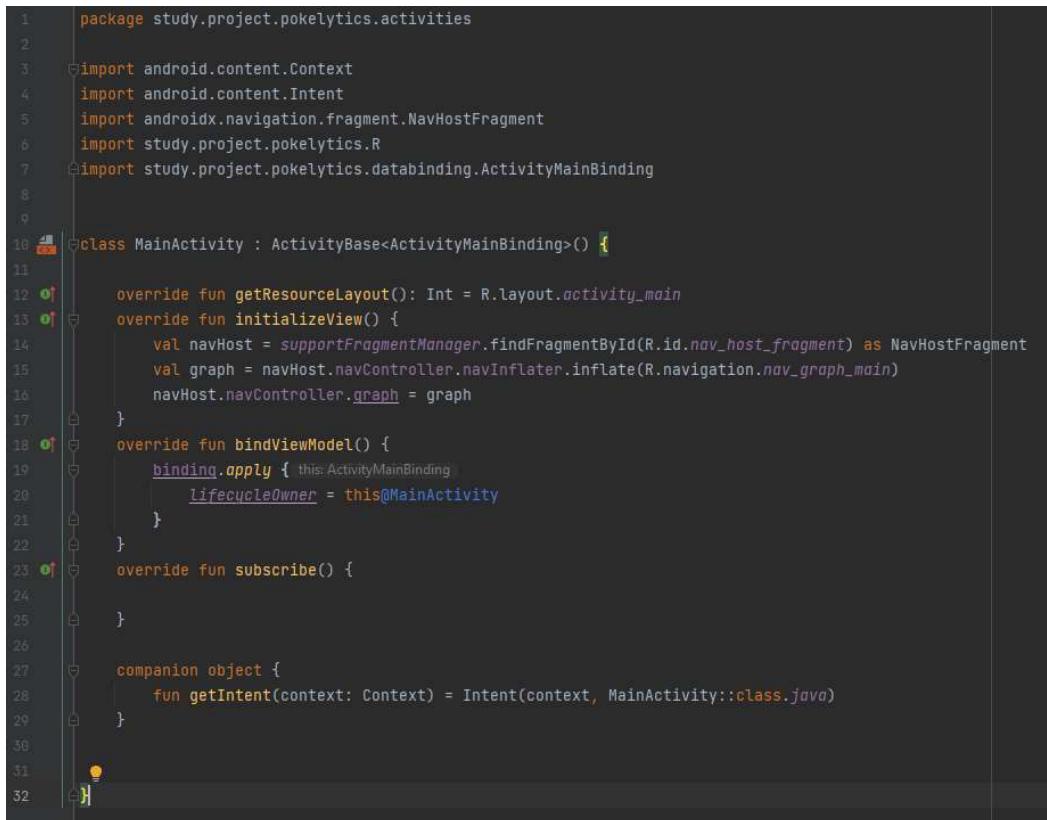
Imagen 19 LoginActivity

7.1.3 MAINACTIVITY

La clase llamada `MainActivity` que extiende la clase `ActivityBase<ActivityMainBinding>`. Esta clase representa la actividad principal de una aplicación de Android. Proporciona implementaciones personalizadas para los métodos abstractos heredados. Estos métodos se utilizan para configurar la vista, vincular la vista con el modelo de vista y proporcionar el diseño de recursos correspondiente. La clase también define un método estático para obtener un intento para iniciar esta actividad.

1. La línea `class MainActivity : ActivityBase<ActivityMainBinding>()` define la clase `MainActivity` que hereda de la clase `ActivityBase` parametrizada con `ActivityMainBinding`. Esto significa que `MainActivity` utilizará la clase `ActivityMainBinding` para realizar el enlace de vistas y datos.
2. El método `getResourceLayout(): Int` se sobrescribe para proporcionar el diseño de recursos correspondiente a la actividad principal. En este caso, devuelve `R.layout.activity_main`, que es el identificador del archivo de diseño XML `activity_main.xml`.
3. El método `initializeView()` se sobrescribe para inicializar la vista de la actividad. En este caso, se encuentra el fragmento de navegación (`NavHostFragment`) en la vista y se infla un gráfico de navegación (`nav_graph_main`) en el controlador de navegación asociado. Esto configura la navegación de la actividad principal.
4. El método `bindViewModel()` se sobrescribe para realizar la vinculación entre la vista y el modelo de vista. Aquí, se establece el `lifecycleOwner` del enlace de vista de datos (`binding`) para que sea esta instancia de `MainActivity`. Esto permite que la vista observe los cambios en los datos del modelo de vista.

5. El método subscribe() se sobrescribe, pero en este caso, no contiene ninguna implementación. Este método se puede utilizar para suscribirse a eventos u observar cambios en los datos.
6. El bloque companion object define un método estático getIntent(context: Context) que devuelve un intento para iniciar MainActivity. Esto permite que otras partes de la aplicación obtengan un intento para iniciar esta actividad específica.



```

1 package study.project.pokelytics.activities
2
3 import android.content.Context
4 import android.content.Intent
5 import androidx.navigation.fragment.NavHostFragment
6 import study.project.pokelytics.R
7 import study.project.pokelytics.databinding.ActivityMainBinding
8
9
10 class MainActivity : ActivityBase<ActivityMainBinding>() {
11
12     override fun getLayoutResource(): Int = R.layout.activity_main
13     override fun initializeView() {
14         val navHost = supportFragmentManager.findFragmentById(R.id.nav_host_fragment) as NavHostFragment
15         val graph = navHost.navController.navInflater.inflate(R.navigation.nav_graph_main)
16         navHost.navController.graph = graph
17     }
18     override fun bindViewModel() {
19         binding.apply { this@ActivityMainBinding.lifecycleOwner = this@MainActivity
20             }
21     }
22     override fun subscribe() {
23     }
24
25
26     companion object {
27         fun getIntent(context: Context) = Intent(context, MainActivity::class.java)
28     }
29
30
31 }
32

```

Imagen 20 MainActivity

7.1.4 SPLASHACTIVITY

El código proporcionado muestra una clase llamada SplashActivity que extiende la clase ActivityBase<ActivitySplashBinding>. Esta clase representa una actividad de pantalla de inicio en una aplicación de Android. Esta clase proporciona implementaciones personalizadas para algunos métodos heredados. En el método onCreate(), se configura la vista de la actividad de la pantalla de inicio y se inician animaciones en las vistas. Después de un retraso de 3 segundos, se abre la siguiente pantalla llamando a navigator.goToLogin().

1. La anotación @SuppressLint("CustomSplashScreen") indica que se debe suprimir el mensaje de advertencia relacionado con la personalización de la pantalla de inicio. Esto se utiliza en relación con las nuevas directrices de personalización de la pantalla de inicio en Android.
2. El método onCreate(savedInstanceState: Bundle?) es el método de ciclo de vida de la actividad que se ejecuta cuando la actividad se crea. En este método, se realiza lo siguiente:

- Se llama al método `super.onCreate(savedInstanceState)` para asegurarse de que el comportamiento predeterminado se ejecute correctamente.
 - Se utiliza `setContentView(R.layout.activity_splash)` para inflar y establecer el contenido de la vista de la actividad con el diseño XML `activity_splash`.
 - Se encuentran las referencias a las vistas fuego y pokeball utilizando `findViewById`.
 - Se cargan las animaciones `animFuego` y `animPokeball` desde los recursos XML utilizando `AnimationUtils.loadAnimation`.
 - Se inicia la animación en las vistas fuego y pokeball utilizando `startAnimation`.
3. El método `openApp()` es un método privado que se utiliza para abrir la aplicación después de un cierto período de tiempo. Utiliza un Handler para retrasar la ejecución de un bloque de código que contiene la llamada a `navigator.goToLogin()`. En este caso, se programó para ejecutarse después de 3000 milisegundos (3 segundos).
 4. `El método getResourceLayout()`: Int se sobrescribe para proporcionar el diseño de recursos correspondiente a la actividad de la pantalla de inicio. En este caso, devuelve `R.layout.activity_splash`, que es el identificador del archivo de diseño XML `activity_splash.xml`.
 5. Los métodos `initializeView()`, `bindViewModel()` y `subscribe()` se sobrescriben, pero en este caso, no contienen ninguna implementación. Estos métodos se pueden utilizar para realizar tareas específicas de la vista, vincular la vista con el modelo de vista y suscribirse a eventos, respectivamente. En el caso de la actividad de la pantalla de inicio, no se requieren tareas adicionales en estos métodos.

```

1 package study.project.pokelytics.activities
2
3 import android.annotation.SuppressLint
4 import android.os.Bundle
5 import android.os.Handler
6 import android.os.Looper
7 import android.view.animation.AnimationUtils
8 import android.widget.ImageView
9 import study.project.pokelytics.NAVIGATE_TIMEOUT
10 import study.project.pokelytics.R
11 import study.project.pokelytics.databinding.ActivitySplashBinding
12
13 @SuppressLint("CustomSplashScreen")
14 class SplashActivity : ActivityBase<ActivitySplashBinding>() {
15
16     override fun onCreate(savedInstanceState: Bundle?) {
17         super.onCreate(savedInstanceState)
18         setContentView(R.layout.activity_splash)
19
20         openApp()
21
22         val fuego = findViewById<ImageView>(R.id.fuego)
23         val pokeball = findViewById<ImageView>(R.id.pokeball)
24
25         val animFuego = AnimationUtils.loadAnimation(context: this, R.anim.move_left_to_right)
26         val animPokeball = AnimationUtils.loadAnimation(context: this, R.anim.rotate_pokeball)
27
28         fuego.startAnimation(animFuego)
29         pokeball.startAnimation(animPokeball)
30     }
31
32     private fun openApp() {
33         Handler(Looper.getMainLooper()).postDelayed(
34             {
35                 navigator.goToLogin()
36             }, delayMillis: 3000
37         )
38     }
39
40     override fun getResourceLayout(): Int = R.layout.activity_splash
41
42     override fun initializeView() {}
43
44     override fun bindViewModel() {}
45
46     override fun subscribe() {}
47

```

Imagen 21 `SplashActivity`

7.2 ¿POR QUÉ HEMOS ELEGIDO LENGUAJES BASADOS EN C?

La elección de los lenguajes de programación para el desarrollo de aplicaciones móviles es un aspecto crucial que debe evaluarse cuidadosamente. En el caso de Pokelytics, se optó por lenguajes basados en C, como Java y Kotlin, en lugar de opciones más reactivas, como React Native. (A continuación, se explorarán las razones detrás de esta elección y se analizará por qué Kotlin resulta más útil en comparación con Java en diversos escenarios.)

Uno de los principales motivos para seleccionar lenguajes basados en C, en lugar de lenguajes más reactivos, es la necesidad de maximizar el rendimiento y la eficiencia en el desarrollo de aplicaciones móviles. Los lenguajes basados en C están más cerca del lenguaje de bajo nivel y, por lo tanto, permiten un control más preciso del hardware y los recursos del dispositivo. Esto es especialmente importante en aplicaciones que requieren un alto grado de optimización y rendimiento, como es el caso de Pokelytics, donde se pueden realizar operaciones de cálculo y manipulación de datos en tiempo real.

Además, el equipo de desarrollo de Pokelytics cuenta con experiencia y conocimientos en Kotlin, lo cual fue un factor determinante en la elección de estos lenguajes. La familiaridad y dominio de un lenguaje de programación por parte del equipo de desarrollo resulta fundamental para garantizar un desarrollo eficiente, una depuración efectiva y un mantenimiento sostenible a largo plazo de la aplicación.

En cuanto a la elección específica entre Java y Kotlin, se consideró que Kotlin es más útil en todos los escenarios en comparación con Java. Kotlin es un lenguaje moderno que se ha desarrollado con el objetivo de mejorar y superar las limitaciones de Java, brindando un conjunto de características más concisas, expresivas y seguras. Kotlin permite una programación más declarativa y funcional, lo cual facilita el desarrollo de código más legible, mantenable y libre de errores.

Kotlin también ofrece una excelente interoperabilidad con Java, lo que significa que el código Java existente puede ser fácilmente utilizado en proyectos Kotlin y viceversa. Esto es especialmente valioso para aplicaciones como Pokelytics, donde es posible que haya componentes o bibliotecas escritas en Java que deban integrarse con el nuevo código Kotlin. Además, Kotlin proporciona herramientas y extensiones que simplifican la programación en Android, como la eliminación de la verbosidad en la definición de propiedades, el manejo nulo seguro y la compatibilidad con las características más recientes de la plataforma.

Además, Google eligió a Kotlin como lenguaje oficial de programación para Android debido a su alta compatibilidad con Java, mejoras significativas y creciente adopción en la industria.

La elección del lenguaje de programación adecuado es un factor crítico para el éxito de un proyecto de desarrollo de software. Uno de los lenguajes más utilizados y respetados en la industria es C, que ha sido ampliamente adoptado y sigue siendo relevante incluso después de décadas desde su creación.

1. Eficiencia y rendimiento: C es conocido por su eficiencia y rendimiento, lo cual lo convierte en una opción preferida para aplicaciones que requieren un procesamiento rápido y una gestión eficiente de los recursos del sistema. Este lenguaje se sitúa a un nivel cercano al lenguaje máquina, lo que significa que se puede acceder

directamente a la memoria y los recursos del sistema. Esta capacidad brinda a los desarrolladores un control preciso sobre el rendimiento de sus programas.

Dado que C se ha utilizado durante mucho tiempo en una amplia variedad de aplicaciones, existe una gran cantidad de bibliotecas y herramientas disponibles para optimizar y mejorar el rendimiento de los programas escritos en este lenguaje. Además, C se utiliza a menudo como base para otros lenguajes de programación de alto nivel, lo que demuestra su importancia en la construcción de software eficiente y potente.

2. Portabilidad y compatibilidad: Otra ventaja clave de los lenguajes de programación basados en C es su portabilidad y compatibilidad con diferentes plataformas y sistemas operativos. Los programas escritos en C se pueden compilar y ejecutar en una amplia gama de sistemas, desde sistemas embebidos hasta supercomputadoras.

La portabilidad de C se debe a su enfoque en la programación a nivel de sistema y su dependencia mínima de características específicas de la plataforma. Esto significa que los programas escritos en C se pueden migrar y ejecutar en diferentes entornos sin necesidad de realizar modificaciones significativas. Esta característica es especialmente valiosa en el desarrollo de software que necesita funcionar en varios sistemas operativos o dispositivos.

3. Flexibilidad y control: C es un lenguaje de programación de bajo nivel que proporciona a los desarrolladores un alto grado de flexibilidad y control sobre los detalles internos del programa. A diferencia de los lenguajes de programación de alto nivel que ofrecen abstracciones y simplificaciones, C permite a los programadores trabajar directamente con la memoria y las estructuras de datos. Esto permite optimizar los programas y adaptarlos a necesidades específicas.

La flexibilidad de C también se refleja en su capacidad para interactuar con el hardware del sistema a través de punteros y manipulación de bits. Esto es especialmente importante en el desarrollo de controladores de dispositivo, sistemas operativos y aplicaciones de tiempo real, donde el acceso directo a los recursos del hardware es esencial.

Además, C ofrece un amplio conjunto de características y constructos de programación que permiten implementar algoritmos complejos y estructuras de datos eficientes. Esta capacidad de abordar problemas de programación de manera directa y eficiente es uno de los aspectos más destacados de C.

Los lenguajes de programación basados en C han llevado a la adopción generalizada de C en la industria y lo han mantenido como un lenguaje de programación relevante y poderoso a lo largo del tiempo. Al elegir C como lenguaje principal, los desarrolladores pueden aprovechar estas ventajas para construir software rápido, eficiente y altamente adaptable.

Todo esto hace que el equipo de desarrollo haya optado por emplear lenguajes basados en C para la construcción de la solución de Pokelytics. Además trabajar con estos lenguajes es algo muy común en el mercado actual por lo que trabajar con éstos supone que a posteriori el equipo de desarrollo pueda tener cierta facilidad para encontrar un hueco en futuros proyectos de empresas competitivas.

7.3 USO DE LA CLASE ENCRYPTEDSHARED PREFERENCES

La clase EncryptedSharedPreferences de Android es una poderosa herramienta que ofrece una solución segura para el almacenamiento de datos en SharedPreferences. Con su cifrado AES y su enfoque en la integridad de los datos, esta clase se destaca por varios puntos positivos que la convierten en una elección ideal para proteger información sensible en aplicaciones Android. Sin embargo, también presenta algunas limitaciones que deben tenerse en cuenta. A continuación, destacaremos cinco puntos fundamentales positivos y dos negativos de la clase EncryptedSharedPreferences.

7.3.1 SEGURIDAD AVANZADA

Uno de los principales puntos positivos de EncryptedSharedPreferences es su enfoque en la seguridad. Utiliza el cifrado AES (Advanced Encryption Standard) con una clave generada a partir de una contraseña y un salto (salt), lo que proporciona una sólida capa de protección para los datos almacenados. Además, la autenticación basada en MAC (Message Authentication Code) garantiza la integridad de los datos y previene cualquier manipulación no autorizada.

7.3.2 INTEGRACIÓN SENCILLA

EncryptedSharedPreferences se integra perfectamente con la infraestructura existente de SharedPreferences en Android. La API es bastante similar a la de SharedPreferences convencionales, lo que facilita la migración y la implementación en proyectos existentes. Los métodos para almacenar y recuperar datos son los mismos, lo que permite un uso sencillo y sin complicaciones.

7.3.3 COMPATIBILIDAD CON VERSIONES ANTERIORES:

Aunque EncryptedSharedPreferences se introdujo en Android 6.0 (Marshmallow), está disponible a través de AndroidX Security, lo que permite utilizarlo en versiones anteriores de Android a través de la compatibilidad de bibliotecas. Esto es especialmente importante para los desarrolladores que desean ofrecer una mayor seguridad a los usuarios sin perder la compatibilidad con versiones más antiguas del sistema operativo.

7.3.4 CIFRADO DE VALORES ESPECÍFICOS:

Una ventaja significativa de EncryptedSharedPreferences es su capacidad para cifrar valores específicos en lugar de todo el archivo de SharedPreferences. Esto brinda mayor flexibilidad al desarrollador para elegir qué datos deben ser protegidos y cuáles pueden permanecer sin cifrar, lo que puede ser útil en situaciones donde cierta información no es confidencial.

7.3.5 RENDIMIENTO ACEPTABLE:

Aunque el cifrado y descifrado de datos siempre conlleva una carga adicional, EncryptedSharedPreferences ha sido diseñado para proporcionar un rendimiento aceptable. La implementación de cifrado AES es eficiente y se ha optimizado para minimizar el impacto en el rendimiento de la aplicación. Sin embargo, es importante tener en cuenta que el rendimiento puede variar según el tamaño de los datos y las características del dispositivo.

A pesar de los numerosos puntos positivos, también hay algunas limitaciones asociadas con EncryptedSharedPreferences que deben considerarse:

7.3.5.1 DEPENDENCIA DE LA CONTRASEÑA DEL USUARIO:

La seguridad de EncryptedSharedPreferences depende en gran medida de la fortaleza de la contraseña proporcionada por el usuario. Si el usuario elige una contraseña débil o fácilmente adivinable, el cifrado y la protección de los datos almacenados pueden verse comprometidos. Por lo tanto, es importante educar a los usuarios sobre la importancia de elegir contraseñas seguras y proporcionar mecanismos para garantizar su fortaleza.

7.3.5.2 VULNERABILIDADES EN EL ALMACENAMIENTO TEMPORAL:

Aunque EncryptedSharedPreferences protege los datos almacenados en el dispositivo, puede haber vulnerabilidades en el almacenamiento temporal de los mismos. Durante el uso de la aplicación, los datos se pueden desencriptar temporalmente en la memoria para su acceso, lo que puede ser susceptible a ataques si el dispositivo está comprometido o si hay aplicaciones maliciosas en ejecución en segundo plano. Para mitigar este riesgo, es esencial aplicar buenas prácticas de seguridad en el desarrollo de la aplicación y mantener actualizado el sistema operativo del dispositivo.

EncryptedSharedPreferences es una clase importante de la API de Android que ofrece una solución segura para el almacenamiento de datos en SharedPreferences. Con su cifrado AES y su enfoque en la integridad de los datos, proporciona una capa adicional de protección para información sensible en aplicaciones Android. Su integración sencilla, compatibilidad con versiones anteriores y capacidad de cifrar valores específicos son aspectos destacados que hacen de EncryptedSharedPreferences una opción valiosa para los desarrolladores. Sin embargo, es fundamental tener en cuenta las limitaciones asociadas, como la dependencia de la contraseña del usuario y las vulnerabilidades en el almacenamiento temporal. Al considerar estos aspectos y aplicar buenas prácticas de seguridad, los desarrolladores pueden aprovechar al máximo EncryptedSharedPreferences para garantizar la protección de los datos confidenciales en sus aplicaciones Android.

7.3.6 CÓMO GUARDAR DATOS SIMPLES CON SHARED PREFERENCES

Si tienes una colección relativamente pequeña de pares clave-valor que deseas guardar, debes usar las APIs de SharedPreferences. Un objeto SharedPreferences apunta a un archivo que contiene pares clave-valor y proporciona métodos sencillos para leerlos y escribirlos. El framework administra cada archivo de SharedPreferences, que puede ser privado o compartido.

En esta página, se muestra el uso de las APIs de SharedPreferences para almacenar y recuperar valores simples.

Las APIs de SharedPreferences son para leer y escribir pares clave-valor. No se debe confundir con las APIs de Preference, que te ayudan a crear una interfaz de usuario para la configuración de tu app (aunque también usan SharedPreferences para guardar la configuración del usuario). Para obtener información sobre las APIs de Preference, consulta la Guía para desarrolladores de configuración.

7.3.7 CÓMO CONTROLAR LAS PREFERENCIAS COMPARTIDAS

Puedes crear un nuevo archivo de preferencias compartidas o acceder a uno existente llamando a uno de estos métodos:

- `getSharedPreferences()`: Usa este método si necesitas varios archivos de preferencias compartidas identificados por nombre, que especificas con el primer parámetro. Puedes llamar a este método desde cualquier instancia de Context en tu app.
- `getPreferences()`: Usa este método desde una instancia de Activity si necesitas utilizar un solo archivo de preferencias compartidas para la actividad. Como este método recupera un archivo de preferencias compartidas predeterminado que pertenece a la actividad, no necesitas indicar un nombre.

Por ejemplo, el siguiente código accede al archivo de preferencias compartidas identificado por la string de recursos R.string.preference_file_key y lo abre con el modo privado para que solo tu app pueda acceder al archivo:

```
val sharedPref = activity?.getSharedPreferences(  
    getString(R.string.preference_file_key), Context.MODE_PRIVATE)
```

Cuando asignes un nombre a tus archivos de preferencias compartidas, este debe ser identificable de manera inequívoca con tu app. Una manera fácil de hacerlo es agregar el nombre del archivo al ID de aplicación. Por ejemplo: "com.example.myapp.PREFERENCE_FILE_KEY".

De forma alternativa, si necesitas solo un archivo de preferencias compartidas para tu actividad, puedes utilizar el método `getPreferences()`:

```
val sharedPref = activity?.getPreferences(Context.MODE_PRIVATE)
```

A partir de Android 7.0 (nivel de API 24), Android arroja un SecurityException si los usas. Si tu app necesita compartir archivos privados con otras apps, puede usar un FileProvider con la FLAG_GRANT_READ_URI_PERMISSION. Para obtener más información, consulta Cómo compartir archivos.

Si usas la API de SharedPreferences para guardar la configuración de la app, debes usar `getDefaultSharedPreferences()` en su lugar, a fin de obtener el archivo de preferencias compartidas predeterminado para toda la app. Para obtener más información, consulta la Guía para desarrolladores de configuración.

7.3.8 CÓMO ESCRIBIR EN LAS PREFERENCIAS COMPARTIDAS

Para realizar operaciones de escritura en el archivo de preferencias compartidas, crea un SharedPreferences.Editor llamando a edit() en tu SharedPreferences.

Nota: Puedes editar las preferencias compartidas de manera más segura llamando al método edit() en un objeto EncryptedSharedPreferences, en lugar de hacerlo en un objeto SharedPreferences. Si quieres obtener más información, consulta la guía para trabajar con datos de forma más segura.

Pasa las claves y los valores que deseas escribir con métodos como putInt() y putString(). Luego, llama a apply() o a commit() para guardar los cambios. Por ejemplo:

```
val sharedPref = activity?.getPreferences(Context.MODE_PRIVATE) ?: return
with (sharedPref.edit()) {
    putInt(getString(R.string.saved_high_score_key), newHighScore)
    apply()
}
```

apply() cambia el objeto SharedPreferences en la memoria de inmediato, pero escribe las actualizaciones en el disco de forma asíncrona. Como alternativa, puedes usar commit() para escribir los datos en el disco de forma síncrona. Sin embargo, debido a que commit() es síncrono, debes evitar llamarlo desde tu subprocesso principal, ya que podría pausar el procesamiento de la IU.

7.3.9 CÓMO LEER DESDE LAS PREFERENCIAS COMPARTIDAS

Para recuperar valores de un archivo de preferencias compartidas, llama a métodos como getInt() y getString(), proporciona la clave del valor que deseas y, opcionalmente, un valor predeterminado para mostrar si no se encuentra la clave. Por ejemplo:

```
val sharedPref = activity?.getPreferences(Context.MODE_PRIVATE) ?: return
val defaultValue = resources.getInteger(R.integer.saved_high_score_default_key)
val highScore = sharedPref.getInt(getString(R.string.saved_high_score_key),
    defaultValue)
```

7.4 PROBLEMAS ENCONTRADOS Y SOLUCIONES

Entre los problemas destacados se encuentran las dificultades relacionadas con la migración de datos de la API, los inconvenientes de memoria en dispositivos Android y las complicaciones en la implementación del RecyclerView.

1. Migración de Datos de la API: Durante el proceso de desarrollo, nos encontramos con un problema al migrar los datos de la API a nuestra aplicación. En la API, existían diferentes convenciones de nomenclatura para los identificadores, como guiones medios, guiones bajos y algunos identificadores que también contenían guiones. Esto generó conflictos y dificultades a la hora de mapear correctamente los datos en nuestra aplicación.

Solución: Para resolver este problema, implementamos adapters y excepciones personalizadas. Los adapters se encargaron de realizar las transformaciones necesarias en los identificadores de los datos durante el proceso de migración. Además, desarrollamos excepciones específicas para manejar los casos en los que encontramos guiones medios, bajos y otros caracteres especiales en las URL de la API. Estas soluciones nos permitieron realizar una migración exitosa y asegurarnos de que los datos se mapearan correctamente en nuestra aplicación.

2. Problemas de Memoria en Dispositivos Android: Otro desafío importante que enfrentamos fue la limitada memoria disponible en dispositivos Android. Esto afectó el rendimiento y la estabilidad de nuestra aplicación, especialmente al cargar y mostrar contenido multimedia y al trabajar con grandes conjuntos de datos.

Solución: Para abordar los problemas de memoria, realizamos modificaciones en el archivo manifest de la aplicación. Ajustamos las configuraciones relacionadas con el manejo de memoria, como el límite máximo de memoria asignada a la aplicación y la administración de la memoria en segundo plano. Estas modificaciones nos permitieron optimizar el uso de la memoria y mejorar el rendimiento general de la aplicación en dispositivos con recursos limitados.

3. Implementación del RecyclerView y llamadas personalizadas a la API: En nuestro proyecto, utilizamos el componente RecyclerView para listar elementos en la interfaz de usuario. Sin embargo, surgieron dificultades al realizar llamadas personalizadas a la API para obtener información adicional para cada elemento de la lista. Debido a que estábamos utilizando ViewModel, nos encontramos con el problema de que todos los elementos compartían la misma instancia del ViewModel y, por lo tanto, se mostraba la misma información para cada elemento de la lista.

Solución: Para solucionar este problema, desarrollamos un ViewModel personalizado y utilizamos una interfaz como parámetro en el RecyclerView. De esta manera, logramos tener una instancia única de ViewModel para cada elemento de la lista y asegurarnos de que se mostrara la información correcta para cada elemento. La interfaz nos permitió comunicarnos entre el RecyclerView y el ViewModel personalizado, lo que resultó en una visualización precisa y actualizada de la información adicional para cada elemento de la lista.

4. Implementación de Fragments: los fragments no funcionan como cualquier otro activity. A veces presentan ciertas particularidades que hacen que su contenido

no reaccione como en un principio esperábamos. Son errores por lo general que son bastante sencillos de solucionar por lo que no ha supuesto un gran perjuicio o un excesivo retraso a la hora de continuar con el proyecto.

Solución: Para solucionar estos problemas hemos recurrido a las herramientas que nos ofrece internet para ver las soluciones aplicadas por otros desarrolladores en nuestra misma situación. Desde tutoriales en youtube, foros, páginas web como stackoverflow o incluso haciendo consultas a la IA chatgpt.

Durante el desarrollo de nuestro proyecto de creación de una aplicación, nos enfrentamos a diversos desafíos relacionados con la migración de datos de la API, los problemas de memoria en dispositivos Android y la implementación del RecyclerView. Sin embargo, mediante la implementación de soluciones como adapters y excepciones personalizadas, ajustes en el archivo manifest y la creación de un ViewModel personalizado con una interfaz para el RecyclerView, logramos superar estos obstáculos con éxito.

Estas soluciones no solo nos permitieron resolver los problemas identificados, sino que también mejoraron la calidad, el rendimiento y la estabilidad general de nuestra aplicación. Aprendimos la importancia de anticipar y abordar estos desafíos durante el proceso de desarrollo, así como la necesidad de adaptarnos y buscar soluciones creativas para garantizar el éxito del proyecto.

8. VALIDACIÓN DE LA SOLUCIÓN

8.1. DOCUMENTACIÓN DESCRIPTIVA

8.1.1 METODOLOGÍA TDD

Antes de comenzar con el desarrollo de la app el equipo de desarrollo decidió llevar a cabo el proyecto empleando una metodología llamada TDD o Test Driven Development.

El desarrollo basado en pruebas o TDD (test-driven development) es un método de programación que apuesta por el diseño de las pruebas antes de escribir el código de un programa o aplicación de software. Con esta alternativa, los programadores evitan que queden lagunas o aspectos sin abordar en un proceso de creación y desarrollo de software.

El nacimiento de TDD viene con la aparición del concepto test first que vino a dar la vuelta a la forma tradicional de programación, donde primero se desarrollaba el código, y luego se aplicaban los correspondientes tests. Con TDD, primero se diseñan las pruebas que debe pasar el programa, para después comenzar con su desarrollo.

Las ventajas del desarrollo dirigido por pruebas

El método TDD de desarrollo de software es muy popular debido a las múltiples ventajas que ofrece, entre las que podemos destacar:

- Detección temprana de errores.
- Identificación más sencilla de los errores que se han producido.
- Facilita el trabajo colaborativo entre distintos desarrolladores.
- Reduce el costo de mejoras y optimizaciones.
- Incrementa el nivel de calidad y seguridad del software.
- Elimina el “temor” a fallar en el proceso de desarrollo (al comenzar desarrollando pruebas para detectar los errores).

El funcionamiento de desarrollo bajo test-driven development implica la creación previa, de las pruebas que debe superar el software. Cuando se aplica esta metodología, no solo se diseñan pruebas y luego el código, sino que se crea un ciclo que envuelve el desarrollo en un conjunto para optimizar el código y conseguir programas con un mayor nivel de calidad y seguridad.

El ciclo de TDD empieza con la creación de la prueba, continúa con el desarrollo del código y luego el testeo de ese código. Si pasa la prueba se limpia, refactoriza y avanza a la siguiente fase. En caso de no superar la prueba, se pulse o modifica el código y se vuelve a realizar el test.

Este ciclo se suele representar en tres fases:

- Fase roja, donde el código no ha superado el test y necesita ser pulido para que vuelva a intentarlo.
- Fase verde, donde el código supera el test y se optimiza para que pase al siguiente ciclo de desarrollo.
- Refactoring. Es el proceso de limpiar y optimizar el código una vez se ha superado la prueba.

Cómo podemos ver, aunque las pruebas son fundamentales en esta metodología, se trata de un sistema de desarrollo cíclico, donde el código se va limpiando a medida que se van superando etapas.

Limitaciones que presenta la metodología

El desarrollo guiado por pruebas requiere que las pruebas puedan automatizarse. Esto resulta complejo en los siguientes dominios:

- Interfaces Gráfica de usuario (GUIs), aunque hay soluciones parciales propuestas.
- Objetos distribuidos, aunque los objetos simulados (*MockObjects*) pueden ayudar.
- Bases de datos. Hacer pruebas de código que trabaja con base de datos es complejo porque requiere poner en la base de datos unos datos conocidos antes de hacer las pruebas y verificar que el contenido de la base de datos es el esperado después de la prueba. Todo esto hace que la prueba sea costosa de codificar, aparte de tener disponible una base de datos que se pueda modificar libremente. Además, la prueba puede tardar demasiado en ejecutarse, lo que viola uno de los principios de TDD.

Estrategias para implementación de esta metodología

Uno de los puntos más delicados a la hora de aplicar TDD como herramienta de diseño es en el paso en el que ya tenemos un test que falla y debemos crear la implementación mínima para que el test pase. Para ello Kent Beck expone un conjunto de estrategias que nos van a permitir avanzar en pasos pequeños hacia la solución del problema.

Implementación falsa: Una vez que tenemos el test fallando, la forma más rápida de obtener la primera implementación es creando un fake que devuelva una constante. Esto nos ayudará a ir progresando poco a poco en la resolución del problema, ya que al tener la prueba pasando estamos listos para afrontar el siguiente caso.

Triangular: o la técnica de la triangulación, es el paso natural que sigue a la técnica de la implementación falsa. Es más, en la mayoría de los contextos, forma parte de la triangulación, basándose en lo siguiente:

- 1.Escoger el caso más simple que debe resolver el algoritmo.
- 2.Aplicar el algoritmo del TDD.
- 3.Repetir los pasos anteriores cubriendo las diferentes casuísticas.

Implementación obvia: cuando la solución parece muy sencilla, lo ideal es escribir la implementación obvia en las primeras iteraciones del ciclo *del TDD*.

Una característica de esta forma de programación es el evitar escribir código innecesario ("You Ain't Gonna Need It" (YAGNI)). Se intenta escribir el mínimo código posible, y si el código pasa una prueba aunque sepamos que es incorrecto nos da una idea de que tenemos que modificar nuestra lista de requisitos agregando uno nuevo.

La generación de pruebas para cada funcionalidad hace que el programador confíe en el código escrito. Esto permite hacer modificaciones profundas del código (posiblemente en una etapa de mantenimiento del programa) pues sabemos que si luego logramos hacer pasar todas las pruebas tendremos un código que funcione correctamente.

Otra característica del Test Driven Development es que requiere que el programador primero haga fallar los casos de prueba. La idea es asegurarse de que los casos de prueba realmente funcionen y puedan recoger un error.

8.1.2 MODELO PARA POKELYTICS

En los últimos años han surgido diferentes enfoques sobre cómo organizar los proyectos de desarrollo Android:

La comunidad se ha alejado del patrón MVC (Model View Controller) para dar cabida a patrones más modulares, y que pueden ser testeados con mayor facilidad.

Model View Presenter (MVP) y Model View ViewModel (MVVM) son 2 de las alternativas más ampliamente adoptadas. Sin embargo, los desarrolladores siempre discuten sobre qué enfoque es mejor para el desarrollo Android.

En los últimos años se han publicado varios artículos, donde se aboga que un enfoque es superior al otro.

Sin embargo, muchas veces las opiniones están influenciadas por las propias preferencias del autor, quienes no siempre presentan su crítica de manera objetiva.

En el caso de Pokelytics, el patrón MVVM ha sido el elegido como enfoque de organización del proyecto ya que uno de los miembros del equipo de desarrollo es conocedor de las ventajas que esto conlleva.

8.1.2.1 MVVM

MVVM, con DATA BINDING en Android, tiene los beneficios de facilitar las pruebas y la modularidad, reduciendo a su vez la cantidad de código que tenemos que escribir para conectar vista + modelo.

Examinemos las partes de MVVM.

8.1.2.2 MODEL

Lo mismo que para MVC y MVP. No hay cambios.

8.1.2.3 VIEW

La vista se vincula con variables "observables" y "acciones" expuestas por el VIEWMODEL de forma flexible.

8.1.2.4 VIEWMODEL

Es el responsable de ajustar el modelo y preparar los datos observables que necesita la vista. También proporciona hooks para que la vista pase eventos al modelo. Sin embargo, el ViewModel no está vinculado a la vista.

8.1.2.5 REPRESENTACIÓN

Aquí un desglose a alto nivel para nuestra aplicación Tic Tac Toe.

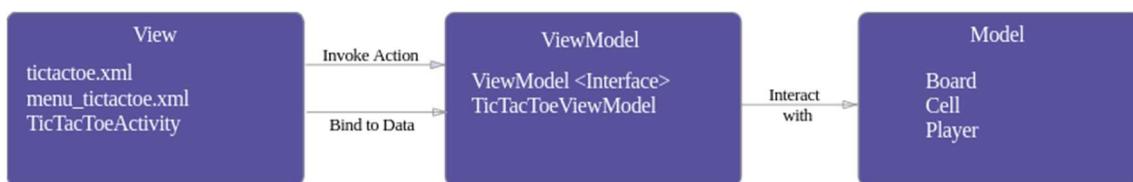


Imagen 22 Diagrama MVVM

Echemos un vistazo más de cerca a las partes que aquí intervienen, empezando por nuestra clase ViewModel.

```
public class TicTacToeViewModel implements ViewModel {  
  
    private Board model;  
  
    /*  
     * Estas son variables observables que el ViewModel actualizará según corresponda.  
     * Los componentes visuales están vinculados directamente a estos objetos y  
     * reaccionan ante cambios  
     * inmediatamente, sin que el ViewModel les diga qué hacer. No tienen que tener  
     * visibilidad public; pueden ser private con un método getter que facilite el acceso.  
     */  
  
    public final ObservableArrayMap<String, String> cells = new  
    ObservableArrayMap<>();  
    public final ObservableField<String> winner = new ObservableField<>();
```

```

public TicTacToeViewModel() {
    model = new Board();
}

// Tal como ocurre con el presenter, implementamos métodos asociados al ciclo de
vida,
// en caso que necesitemos hacer algo con nuestros modelos durante tales eventos.
public void onCreate() {}
public void onPause() {}
public void onResume() {}
public void onDestroy() {}

/**
 * Una acción (Action), llamable desde la vista. Esta acción comunica al modelo
 * sobre qué celda se hizo clic, y actualiza los campos observables con
 * el estado actual del modelo.
 */
public void onClickedCellAt(int row, int col) {
    Player playerThatMoved = model.mark(row, col);
    cells.put(" " + row + col, playerThatMoved == null ?
        null : playerThatMoved.toString());
    winner.set(model.getWinner() == null ? null : model.getWinner().toString());
}

/**
 * Otra acción (Action), llamable desde la vista. Esta acción solicita al modelo
 * resetear su estado, y limpia la data observable en este ViewModel.
 */
public void onResetSelected() {
    model.restart();
    winner.set(null);
    cells.clear();
}

}

```

A continuación, una parte del XML, para ver cómo se vinculan las variables y acciones en la vista:

```
<!--  
    Con Data Binding, el elemento raíz es <layout>. Y contiene 2 partes en su interior.  
    1. <data> - Definimos aquí las variables sobre las que vamos a definir nuestras  
binding expressions, e  
        importamos las clases que podríamos necesitar como referencia, tal como  
        android.view.View.  
    2. <root layout> - Este es el elemento raíz que usaremos como layout de nuestra  
vista.  
        Es la etiqueta xml raíz equivalente a los ejemplos de MVC y MVP.  
-->  
<layout xmlns:android="http://schemas.android.com/apk/res/android"  
    xmlns:tools="http://schemas.android.com/tools"  
    xmlns:app="http://schemas.android.com/apk/res-auto">  
  
    <!-- Vamos a hacer referencia a TicTacToeViewModel con el nombre viewModel, tal  
como lo definimos aquí. -->  
    <data>  
        <import type="android.view.View" />  
        <variable  
            name="viewModel"  
            type="com.acme.tictactoe.viewmodel.TicTacToeViewModel" />  
    </data>  
    <LinearLayout...>  
        <GridLayout...>  
            <!-- Evento onClick para cada celda del tablero. Cada botón invoca al método  
            onClickedCellAt con los valores row y col correspondientes.  
            El valor mostrado proviene del ObservableArrayMap definido en el  
            ViewModel. -->  
            <Button  
                style="@style/tictactoebutton"  
                android:onClick="@{() -> viewModel.onClickedCellAt(0,0)}"  
                android:text='@{viewModel.cells["00"]}' />  
            ...  
            <Button  
                style="@style/tictactoebutton"  
                android:onClick="@{() -> viewModel.onClickedCellAt(2,2)}"  
                android:text='@{viewModel.cells["22"]}' />  
        </GridLayout>
```

```
<!-- La visibilidad del LinearLayout que muestra al ganador depende de si "winner"  
es null o no.
```

Se debe tener cuidado de no agregar lógica de presentación a la vista. Para este caso

tiene sentido que el valor de "visibility" dependa de la expresión. Sería extraño que la vista renderice

esta sección si el valor de winner está vacío. -->

```
<LinearLayout...
```

```
    android:visibility="@{viewModel.winner != null ? View.VISIBLE : View.GONE}"  
    tools:visibility="visible">
```

```
<!-- El valor de este texto está vinculado a viewModel.winner y reacciona si dicho  
valor cambia. -->
```

```
    <TextView  
        ...  
        android:text="@{viewModel.winner}"  
        tools:text="X" />  
    ...  
    </LinearLayout>  
</LinearLayout>  
</layout>
```

Los atributos "tools" sobreescriben valores a fin de previsualizar elementos. Los hemos usado en el ejemplo anterior para mostrar un valor para "winner" y para "visibility". Sin ellos sería complicado ver qué está sucediendo mientras se diseña.

Y una nota al margen de lo que estamos viendo:

Esto es sólo una pequeña muestra de todo lo que se puede hacer con Data Binding. Te recomiendo consultar la documentación de Data Binding para aprender más acerca de esta gran herramienta. También hay un enlace al final de esta página, hacia el proyecto Google Android Architecture Blueprints para que puedas consultar más ejemplos de MVVM y Data Binding.

8.1.3 INYECCIÓN DE DEPENDENCIAS

La inyección de dependencias (DI) es una técnica muy utilizada en programación y adecuada para el desarrollo de Android. Si sigues los principios de la DI, sentarás las bases para una buena arquitectura de apps.

Implementar la inyección de dependencias te proporciona las siguientes ventajas:

- Reutilización de código
- Facilidad de refactorización
- Facilidad de prueba

8.1.3.1 ASPECTOS BÁSICOS DE LA INYECCIÓN DE DEPENDENCIAS

Antes de analizar específicamente la inyección de dependencias en Android, en esta página se proporciona una descripción más general de cómo funciona la inyección de dependencias.

Las clases suelen requerir referencias a otras clases. Por ejemplo, una clase Car podría necesitar una referencia a una clase Engine. Estas clases se llaman **DEPENDENCIAS** y, en el ejemplo, la clase Car necesita una instancia de la clase Engine, de la que depende para ejecutarse.

Una clase puede obtener un objeto que necesita de tres maneras distintas:

1. La clase construye la dependencia que necesita. En el ejemplo anterior, Car crea e inicializa su propia instancia de Engine.
2. La toma de otro lugar. Algunas API de Android, como los métodos get de Context y getSystemService(), funcionan de esta manera.
3. La recibe como parámetro. La app puede proporcionar estas dependencias cuando se construye la clase o pasárlas a las funciones que necesitan cada dependencia. En el ejemplo anterior, el constructor Car recibe Engine como parámetro.

La tercera opción es la inyección de dependencias. Con este enfoque, tomas las dependencias de una clase y las proporcionas en lugar de hacer que la instancia de la clase las obtenga por su cuenta.

Por ejemplo: Sin inyección de dependencias, la representación de un Car que crea su propia dependencia Engine se ve así en el código:

```
class Car {  
  
    private val engine = Engine()  
  
    fun start() {  
        engine.start()  
    }  
}
```

```

}

fun main(args: Array<String>) {
    val car = Car()
    car.start()
}

```

Este no es un ejemplo de inyección de dependencias, porque la clase Car está construyendo su propio Engine, lo que puede ser problemático debido a lo siguiente:

- Car y Engine están estrechamente vinculados: una instancia de Car usa un tipo de Engine, y no se pueden utilizar subclases ni implementaciones alternativas con facilidad. Si el Car construyera su propio Engine, tendrías que crear dos tipos de Car en lugar de solo reutilizar el mismo Car para motores de tipo Gas y Electric.
- La dependencia estricta de Engine hace que las pruebas sean más difíciles. Car usa una instancia real de Engine, lo que impide utilizar un doble de prueba y modificar Engine para diferentes casos de prueba.

¿Cómo se ve el código con la inyección de dependencias? En lugar de que las diferentes instancias de Car construyan su propio objeto Engine durante la inicialización, cada una recibe un objeto Engine como parámetro en su constructor:

Kotlin

```

class Car(private val engine: Engine) {
    fun start() {
        engine.start()
    }
}

fun main(args: Array<String>) {
    val engine = Engine()
    val car = Car(engine)
    car.start()
}

```

La función main usa Car. Debido a que Car depende de Engine, la app crea una instancia de Engine y, luego, la usa para construir una instancia de Car. Los beneficios de este enfoque basado en DI son los siguientes:

- Reutilización de Car. Puedes pasar diferentes implementaciones de Engine a Car. Por ejemplo, puedes definir una nueva subclase de Engine, llamada ElectricEngine, para utilizar con Car. Si usas DI, solo debes pasar una instancia de la subclase actualizada de ElectricEngine y Car seguirá funcionando sin más cambios.
- Prueba fácil de Car. Puedes pasar dobles de prueba para probar diferentes situaciones. Por ejemplo, puedes crear un doble de prueba de Engine, llamado FakeEngine, y configurarlo para diferentes pruebas.

Existen dos formas principales de realizar la inyección de dependencias en Android:

- Inyección de constructor. Esta es la manera descrita anteriormente. Pasas las dependencias de una clase a su constructor.
- Inyección de campo (o inyección de método set). El sistema crea instancias de ciertas clases de framework de Android, como actividades y fragmentos, por lo que no es posible implementar la inyección de constructor. Con la inyección de campo, se crean instancias de dependencias después de crear la clase. El código se vería así:

```
class Car {  
    lateinit var engine: Engine  
  
    fun start() {  
        engine.start()  
    }  
}  
  
fun main(args: Array) {  
    val car = Car()  
    car.engine = Engine()  
    car.start()  
}
```

8.1.3.2 INYECCIÓN DE DEPENDENCIAS AUTOMATIZADA

En el ejemplo anterior, creaste, proporcionaste y administraste por tu cuenta las dependencias de las diferentes clases, sin recurrir a una biblioteca. Esto se denomina inyección de dependencias manual. En el ejemplo de Car, solo había una dependencia, pero, si hay varias dependencias y clases, la inyección manual puede resultar más tediosa. Además, la inyección de dependencias manual presenta varios problemas:

- En el caso de aplicaciones grandes, tomar todas las dependencias y conectarlas correctamente puede requerir una gran cantidad de código estándar. En una arquitectura de varias capas, para crear un objeto en una capa superior, debes proporcionar todas las dependencias de las capas que se encuentran debajo de ella. Por ejemplo, para construir un automóvil real, es posible que necesites un motor, una transmisión, un chasis y otras piezas; a su vez, el motor necesita cilindros y bujías.
- Cuando no puedes construir dependencias antes de pasárlas (por ejemplo, si usas inicializaciones diferidas o solicitas permisos para objetos en los flujos de tu app), necesitas escribir y conservar un contenedor personalizado (o un grafo de dependencias) que administre las dependencias en la memoria desde el principio.

Hay bibliotecas que resuelven este problema automatizando el proceso de creación y provisión de dependencias. Se dividen en dos categorías:

- Soluciones basadas en reflexiones que conectan las dependencias durante el tiempo de ejecución.
- Soluciones estáticas que generan el código para conectar las dependencias durante el tiempo de compilación.

Dagger es una biblioteca de inserción de dependencias popular para Java, Kotlin y Android que mantiene Google. Dagger facilita el uso de la DI en tu app mediante la creación y administración del grafo de dependencias. Proporciona dependencias totalmente estáticas y en tiempo de compilación que abordan muchos de los problemas de desarrollo y rendimiento de las soluciones basadas en reflexiones, como Guice.

8.1.3.3 ALTERNATIVAS A LA INSERCIÓN DE DEPENDENCIAS

Una alternativa a la inserción de dependencias es usar un localizador de servicios. El patrón de diseño del localizador de servicios también mejora el desacoplamiento de clases de las dependencias concretas. En este procedimiento, creas una clase conocida como LOCALIZADOR DE SERVICIOS que, a su vez, crea y almacena dependencias, y luego proporciona esas dependencias a pedido.

```
object ServiceLocator {  
    fun getEngine(): Engine = Engine()  
}  
  
class Car {  
    private val engine = ServiceLocator.getEngine()  
  
    fun start() {  
        engine.start()  
    }  
}  
  
fun main(args: Array<String>) {  
    val car = Car()  
    car.start()  
}
```

El patrón del localizador de servicios se diferencia de la inyección de dependencias en la forma en que se consumen los elementos. Con el patrón del localizador de servicios, las clases tienen el control y solicitan que se inyecten objetos. Con la inyección de dependencias, la app tiene el control e inyecta los objetos solicitados de manera proactiva.

En comparación con la inyección de dependencias:

- La colección de dependencias que requiere un localizador de servicios hace que el código sea más difícil de probar, ya que todas las pruebas tienen que interactuar con el mismo localizador de servicios global.

- Las dependencias se codifican en la implementación de la clase, no en la superficie de la API. De esta manera, es más difícil saber qué necesita una clase del exterior. Por ese motivo, los cambios en Car o las dependencias disponibles en el localizador de servicios pueden provocar que fallen las referencias, y así generar errores durante el tiempo de ejecución o en las pruebas.
- Administrar los ciclos de vida de los objetos es más difícil si no quieras establecer alcances que abarquen el ciclo de vida completo de toda la app.

8.1.4 HILT

Hilt es la biblioteca de Jetpack recomendada para la inserción de dependencias en Android. Hilt establece una forma estándar de usar la inserción de dependencias en tu aplicación, ya que proporciona contenedores para cada clase de Android en tu proyecto y administra automáticamente sus ciclos de vida.

Hilt se basa en la popular biblioteca de inserción de dependencias Dagger y se beneficia de la corrección en tiempo de compilación, el rendimiento del entorno de ejecución, la escalabilidad y la compatibilidad con Android Studio que proporciona.

Para obtener más información sobre Hilt, consulta [Inserción de dependencias con Hilt](#).

Para comprender por completo los beneficios de la inserción de dependencias, debes probarla de forma manual en tu app, como se muestra en la sección [Inserción de dependencias manual](#).

Para llevar a cabo el proyecto Pokelytics el equipo de desarrollo ha enfocado el proyecto de una determinada forma escogiendo determinadas herramientas, metodologías u otros aspectos que han condicionado la solución final presentada.

Vamos a continuar hablando de inyección de dependencias. Para la inyección de dependencias. La inyección de dependencias nos ayuda con la creación de objetos y nos permite evitar la dependencia entre clases. Esta técnica reduce las opciones para escribir código repetitivo (boilerplate code) y también hace que el proceso de desarrollo sea mucho más fluido. Por lo tanto, algunos de los beneficios de usar la inyección de dependencias en nuestros proyectos son (Sinhala, 2017):

- Acoplamiento débil (loose coupling): tu código depende menos de otras clases. Por lo tanto, cuantas menos responsabilidades tenga un fragmento de código, menos propenso al error es.
- Código fácilmente testeable: debido al hecho de que no estás creando un objeto en un método, puedes crear objetos de prueba (mock objects) para testear tu código.
- Reutilización del código: cuando implementas la inyección de dependencias, inicias tu código una vez. Luego, puedes pasar ese objeto a cualquier parte del código que lo requiera. Esto le da a tu código más posibilidades de ser reutilizado.

Para esta tarea con frecuencia tenemos dos opciones: Koin o Dagger.

Dagger está principalmente compuesto por módulos y componentes. Para incluirlos en tu código, debes usar las anotaciones `@Module` y `@Component` y a su vez, Dagger utiliza otras anotaciones útiles como `@Inject` y `@Provides` (hablaremos de ellas en los párrafos siguientes), así como también `@Scope`, `@Named`, `@Qualifier`, etc.

Koin es un framework de inyección de dependencias simple pero poderoso. Hace la vida de los desarrolladores más sencilla al reducir mucho del código repetitivo (boilerplate code) que sí es necesario escribir en otros frameworks, como Dagger. Está escrito en Kotlin, y consiste en una biblioteca pequeña y simple que hace de la inyección de dependencias una tarea sencilla.

Koin está escrito en Kotlin en su totalidad y, a pesar del hecho de que la biblioteca no es compatible con Google, está ganando adeptos a un ritmo increíble. Según sus creadores, Koin es una alternativa ligera para suplantar a otros frameworks más pesados o robustos. Técnicamente, Koin no implementa el sistema de inyección de dependencias, sino que implementa el patrón service locator.

8.1.5 ¿KOIN VS DAGGER?

Tanto Dagger como Koin son frameworks de inyección de dependencias muy buenos. Dagger tiene la ventaja de que ha estado en el mercado por más tiempo que Koin. Sin embargo, Koin está ganando adeptos a un ritmo acelerado. Si eres un desarrollador de Kotlin, te será más sencillo empezar con Koin en lugar de Dagger. Sin embargo, hay más información sobre Dagger en blogs, en artículos, y en preguntas en Stack Overflow. Una de las desventajas de Dagger es que la biblioteca crea clases intermedias que aumentan el tamaño de tu apk (Android Application Package). Si eso es algo que te preocupa, entonces se trata de otro punto a favor para Koin, ya que este último no crea clases intermedias. Finalmente, podemos decir que Koin es una gran alternativa cuando se trata de proyectos pequeños. En los proyectos grandes, existe riesgo de sobrecoste ya que la generación de códigos Koin puede exceder las posibles ventajas del framework. Por lo tanto, los frameworks con más años y más maduros como Dagger son recomendables a la hora de realizar proyectos grandes con un equipo distribuido. Sin embargo, Koin parece ser una alternativa excelente para aplicaciones pequeñas y medianas (Stoltman, 2018). Finalmente, la decisión entre uno u otro va a depender de qué tan fácilmente se adapte la biblioteca elegida a tu estilo de código y qué tan cómodo te sientas tú o se sienta tu equipo al usarla.

Diferencias entre inyección de dependencias y service locator

La inyección de dependencias y el service locator son dos patrones de diseño utilizados en el desarrollo de software para gestionar las dependencias entre componentes. Ambos enfoques tienen como objetivo principal mejorar la modularidad, la flexibilidad y la testabilidad de una aplicación. Sin embargo, existen diferencias significativas entre ellos en cuanto a su implementación y filosofía subyacente.

La inyección de dependencias (DI, por sus siglas en inglés) es un patrón de diseño que se basa en el principio de inversión de dependencias. En este enfoque, las dependencias de un componente se proporcionan externamente, en lugar de ser creadas o gestionadas por el propio componente. En otras palabras, en lugar de que un

objeto cree y mantenga sus propias dependencias, estas se inyectan desde fuera, a menudo a través de un contenedor de inyección de dependencias.

La inyección de dependencias se basa en la idea de que un componente no debería ser responsable de conocer y crear todas sus dependencias, lo cual aumentaría el acoplamiento y dificultaría las pruebas unitarias. En cambio, se delega la responsabilidad de crear y suministrar las dependencias a un contenedor de inyección de dependencias, que actúa como intermediario entre los componentes y sus dependencias. El contenedor resuelve las dependencias y las inyecta en los componentes que las necesitan.

Este enfoque promueve la modularidad y la reutilización de componentes, ya que los componentes no están fuertemente acoplados a las implementaciones concretas de sus dependencias. Además, facilita las pruebas unitarias al permitir la sustitución de dependencias reales por objetos simulados o "mocks" durante las pruebas.

Por otro lado, el service locator es otro patrón de diseño utilizado para gestionar las dependencias, pero se basa en un enfoque diferente. En lugar de inyectar directamente las dependencias en los componentes, el service locator proporciona un mecanismo centralizado para localizar y obtener las dependencias cuando son necesarias.

En el service locator, los componentes dependientes no conocen directamente las implementaciones concretas de sus dependencias. En su lugar, solicitan las dependencias al service locator utilizando una interfaz común. El service locator, a su vez, es responsable de conocer las implementaciones concretas de las dependencias y proporcionarlas a los componentes cuando se solicitan.

A diferencia de la inyección de dependencias, donde las dependencias se definen y resuelven externamente, en el service locator, las dependencias se resuelven internamente dentro del service locator. Esto significa que los componentes dependientes están acoplados al service locator y conocen su existencia. Esto puede introducir acoplamiento adicional y hacer que el código sea menos modular y más difícil de probar.

Otra diferencia clave entre ambos enfoques radica en el momento de la resolución de dependencias. En la inyección de dependencias, la resolución de dependencias se realiza en tiempo de compilación o en tiempo de ejecución temprano, lo que permite detectar errores de configuración o de dependencias faltantes de manera más temprana. En el service locator, la resolución de dependencias se realiza en tiempo de ejecución, lo que implica un mayor riesgo de errores en tiempo de ejecución relacionados con la disponibilidad de las dependencias.

Se podría decir que la inyección de dependencias y el service locator son dos enfoques diferentes para gestionar las dependencias en una aplicación. La inyección de dependencias se basa en el principio de inversión de dependencias y promueve la modularidad y la reutilización de componentes al inyectar las dependencias externamente. Por otro lado, el service locator proporciona un mecanismo centralizado para localizar y obtener las dependencias cuando son necesarias. Si bien ambos enfoques tienen sus ventajas y desventajas, la elección entre ellos dependerá de las necesidades y requisitos específicos del proyecto en cuestión.

8.1.6 RETROFIT

Además de todo lo anteriormente mencionado, nuestro proyecto se nutre de una API, pero para poder acceder a toda la información contenida en ella, hemos de hacerle peticiones. Para estas peticiones empleamos Retrofit.

Retrofit es una biblioteca de Android desarrollada por Square que facilita la implementación de peticiones HTTP en nuestras aplicaciones.

Esta biblioteca es muy popular y utilizada por desarrolladores de todo el mundo debido a su simplicidad y facilidad de uso.

En este artículo, veremos cómo utilizar Retrofit para crear una conexión a una API de películas y realizar operaciones CRUD (crear, leer, actualizar y eliminar) utilizando Kotlin como lenguaje de programación.

Para empezar, necesitamos agregar la dependencia de Retrofit y del adapter de Gson en nuestro archivo build.gradle:

```
implementation 'com.squareup.retrofit2:retrofit:2.7.2'
```

```
implementation 'com.squareup.retrofit2:converter-gson:2.4.0'
```

A continuación, creamos una interfaz que define los métodos que utilizaremos para realizar las operaciones CRUD en la API de películas. En este ejemplo, utilizaremos la API de The Movie Database (TMDb):

```
interface MovieAPI {  
  
    // Método para crear una película  
  
    @POST("movie")  
  
    fun createMovie(@Body movie: Movie): Call<Movie>  
  
    // Método para obtener todas las películas  
  
    @GET("movie")  
  
    fun getMovies(): Call<List<Movie>>  
  
    // Método para obtener una película por su ID  
  
    @GET("movie/{id}")  
  
    fun getMovie(@Path("id") id: Int): Call<Movie>  
  
    // Método para actualizar una película  
  
    @PUT("movie/{id}")  
  
    fun updateMovie(@Path("id") id: Int, @Body movie: Movie): Call<Movie>
```

```

// Método para eliminar una película

@DELETE("movie/{id}")

fun deleteMovie(@Path("id") id: Int): Call<Void>

}

```

En cada uno de estos métodos, especificamos el tipo de petición HTTP que utilizaremos (POST, GET, PUT o DELETE) y la ruta de la API donde se realizará la petición.

Además, en los métodos que reciben o envían una película, utilizamos la anotación @Body para indicar que el cuerpo de la petición será el objeto de tipo Movie.

Ahora, creamos una clase que implemente esta interfaz y que utilice Retrofit para realizar las peticiones HTTP:

```

class MovieService {

    // Instancia de Retrofit

    private val retrofit = Retrofit.Builder()

        .baseUrl("https://api.themoviedb.org/3/")

        .addConverterFactory(GsonConverterFactory.create())

        .build()

    // Método para crear una película

    fun createMovie(movie: Movie) {

        movieAPI.createMovie(movie).enqueue(object : Callback<Movie> {

            override fun onResponse(call: Call<Movie>, response: Response<Movie>) {

                // Procesar respuesta exitosa

            }

            override fun onFailure(call: Call<Movie>, t: Throwable) {

                // Procesar error en la petición

            }
        })
    }
}

```

```
}

// Método para obtener todas las películas

fun getMovies() {

    movieAPI.getMovies().enqueue(object : Callback<List<Movie>> {

        override fun onResponse(call: Call<List<Movie>>, response: Response<List<Movie>>) {

            // Procesar respuesta exitosa

        }

        override fun onFailure(call: Call<List<Movie>>, t: Throwable) {

            // Procesar error en la petición

        }
    })
}

// Método para obtener una película por su ID

fun getMovie(id: Int) {

    movieAPI.getMovie(id).enqueue(object : Callback<Movie> {

        override fun onResponse(call: Call<Movie>, response: Response<Movie>) {

            // Procesar respuesta exitosa

        }

        override fun onFailure(call: Call<Movie>, t: Throwable) {

            // Procesar error en la petición

        }
    })
}
```

```
// Método para actualizar una película

fun updateMovie(id: Int, movie: Movie) {

    movieAPI.updateMovie(id, movie).enqueue(object : Callback<Movie> {

        override fun onResponse(call: Call<Movie>, response: Response<Movie>) {

            // Procesar respuesta exitosa

        }

        override fun onFailure(call: Call<Movie>, t: Throwable) {

            // Procesar error en la petición

        }
    })
}

// Método para eliminar una película

fun deleteMovie(id: Int) {

    movieAPI.deleteMovie(id).enqueue(object : Callback<Void> {

        override fun onResponse(call: Call<Void>, response: Response<Void>) {

            // Procesar respuesta exitosa

        }

        override fun onFailure(call: Call<Void>, t: Throwable) {

            // Procesar error en la petición

        }
    })
}
}
```

En esta clase, utilizamos Retrofit para crear una instancia de la interfaz MovieAPI. Luego, en cada uno de los métodos, utilizamos esta instancia para realizar la petición correspondiente a la operación CRUD deseada. También utilizamos el método enqueue() para ejecutar la petición en un hilo en segundo plano y recibir la respuesta en un callback.

Por último, en nuestro código podemos utilizar esta clase de la siguiente manera para realizar las operaciones CRUD en la API de películas:

```
val movieService = MovieService()

// Crear una película

val movie = Movie(...)

movieService.createMovie(movie)

// Obtener todas las películas

movieService.getMovies()

// Obtener una película por su ID

val id = 1

movieService.getMovie(id)

// Actualizar una película

val updatedMovie = Movie(...)

movieService.updateMovie(id, updatedMovie)

// Eliminar una película

movieService.deleteMovie(id)
```

8.1.6.1 RETROFIT TIENE SOPORTE PARA CORRUTINAS

Si utilizamos el soporte oficial de la librería Retrofit para corrutinas, el código cambia de la siguiente manera.

La interfaz MovieAPI se modifica para crear funciones suspend:

```
interface MovieAPI {

    // Método para crear una película

    @POST("movie")

    suspend fun createMovie(@Body movie: Movie): Movie
```

```

// Método para obtener todas las películas

@GET("movie")

suspend fun getMovies(): List<Movie>

// Método para obtener una película por su ID

@GET("movie/{id}")

suspend fun getMovie(@Path("id") id: Int): Movie

// Método para actualizar una película

@PUT("movie/{id}")

suspend fun updateMovie(@Path("id") id: Int, @Body movie: Movie): Movie

// Método para eliminar una película

@DELETE("movie/{id}")

suspend fun deleteMovie(@Path("id") id: Int): Void

}

```

Y la clase MovieService también tendrá métodos suspend. Un ejemplo:

```

// Método para obtener todas las películas

suspend fun getMovies(): List<Movie> {

    return movieAPI.getMovies()

}

```

En nuestro código, podemos utilizar esta clase de la siguiente manera:

```

// Obtener todas las películas

GlobalScope.launch(Dispatchers.Main) {

    val movies = movieService.getMovies()

    // Procesar lista de películas

}

```

Retrofit nos permite implementar peticiones HTTP de manera sencilla y eficiente en nuestras aplicaciones de Android utilizando Kotlin.

Con esta biblioteca, podemos conectar nuestra aplicación a cualquier API y realizar operaciones CRUD de manera rápida y sencilla.

8.1.7 CLEAN CODE: DRY, SOLID

Una vez definida una arquitectura toca empezar a desarrollar componentes, libs, modelos, interfaces, servicios, hasta aquí bien, ahora bien lo más difícil es que sea mantenible, y la responsabilidad en realidad no es del arquitecto que diseño la primera aproximación de arquitectura, en realidad es de los desarrolladores.

El libro para aprender todo sobre clean code es: Clean Code by Robert C.Martin.

8.1.7.1 QUE ES UNA ARQUITECTURA MANTENIBLE?

Tiene que tener

- Todos los artefactos fáciles de encontrar, su contenido tiene que ser el esperado y su código fácil y sencillo de leer.
- Todos los artefactos tienen que seguir las convenciones de clean code y SOLID.
- Alta cobertura de test unitario, los test junto con los mocks nos van a describir lo que hace cada componente, y es su mejor documentación.

8.1.7.2 COSTES

La gran versatilidad de Javascript ha derivado en algunas malas prácticas, de ahí que haya que vigilar los costes derivados del mantenimiento del software (que suele ser más alto que el del desarrollo).

El coste del mantenimiento viene dado por los costes (tiempo) de entenderlo, cambiarlo, testearlo y desplegarlo.

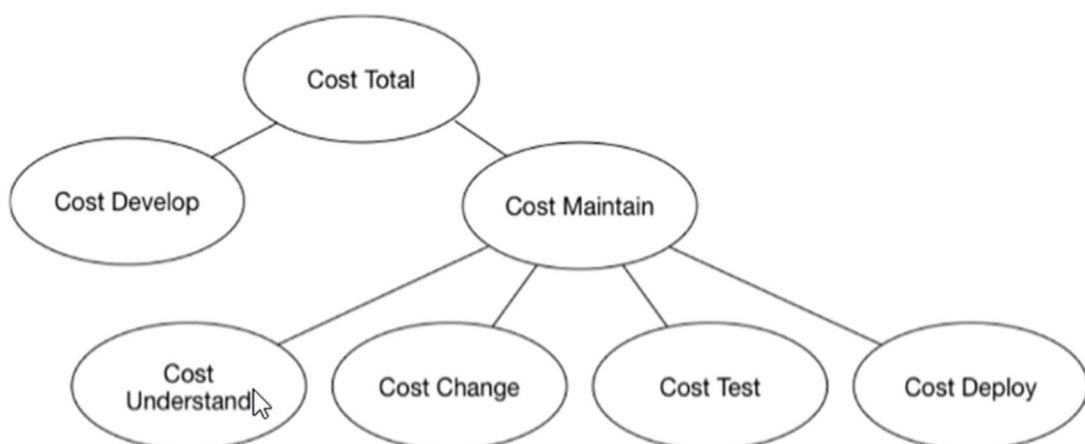


Imagen 23 Esquema de costes de Kent Beck

8.1.7.2 DEUDA TÉCNICA.

La falta de calidad de código genera una deuda que acabar generando sobre costes. Muchos programadores estudian a fondo una librería y patrones, pero hay pocos que lo combinen con Clean Code y SOLID.

8.1.7.3 TOLERANCIA AL CAMBIO DE NUESTRO CÓDIGO

Si tenemos una deuda técnica nula, nuestro código sea muy tolerante al cambio, eso nos va a permitir arreglar bugs y ampliar funcionalidad ágilmente .

8.1.7.4 REGLAS DEL DISEÑO SIMPLE

Para prevenir la deuda técnica nada mejor que cumplir las cuatro reglas del diseño simple de Kent Beck

- El código pasa correctamente los tests.
- Revela la intención del diseño.
- Respeta el principio DRY.
- Tiene el menor número posible de elementos.

8.1.7.5 CONVENCIONES DE CLEAN CODE

Código limpio es el código que ha sido escrito con la intención de que otra persona lo entienda.

- Ausencia de información técnica y según el tipo de dato
- Funciones: Sabemos que estamos desarrollando código limpio cuando cada función hace exactamente lo que su nombre indica, tienen que resultar fáciles de leer, entender y tiene que transmitir su intención. Deben tener máximo 4/5 líneas pudiendo excederlo en causas excepcionales. Limitar el número de parámetros a 3 y para más de 3 mejor usar un objeto como parámetro de entrada con desestructuración.

Al final lo más importante es el nombre que le asignas a todo, a las carpetas, los componentes, las funciones, las variables, los imports ...

Y si a eso le juntas que el código hace lo que describía su título (SRP), una maravilla!.

En este ejemplo podemos ver como la función check login es muy ambigua, no acaba de dejar claro si tiene un retorno o que va a hacer, mejor crear una función que nos diga si esta logado o no.

```

public function check_login()
{
    if(!$_SESSION['uid']){
        redirect( uri: "login");
    }
}

public function isLoggedIn()
{
    if(isset($_SESSION['uid']) && !empty($_SESSION['uid'])){
        return true;
    }

    return false;
}

```

Imagen 24 Funciones de Clean Code

Cuando hacemos imports es muy importante que las funciones importadas describan perfectamente no solamente lo que van a hacer sino lo que son, p.ej: closeModalAction (un action que nos va a cerrar el popup), o isAzureLoginModalOpenSelector (un selector con un boolean que nos va a indicar si el popup de login está abierto)

```

src > containers > PageAzureLogin > AzureLoginModal > index.tsx > ...
1  import * as React from "react";
2  import { connect } from "react-redux";
3  import asModal from "../../components/Modal";
4  import { injectIntl, InjectedIntlProps } from "react-intl";
5  import { closeModalAction } from "../../store/modals/action";
6  import { AppState } from "../../reducers";
7  import RootAction from "../../store/ActionType";
8  import { isAzureLoginModalOpenSelector } from "../../selectors/modalSelector";
9  import buttonStyles from "../../css/buttonStyles.module.css";
10 import styles from "./styles.module.css";

```

Imagen 25 Imports

Igual de importante es definir nuestro código lo más cercano posible a nuestro store, que sea facil de identificar nuestros actions, model y selectore de redux con nuestro store real, p.ej este código:

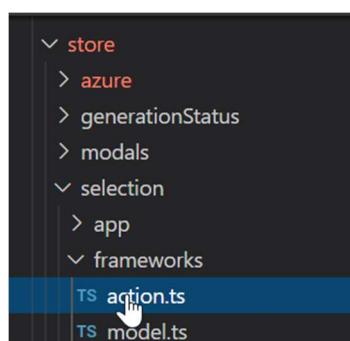
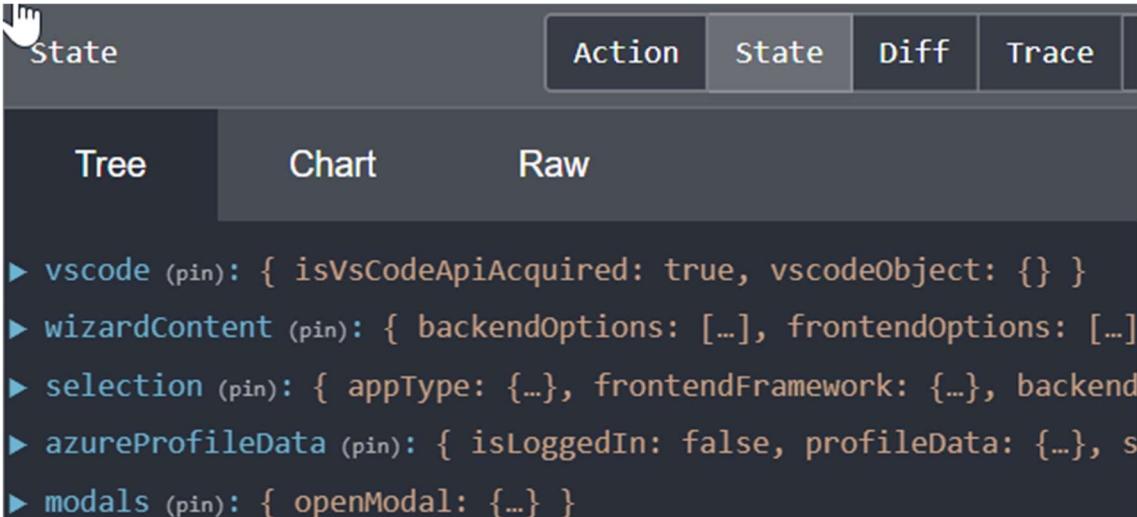


Imagen 26 Árbol de directorios

Luego se pueden localizar en el store sin problema:



The screenshot shows the React DevTools State tab. At the top, there are tabs for Action, state, Diff, and Trace, with Action being the active tab. Below that, there are three sub-tabs: Tree (selected), Chart, and Raw. The main area displays a list of state keys with their types and initial values. The list includes:

- vscode (pin): { isVsCodeApiAcquired: true, vscodeObject: {} }
- wizardContent (pin): { backendOptions: [...], frontendOptions: [...] }
- selection (pin): { appType: {...}, frontendFramework: {...}, backend: ... }
- azureProfileData (pin): { isLoggedIn: false, profileData: {...}, selectedProfile: ... }
- modals (pin): { openModal: {...} }

Imagen 27 Store

Aunque eso sí, los code reviews con mucho tacto, lo mejor al principio es ver código del propio ponente, reírnos de nuestro propio código y como ya no lo defendemos ni pasados 15 días del commit, así podemos dar una visión global, pero lo más importante es hacer talleres donde cada desarrollador se vea obligado a separar competencias de código, elegir nombres, etc ... y mucha calma, obligar a tu equipo a escribir código clean y solid es modificar la manera de desarrollar de muchos años, se tarda tiempo.

8.1.8 CÓMO GUARDAR CONTENIDO EN UNA BASE DE DATOS LOCAL CON ROOM

Las apps que controlan grandes cantidades de datos estructurados pueden beneficiarse con la posibilidad de conservar esos datos localmente. El caso de uso más común consiste en almacenar en caché datos relevantes para que el dispositivo no pueda acceder a la red, de modo que el usuario pueda explorar ese contenido mientras está sin conexión.

La biblioteca de persistencias Room brinda una capa de abstracción para SQLite que permite acceder a la base de datos sin problemas y, al mismo tiempo, aprovechar toda la tecnología de SQLite. En particular, Room brinda los siguientes beneficios:

- Verificación del tiempo de compilación de las consultas en SQL
- Anotaciones de conveniencia que minimizan el código estándar repetitivo y propenso a errores
- Rutas de migración de bases de datos optimizadas

Debido a estas consideraciones, te recomendamos que uses Room en lugar de usar las API de SQLite directamente.

8.1.8.1 CONFIGURACIÓN

Para usar Room en tu app, agrega las siguientes dependencias al archivo build.gradle de la app:

```
dependencies {  
    def room_version = "2.5.0"  
  
    implementation "androidx.room:room-runtime:$room_version"  
    annotationProcessor "androidx.room:room-compiler:$room_version"  
  
    // To use Kotlin annotation processing tool (kapt)  
    kapt "androidx.room:room-compiler:$room_version"  
    // To use Kotlin Symbol Processing (KSP)  
    ksp "androidx.room:room-compiler:$room_version"  
  
    // optional - RxJava2 support for Room  
    implementation "androidx.room:room-rxjava2:$room_version"  
  
    // optional - RxJava3 support for Room  
    implementation "androidx.room:room-rxjava3:$room_version"  
  
    // optional - Guava support for Room, including Optional and ListenableFuture  
    implementation "androidx.room:room-guava:$room_version"  
  
    // optional - Test helpers  
    testImplementation "androidx.room:room-testing:$room_version"  
  
    // optional - Paging 3 Integration  
    implementation "androidx.room:room-paging:$room_version"  
}
```

8.1.8.2 COMPONENTES PRINCIPALES

Estos son los tres componentes principales de Room:

- La clase de la base de datos que contiene la base de datos y sirve como punto de acceso principal para la conexión subyacente a los datos persistentes de la app.
- Las entidades de datos que representan tablas de la base de datos de tu app.
- Objetos de acceso a datos (DAOs) que proporcionan métodos que tu app puede usar para consultar, actualizar, insertar y borrar datos en la base de datos.

La clase de base de datos proporciona a tu app instancias de los DAOs asociados con esa base de datos. A su vez, la app puede usar los DAOs para recuperar datos de la base de datos como instancias de objetos de entidad de datos asociados. La app también puede usar las entidades de datos definidas a fin de actualizar filas de las tablas correspondientes o crear filas nuevas para su inserción. En la figura 1, se muestran las relaciones entre los diferentes componentes de Room.

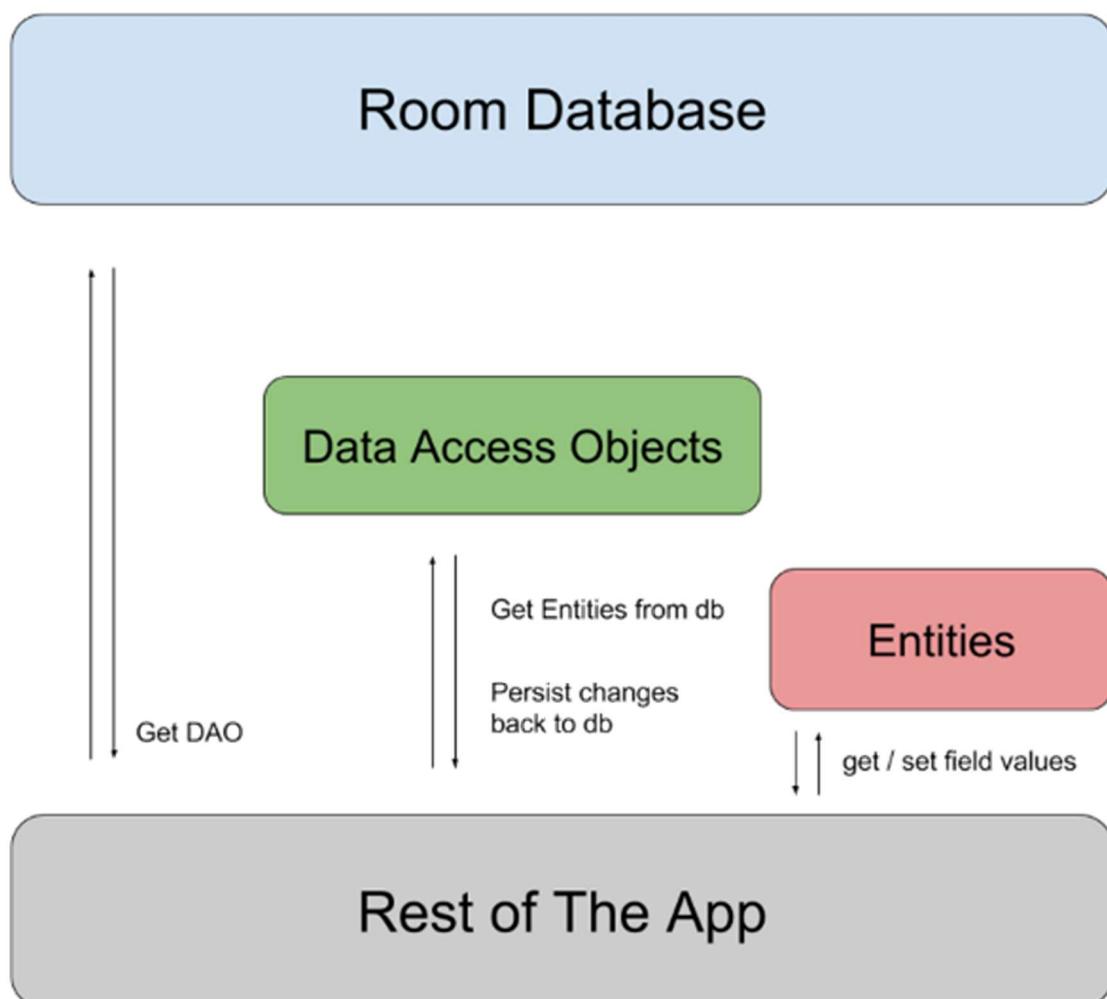


Imagen 28 Diagrama de la arquitectura de la biblioteca de Room

8.1.8.3 EJEMPLO DE IMPLEMENTACIÓN

En esta sección, se presenta un ejemplo de implementación de una base de datos de Room con una sola entidad de datos y un DAO único.

El siguiente código define una entidad de datos User. Cada instancia de User representa una fila en una tabla de user en la base de datos de la app.

```
@Entity  
data class User(  
    @PrimaryKey val uid: Int,  
    @ColumnInfo(name = "first_name") val firstName: String?,  
    @ColumnInfo(name = "last_name") val lastName: String?  
)
```

Para obtener más información sobre las entidades de datos de Room, consulta [Cómo definir datos con entidades de Room](#).

El siguiente código define un DAO llamado UserDao. UserDao proporciona los métodos que el resto de la app usa para interactuar con los datos de la tabla user.

```
@Dao  
interface UserDao {  
    @Query("SELECT * FROM user")  
    fun getAll(): List<User>  
  
    @Query("SELECT * FROM user WHERE uid IN (:userIds)")  
    fun loadAllByIds(userIds: IntArray): List<User>  
  
    @Query("SELECT * FROM user WHERE first_name LIKE :first AND " +  
        "last_name LIKE :last LIMIT 1")  
    fun findByName(first: String, last: String): User  
  
    @Insert  
    fun insertAll(vararg users: User)  
  
    @Delete  
    fun delete(user: User)  
}
```

Para obtener más información sobre los DAOs, consulta Cómo acceder a los datos con DAO de Room.

Con el siguiente código, se define una clase AppDatabase para contener la base de datos. AppDatabase define la configuración de la base de datos y sirve como el punto de acceso principal de la app a los datos persistentes. La clase de la base de datos debe cumplir con las siguientes condiciones:

- La clase debe tener una anotación `@Database` que incluya un array `entities` que enumere todas las entidades de datos asociados con la base de datos.
- Debe ser una clase abstracta que extienda `RoomDatabase`.
- Para cada clase DAO que se asoció con la base de datos, esta base de datos debe definir un método abstracto que tenga cero argumentos y muestre una instancia de la clase DAO.

```
@Database(entities = [User::class], version = 1)
abstract class AppDatabase : RoomDatabase() {
    abstract fun userDao(): UserDao
}
```

Uso

Después de definir la entidad de datos, el DAO y el objeto de base de datos, puedes usar el siguiente código para crear una instancia de la base de datos:

```
val db = Room.databaseBuilder(
    applicationContext,
    AppDatabase::class.java, "database-name"
).build()
```

Luego, puedes usar los métodos abstractos de AppDatabase para obtener una instancia del DAO. A su vez, puedes usar los métodos de la instancia del DAO para interactuar con la base de datos:

```
val userDao = db.userDao()
val users: List<User> = userDao.getAll()
```

8.1.9 JUNIT

En el desarrollo de software, la calidad es un factor clave para asegurar el éxito de un proyecto. Una de las formas más efectivas de garantizar la calidad del código es mediante la implementación de pruebas unitarias. En nuestro proyecto, Pokelytics, hemos utilizado JUnit, un popular framework de pruebas unitarias para Java, con el objetivo de asegurar la funcionalidad y estabilidad de nuestra aplicación. A continuación, exploraremos qué es JUnit, por qué lo hemos utilizado en Pokelytics y cómo nos ha ayudado a mejorar la calidad del código.

JUnit es un framework de pruebas unitarias para el lenguaje de programación Java. Proporciona un conjunto de anotaciones, aserciones y herramientas para crear pruebas automatizadas y ejecutarlas de manera eficiente. JUnit permite a los desarrolladores escribir casos de prueba que verifican el comportamiento esperado de componentes individuales o unidades de código, como métodos, clases o módulos.

En el contexto de nuestro proyecto, Pokelytics, JUnit desempeñó un papel fundamental en la garantía de calidad de nuestro código. A continuación, se presentan las razones por las cuales decidimos utilizar JUnit como framework de pruebas unitarias:

1. Identificar y prevenir errores: Las pruebas unitarias nos permitieron identificar y prevenir errores en las etapas tempranas del desarrollo. Al escribir pruebas para cada unidad de código, pudimos validar su comportamiento esperado y detectar posibles problemas antes de que se propagaran a otras partes del sistema. Esto nos ayudó a reducir los costos y el tiempo invertido en la depuración posterior.
2. Facilitar la refactorización: La refactorización es un proceso esencial en el desarrollo de software para mejorar la estructura y el diseño del código. Sin embargo, puede introducir errores si no se realiza correctamente. JUnit nos proporcionó un conjunto de pruebas confiables que nos permitieron refactorizar el código con confianza. Después de cada refactorización, ejecutamos las pruebas unitarias para asegurarnos de que el comportamiento previsto se mantuviera intacto.
3. Asegurar la integridad del código: Pokelytics es una aplicación sencilla pero con algunos componentes interconectados. Las pruebas unitarias nos permitieron verificar la interacción correcta entre estas unidades de código y asegurarnos de que funcionaran correctamente en conjunto. Esto fue especialmente valioso en escenarios donde cambios en un componente podían tener efectos no deseados en otros.
4. Fomentar la colaboración: JUnit promovió la colaboración efectiva entre los miembros del equipo de desarrollo. Al escribir pruebas unitarias, pudimos comunicar claramente las expectativas del comportamiento de una unidad de código a otros desarrolladores. Además, las pruebas unitarias proporcionaron una base para la integración continua y facilitaron la detección rápida de conflictos de integración.
5. Mejorar la documentación: Las pruebas unitarias actúan como una forma de documentación viva del código. Al revisar las pruebas, los desarrolladores pueden comprender rápidamente cómo se supone que deben usarse y cómo deben comportarse las distintas unidades de código (esto es especialmente valioso cuando se trabaja en un proyecto a largo plazo o cuando hay rotación de personal en el equipo, aunque no es nuestro caso).

6. Incrementar la confianza: Mediante la implementación de pruebas unitarias con JUnit, pudimos ganar confianza en la calidad y estabilidad de nuestro código. Las pruebas automatizadas nos dieron la tranquilidad de que, a medida que se realizaban cambios en el código, se mantendría su funcionamiento correcto. Esto nos permitió avanzar más rápidamente y centrarnos en agregar nuevas funcionalidades sin preocuparnos por romper el código existente.

En Pokelytics, reconocimos la importancia de la calidad del código y la necesidad de garantizar su estabilidad y funcionalidad. Para lograrlo, utilizamos JUnit como framework de pruebas unitarias. Las pruebas automatizadas con JUnit nos permitieron identificar y prevenir errores, facilitar la refactorización, asegurar la integridad del código, fomentar la colaboración, mejorar la documentación y aumentar nuestra confianza en el producto final.

El uso de JUnit en nuestro proyecto fue fundamental para mantener una base de código sólida y confiable. Las pruebas unitarias nos brindaron la seguridad necesaria para realizar cambios con confianza y evolucionar continuamente nuestra aplicación. Recomendamos encarecidamente la adopción de JUnit y las pruebas unitarias en cualquier proyecto de desarrollo de software, ya que pueden marcar una gran diferencia en términos de calidad y estabilidad del código.

En la red podemos encontrar múltiple información a cerca de cómo realizar estas pruebas. A continuación vamos a mostrar un modelo que nos ha resultado muy útil:

8.1.9.1 DISEÑO DE JUNIT

JUnit está diseñado alrededor de dos patrones de diseño principales: el patrón Command y el patrón Composite.

Un TestCase es un objeto Command. Cualquier clase que contenga métodos de testeo debería extender la clase TestCase. Un TestCase puede definir cualquier número de métodos públicos testXXX(). Cuando quieres comprobar el resultado esperado y el real, invocas una variante del método assert().

Las subclases de TestCase que contienen varios métodos testXXX() pueden utilizar los métodos setUp() y tearDown() para inicializar y liberar cualquier objeto común que se vaya a testear, conocido como la "instalación" (material) del test. Cada método de tests se ejecuta sobre su propia instalación, llamando a setUp() antes y a tearDown() después de cada método para asegurarse de que no hay efectos colaterales entre ejecuciones de tests.

Los ejemplares de TestCase pueden unirse en árboles de TestSuite que invocan automáticamente todos los métodos testXXX() definidos en cada ejemplar de TestCase. Un TestSuite es una composición de otros tests, bien ejemplares de TestCase u otros ejemplares de TestSuite. El comportamiento compuesto exhibido por TestSuite te permite ensamblar suites de tests de suites de tests, de una profundidad arbitraria, y ejecutar todos los tests automáticamente para obtener un simple estado de pasado o fallido.

Paso 1: Escribir un Test

Primero, escribiremos un test para probar un único componente de software. Nos enfocaremos en escribir test que comprueben el comportamiento que tiene el mayor potencial de rotura, así maximizaremos los beneficios de nuestra inversión en testeo.

Para escribir un test, sigue estos pasos:

1. Define una subclase de TestCase.
2. Sobreescribe el método setUp() para inicializar el objeto(s) a probar.
3. Sobreescribe el método tearDown() para liberar el objeto(s) a probar.
4. Define uno o más métodos testXXX() públicos que prueben el objeto(s) y aserten los resultados esperados.
5. Define un método factoría suite() estático que cree un TestSuite que contenga todos los métodos testXXX() del TestCase.
6. Opcionalmente, define un método main() que ejecute el TestCase en modo por lotes.

Abajo puedes ver un test de ejemplo:

```
import junit.framework.Test;
import junit.framework.TestCase;
import junit.framework.TestSuite;

public class ShoppingCartTest extends TestCase {

    private ShoppingCart _bookCart;
    private Product _defaultBook;

    /**
     * Constructs a ShoppingCartTest with the specified name.
     * @param name Test case name.
     */
    public ShoppingCartTest(String name) {
        super(name);
    }

    /**
     * Sets up the test fixture.
     * Called before every test case method.
     */
    protected void setUp() {
```

```

    _bookCart = new ShoppingCart();

    _defaultBook = new Product("Extreme Programming", 23.95);
    _bookCart.addItem(_defaultBook);
}

/**
 * Tears down the test fixture.
 * Called after every test case method.
 */
protected void tearDown() {
    _bookCart = null;
}

/**
 * Tests adding a product to the cart.
 */
public void testProductAdd() {

    Product newBook = new Product("Refactoring", 53.95);
    _bookCart.addItem(newBook);

    double expectedBalance = _defaultBook.getPrice() + newBook.getPrice();

    assertEquals(expectedBalance, _bookCart.getBalance(), 0.0);

    assertEquals(2, _bookCart.getItemCount());
}

/**
 * Tests the emptying of the cart.
 */
public void testEmpty() {

    _bookCart.empty();

    assertTrue(_bookCart.isEmpty());
}

```

```

/**
 * Tests removing a product from the cart.
 * @throws ProductNotFoundException If the product was not in the cart.
 */
public void testProductRemove() throws ProductNotFoundException {

    _bookCart.removeItem(_defaultBook);

    assertEquals(0, _bookCart.getItemCount());

    assertEquals(0.0, _bookCart.getBalance(), 0.0);
}

/**
 * Tests removing an unknown product from the cart.
 * This test is successful if the
 * ProductNotFoundException is raised.
 */
public void testProductNotFound() {

    try {

        Product book = new Product("Ender's Game", 4.95);
        _bookCart.removeItem(book);

        fail("Should raise a ProductNotFoundException");

    } catch(ProductNotFoundException success) {
        // successful test
    }
}

/**
 * Assembles and returns a test suite for
 * all the test methods of this test case.
 * @return A non-null test suite.
 */
public static Test suite() {

```

```

//  

// Reflection is used here to add all  

// the testXXX() methods to the suite.  

//  

TestSuite suite = new TestSuite(ShoppingCartTest.class);  

//  

// Alternatively, but prone to error when adding more  

// test case methods...  

//  

// TestSuite suite = new TestSuite();  

// suite.addTest(new ShoppingCartTest("testEmpty"));  

// suite.addTest(new ShoppingCartTest("testProductAdd"));  

// suite.addTest(new ShoppingCartTest("testProductRemove"));  

// suite.addTest(new ShoppingCartTest("testProductNotFound"));  

//  

//  

return suite;  

}  

/**  

 * Runs the test case.  

 */  

public static void main(String args[]) {  

    junit.textui.TestRunner.run(suite());  

}
}

```

Paso 2: Escribir la Clase a Testear

Ahora debes escribir las clases necesarias para pasar el test construido en el paso anterior:

Primero necesitas la clase principal: ShoppingCart.java:

```

import java.util.*;  

/**  

 * An example shopping cart.  

 * This class should not be mistaken for a production-quality shopping cart. It's  

 * merely provided as an example class under test as described in the JUnitPrimer.

```

```

/*
 * @author <a href="mailto:mike@clarkware.com">Mike Clark</a>
 * @author <a href="http://www.clarkware.com">Clarkware Consulting, Inc.</a>
 */

public class ShoppingCart {

    private ArrayList _items;

    /**
     * Constructs a ShoppingCart instance.
     */
    public ShoppingCart() {
        _items = new ArrayList();
    }

    /**
     * Returns the balance.
     * @return Balance.
     */
    public double getBalance() {
        Iterator i = _items.iterator();
        double balance = 0.00;
        while (i.hasNext()) {
            Product p = (Product)i.next();
            balance = balance + p.getPrice();
        }

        return balance;
    }

    /**
     * Adds the specified product.
     * @param p Product.
     */
    public void addItem(Product p) {
        _items.add(p);
    }
}

```

```

/**
 * Removes the specified product.
 * @param p Product.
 * @throws ProductNotFoundException If the product does not exist.
 */
public void removeItem(Product p) throws ProductNotFoundException {
    if (!_items.remove(p)) {
        throw new ProductNotFoundException();
    }
}

/**
 * Returns the number of items in the cart.
 * @return Item count.
 */
public int getItemCount() {
    return _items.size();
}

/**
 * Empties the cart.
 */
public void empty() {
    _items = new ArrayList();
}

/**
 * Indicates whether the cart is empty.
 * @return true if the cart is empty;
 *         false otherwise.
 */
public boolean isEmpty() {
    return (_items.size() == 0);
}

```

También necesitarás la clase Product.java:

```
/**
```

```

 * An example product for use in the example shopping cart.
 * @author <a href="mailto:mike@clarkware.com">Mike Clark</a>
 * @author <a href="http://www.clarkware.com">Clarkware Consulting, Inc.</a>
 */

public class Product {

    private String _title;
    private double _price;

    /**
     * Constructs a <codigoenlinea>Product</codigoenlinea>.
     * @param title Product title.
     * @param price Product price.
     */
    public Product(String title, double price) {
        _title = title;
        _price = price;
    }

    /**
     * Returns the product title.
     * @return Title.
     */
    public String getTitle() {
        return _title;
    }

    /**
     * Returns the product price.
     * @return Price.
     */
    public double getPrice() {
        return _price;
    }

    /**
     * Tests product equality.
     * @return true if the products are equal.
     */
}

```

```

    */
    public boolean equals(Object o) {

        if (o instanceof Product) {
            Product p = (Product)o;
            return p.getTitle().equals(_title);
        }

        return false;
    }
}

```

Y la clase que generará la excepción ProductNotFoundException:

```

/**
 * Exception thrown when a product is not found in a shopping cart.
 * @author <a href="mailto:mike@clarkware.com">Mike Clark</a>
 * @author <a href="http://www.clarkware.com">Clarkware Consulting, Inc.</a>
 */

public class ProductNotFoundException extends Exception {

    /**
     * Constructs a <codigoenlinea>ProductNotFoundException</codigoenlinea>.
     */
    public ProductNotFoundException() {
        super();
    }
}

```

Paso 3: Escribir una Suite de Tests

Luego, escribiremos una suite de tests que incluya varios tests. La suite nos permitirá ejecutar todos los tests como si fueran uno sólo.

1. Define una subclase de TestCase.
2. Define un método factoría suite() estático que cree un TestSuite que contenga todos los métodos testXXX() del TestCase.
3. Opcionalmente, define un método main() que ejecute el TestCase en modo por lotes.

Abajo puedes ver un ejemplo de la suite de tests:

```
import junit.framework.Test;
import junit.framework.TestSuite;

public class EcommerceTestSuite {

    /**
     * Assembles and returns a test suite
     * containing all known tests.
     *
     * New tests should be added here!
     *
     * @return A non-null test suite.
     */
    public static Test suite() {

        TestSuite suite = new TestSuite();

        //
        // The ShoppingCartTest we created above.
        //
        suite.addTest(ShoppingCartTest.suite());

        //
        // Another example test suite of tests.
        //
        suite.addTest(CreditCardTestSuite.suite());

        return suite;
    }

    /**
     * Runs the test suite.
     */
    public static void main(String args[]) {
        junit.textui.TestRunner.run(suite());
    }
}
```

Paso 4: Ejecutar los Tests

Ahora que ya hemos escrito una suite de tests que contiene una colección de tests y otras suites, podemos ejecutar toda la suite o cualquiera de sus tests individualmente. Ejecutando un TestSuite se ejecutarán automáticamente todos sus ejemplares de TestCase y de TestSuite subordinados. Ejecutando un TestCase invocará automáticamente a sus métodos testXXX() públicos.

JUnit proporciona dos interfaces de usuario uno de texto y otro gráfico. Ambos interfaces de usuario indican cuántos tests se ejecutaron, cuantos errores o fallos, y un único estado de terminación. Deberías poder ejecutar tus tests y ver de un vistazo su estado, al igual que lo haces con tu compilador.

La interface de modo texto (junit.textui.TestRunner) muestra "OK" si se pasaron todos los tests y un mensaje de fallo si cualquier de los tests falla.

La interface de modo gráfico (junit.swingui.TestRunner) muestra una ventana Swing con una barra de progreso en verde si se pasaron todos los tests o una barra de progreso roja si falló alguno de los tests.

En general, las clases TestSuite y TestCase deberían definir un método main() que emplee la interface de usuario adecuado. Los tests que hemos definido hasta ahora han definido un método main() que emplea la interface de usuario en modo texto.

Para ejecutar nuestros tests según lo hemos definido en el método main(), utilizamos:

```
java ShoppingCartTest
```

Alternativamente, el test se puede ejecutar con el interface de usuario de modo texto utilizando:

```
java junit.textui.TestRunner ShoppingCartTest
```

o con el GUI de Swing utilizando:

```
java junit.swingui.TestRunner ShoppingCartTest
```

La suite EcommerceTestSuite se puede ejecutar de forma similar.

Paso 5: Organizar los Tests

El último paso es decidir donde residirán nuestros tests dentro de nuestro entorno de desarrollo.

Aquí tienes la forma recomendada para organizar tus tests:

1. Crea los tests en el mismo paquete que el código a testear. Por ejemplo, el paquete com.mydotcom.ecommerce contendría todas las clases a nivel de la aplicación y todos los tests para esos componentes.
2. Para evitar la mezcla de código de la aplicación y de testeo en tus directorios de código fuente, crea una estructura de directorio paralela alineada con la estructura de paquete que contenga el código de los tests.

3. Por cada paquete Java de tu aplicación, define una clase TestSuite que contenga todos los tests para validar el código de ese paquete.
4. Define clases TestSuite similares que creen suites de tests de alto y bajo nivel en los otros paquetes (y sub-paquetes) de la aplicación.
5. Asegúrate de que tu proceso de compilación incluye todos los tests. Esto ayuda a asegurar de tus tests siempre están actualizados con el último código y los mantiene frescos.

Mediante la creación de un TestSuite en cada paquete Java, a los distintos niveles de empaquetamiento, podrás ejecutar un TestSuite con cualquier nivel de abstracción. Por ejemplo, puedes definir un com.mydotcom.AllTests que ejecute todos los tests del sistema y un com.mydotcom.ecommerce.EcommerceTestSuite que solo ejecute aquellos que validan los componentes de comercio electrónico.

El árbol de herencia de tests se puede extender hasta una profundidad arbitraria. Dependiendo del nivel de abstracción que estemos desarrollando en el sistema, podrás ejecutar un tests apropiado. Sólo elige una capa del sistema y testeala!

Aquí tienes un ejemplo de árbol de tests:

AllTests (Top-level Test Suite)

 SmokeTestSuite (Structural Integrity Tests)

 EcommerceTestSuite

 ShoppingCartTestCase

 CreditCardTestSuite

 AuthorizationTestCase

 CaptureTestCase

 VoidTestCase

 UtilityTestSuite

 MoneyTestCase

 DatabaseTestSuite

 ConnectionTestCase

 TransactionTestCase

 LoadTestSuite (Performance and Scalability Tests)

 DatabaseTestSuite

 ConnectionPoolTestCase

 ThreadPoolTestCase

Lenguaje de Testeo

Debes tener en mente estas cosas cuando testeas:

- El software hace bien aquellas cosas que los tests chequean.
- Testea un poco, codifica un poco, testea un poco, codifica un poco...
- Asegurate de que los test se ejecutan al 100%.
- Ejecuta todos los tests del sistema al menos una vez al día (o por la noche).
- Escribe tests para las áreas de código que tengan la más alta probabilidad de fallar.
- Escribe los test que te vayan a proporcionar más beneficios en tu inversión en testeo.
- Si te ves depurando utilizando System.out.println(), escribe un test que chequee el resultado automáticamente.
- Cuando se reporta un bug, escribe un test que lo exponga.
- La próxima vez que alguien te pida que le ayudes a depurar, ayudale a escribir un test.
- Escribe los tests antes de escribir el código y sólo escribe código nuevo cuando falle un test.

8.1.10 EMPLEANDO MOCKITO

Mockito es una biblioteca de pruebas de Java que nos permite crear objetos simulados, también conocidos como mocks, para simular el comportamiento de dependencias externas en nuestras pruebas unitarias. Los mocks son objetos que se comportan como las clases reales pero pueden ser configurados para devolver valores predefinidos o lanzar excepciones cuando se les invoca. Mockito proporciona una sintaxis sencilla y expresiva para definir el comportamiento esperado de los mocks y verificar su interacción con el código bajo prueba.

En el contexto de nuestro proyecto, Pokelytics, Mockito se convirtió en una herramienta invaluable para nuestras pruebas unitarias. A continuación, se presentan las razones por las cuales decidimos utilizar Mockito:

Aislamiento de dependencias externas: Pokelytics depende de servicios externos, como la API de Pokémon y la base de datos de usuarios. Al simular estas dependencias externas con mocks, pudimos aislar nuestro código bajo prueba y asegurarnos de que las pruebas no dependieran de la disponibilidad o el estado de estos servicios externos. Esto nos permitió ejecutar las pruebas de manera más eficiente y reducir la posibilidad de errores causados por factores externos.

Control preciso del comportamiento: Mediante el uso de Mockito, pudimos configurar el comportamiento deseado de los mocks para que se adaptaran a nuestros escenarios de prueba. Pudimos definir qué valores devolverían los métodos de los mocks, cómo reaccionarían a ciertos parámetros y cómo se comportarían en distintos contextos. Esto nos permitió simular diferentes situaciones y probar casos extremos que podrían ser difíciles de reproducir en un entorno real.

Simplificación de la configuración del entorno de prueba: En muchos casos, configurar un entorno de prueba puede ser una tarea complicada y llevar mucho tiempo. Con Mockito, pudimos reducir significativamente la cantidad de código necesario para preparar el entorno de prueba, ya que solo nos enfocamos en definir el comportamiento de los mocks necesarios. Esto nos permitió escribir pruebas más rápidamente y mantener un código de prueba más limpio y legible.

Verificación de interacciones: Mockito nos permitió verificar las interacciones entre el código bajo prueba y las dependencias simuladas. Pudimos asegurarnos de que se invocaran los métodos correctos con los parámetros adecuados y en el orden esperado. Esto nos ayudó a validar la comunicación correcta entre componentes y detectar posibles problemas de integración o errores de lógica.

Mejora de la cobertura de pruebas: Al simular dependencias externas, Mockito nos permitió probar escenarios difíciles de reproducir en un entorno real. Pudimos crear casos de prueba específicos para situaciones excepcionales, errores o condiciones límite. Esto mejoró nuestra cobertura de pruebas y nos brindó mayor confianza en la robustez y estabilidad de nuestra aplicación.

Cómo aplicamos Mockito en Pokelytics: En Pokelytics, utilizamos Mockito en varios aspectos de nuestras pruebas unitarias. A continuación, se describen algunas de las aplicaciones específicas de Mockito en nuestro proyecto:

Simulación de la API de Pokémon: Para probar las funcionalidades relacionadas con la obtención y el procesamiento de datos de la API de Pokémon, utilizamos Mockito

para simular las respuestas de la API. Configuramos los mocks para devolver datos predefinidos que representaban diferentes situaciones, como respuestas exitosas, errores o datos incompletos. Esto nos permitió probar escenarios diversos y garantizar un comportamiento correcto de nuestra aplicación frente a las diferentes respuestas de la API.

Mocking de la base de datos de usuarios: En Pokelytics, nuestra aplicación requería acceso a una base de datos de usuarios para almacenar información personalizada. Utilizamos Mockito para simular el comportamiento de la base de datos y garantizar la correcta interacción con ella en nuestras pruebas unitarias. Configuramos los mocks para devolver datos predefinidos o lanzar excepciones según los casos de prueba, lo que nos permitió verificar la lógica relacionada con la persistencia y manipulación de datos de usuario.

Simulación de servicios de notificación: En algunas funcionalidades de Pokelytics, era necesario enviar notificaciones a los usuarios. Utilizamos Mockito para simular los servicios de notificación y garantizar que las notificaciones se enviaran correctamente en los escenarios relevantes. Configuramos los mocks para verificar que se invocaran los métodos de envío de notificaciones con los parámetros adecuados, lo que nos permitió probar y verificar esta funcionalidad de manera aislada.

La utilización de Mockito en nuestras pruebas unitarias nos permitió simular dependencias externas y crear pruebas más eficientes y aisladas. Mockito nos brindó un control preciso del comportamiento de los mocks, simplificó la configuración del entorno de prueba, facilitó la verificación de interacciones y mejoró nuestra cobertura de pruebas. Como resultado, pudimos garantizar la calidad y la estabilidad de nuestra aplicación, además de detectar y solucionar posibles problemas antes de su lanzamiento. Recomendamos el uso de Mockito, o frameworks similares, en proyectos de desarrollo de aplicaciones, ya que pueden ser una herramienta poderosa para garantizar la calidad del código y facilitar el proceso de pruebas unitarias.

8.2 PROBLEMAS ENCONTRADOS Y JUSTIFICACIÓN

Durante el proceso de testeo, los problemas encontrados han sido mayoritariamente con los emuladores de Android. Por algún motivo a la hora de ejecutar algunos de los tests, el emulador ha producido errores donde indicaba que ya se estaba ejecutando, por lo que no podían realizarse los tests.

Para poder solucionar esto, finalmente hemos conectado un terminal móvil al PC y una vez lo detecta el programa podemos lanzar la emulación del test Android en el terminal físico.

También otro de los problemas encontrados ha sido que nunca habíamos realizado estas pruebas por lo que era algo nuevo para algunos de nosotros. Por suerte, uno de los integrantes del equipo de Pokelytics conoce bastante bien la herramienta y ha realizado estos tests en otras ocasiones, por lo que ha sido quien ha ido dirigiendo las operaciones de testeo.

PokeLytics (C:\Users\Carlos\Desktop\PokeLytics\PokeLytics) - bottom_navigation_menu.xml [PokeLytics.app.main]

```
<?xml version="1.0" encoding="utf-8"?>
<menu xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto">
```

Test Results

```
1ms Executing tasks: [:app:testDebugUnitTest] in project C:\Users\Carlos\Desktop\PokeLytics\PokeLytics
```

Starting Gradle Daemon...
Gradle Daemon started in 6 s 294 ms

> Task :app:preBuild UP-TO-DATE
> Task :app:preDebugBuild UP-TO-DATE
> Task :app:compileDebugAidl NO-SOURCE
> Task :app:compileDebugRenderscript NO-SOURCE
> Task :app:dataBindingMergeDependencyArtifactsDebug UP-TO-DATE
AGPBI: {"kind": "warning", "text": "Your project has set 'android.useAndroidX=true', but configuration ':app:debugRuntimeClasspath' still contains
> Task :app:generateDebugResValues UP-TO-DATE
> Task :app:generateDebugResources UP-TO-DATE
> Task :app:processDebugGoogleServices UP-TO-DATE
> Task :app:packageDebugResources
> Task :app:generateDebugTriggerDebug UP-TO-DATE
> Task :app:generateDebugBuildConfig UP-TO-DATE
> Task :app:mergeDebugResources
> Task :app:checkDebugManifests UP-TO-DATE
> Task :app:mapDebugSourceSetPaths UP-TO-DATE
> Task :app:createDebugCompatibleScreenManifests UP-TO-DATE

Test Results

```
Passed 8 tests, 301 ms
```

Logs

```
Logcat Device Info
```

Test Results

- study.project.pokelytics.AndroidComponentTest
 - testMainViewBinding
 - testButtonClick
- study.project.pokelytics.ViewModelTest
 - testLocationListViewModel
 - testBerryListViewModel
 - testMoveListViewModel
 - testPokemonListViewModel
 - testRegionListViewModel

All Tests: 8 total, 8 passed

Tests in 'PokeLytics.app.unitTest': 10 total, 10 passed

study.project.pokelytics.BaseFunctions

- fillBerry
- toList
- fillPokemon
- sortThis
- getMatch

study.project.pokelytics.DataSourceTest

- testPokemonDataSource
- testLocationDataSource
- testBerryDataSource
- testMoveDataSource
- testRegionDataSource

Imagen 29 Tests

9. POKELYTICS COMO PROYECTO EMPRENDEDOR

En la actualidad, el sector productivo se enfrenta a un mercado cada vez más exigente en cuanto a la oferta de productos y servicios, lo que hace que las empresas deban buscar formas innovadoras de diferenciarse y destacar entre su competencia. Una de las formas en que las empresas pueden lograr esto es a través de la implementación de herramientas tecnológicas que les permitan mejorar sus procesos y ofrecer un valor agregado a sus clientes.

En este sentido, una aplicación de Pokédex podría resultar muy útil para el sector productivo, especialmente para aquellos negocios relacionados con la industria de los videojuegos y el entretenimiento. Una Pokédex es una herramienta digital que permite a los usuarios conocer información detallada acerca de los diferentes Pokémon, sus características, habilidades y debilidades, entre otros datos relevantes.

Por ejemplo, una empresa que se dedique a la venta de videojuegos podría utilizar una aplicación de Pokédex para ofrecer a sus clientes una experiencia más completa al momento de adquirir un juego de la saga Pokémon. La aplicación podría incluir información detallada acerca de los diferentes Pokémon disponibles en el juego, lo que permitiría a los usuarios conocer de antemano las habilidades y debilidades de cada uno, y así tomar decisiones informadas sobre qué Pokémon incluir en su equipo de combate.

De igual forma, una aplicación de Pokédex podría ser utilizada por empresas que se dediquen a la producción de contenido relacionado con el mundo de Pokémon, como por ejemplo animaciones, series de televisión, o productos de merchandising. La aplicación podría ser utilizada como una herramienta de investigación para el desarrollo de personajes, historias y productos, permitiendo a los creadores obtener información detallada acerca de los diferentes Pokémon y así crear contenido más interesante y atractivo para su audiencia.

En definitiva, una aplicación de Pokédex podría resultar muy útil para el sector productivo, especialmente en aquellos negocios relacionados con la industria de los videojuegos y el entretenimiento. La implementación de esta herramienta permitiría a las empresas mejorar sus procesos y ofrecer un valor añadido a sus clientes, lo que se traduciría en una ventaja competitiva y una mayor satisfacción del consumidor.

A continuación, se identifican las necesidades detectadas en el sector productivo que originan la oportunidad de negocio que se detalla en los siguientes puntos.

9.1 ANÁLISIS DE LA SITUACIÓN ACTUAL

En la actualidad contamos con diversas aplicaciones en Play Store que dan un soporte similar al consumidor o cliente. Estas aplicaciones tienen múltiples descargas, algunas de ellas llegan a situarse en un volumen de más de 100 mil descargas lo que es indicativo de la alta demanda de este producto.

De manera oficial la franquicia Pokémon posee una aplicación en la AppStore de Apple, pero no en la Play Store. Además, ésta tiene un coste para todo aquel que desee descargarla. Sin embargo, prácticamente la totalidad de las aplicaciones de este tipo

desarrolladas en Android no tienen ningún cargo, lo que hace que sea más accesible a los usuarios dado que no tienen que hacer un desembolso para usarla.

Respecto a los usuarios que potencialmente utilizarán la app, nos hemos basado en estudios realizados por Nintendo y medios de comunicación referentes en el ámbito de los videojuegos.

Comenzamos analizando los datos de un medio de comunicación argentino llamado CulturaGeek donde un artículo nos proporciona algunos datos a cerca de la edad que tienen los jugadores de la saga Pokémon. Esto probablemente sea el dato más relevante a tener en cuenta ya que el usuario potencial de nuestra aplicación es toda aquella persona que juega a cualquiera de los videojuegos de la franquicia y que posee un smartphone. El artículo del diario dice en su titular: "La franquicia de videojuegos de Pokémon es una de las más atemporales. Con el correr de los años, su audiencia de todas las edades crece y se renueva. Muchos jóvenes se suman a estas nuevas historias, mientras que otros están en este mundo desde los '90."

Pokémon Scarlet & Violet está dando mucho de qué hablar. Hace poco les contábamos que estos videojuegos se convirtieron en el lanzamiento exclusivo con más ventas en la historia, registrando más de 10 millones de unidades en sus primeros 3 días. Por supuesto que esto demuestra que la franquicia está más viva que nunca y que tienen una audiencia fiel que no para de crecer y renovarse.

Como Pokémon es una franquicia de videojuegos atemporal, se sabe que reúne multitudes de todas las edades. Sin embargo, muchos de estos juegos suelen estar destinados a un público más infantil o se piensa que son en su totalidad para menores de edad, a pesar de que muchísima de su audiencia es mayor.

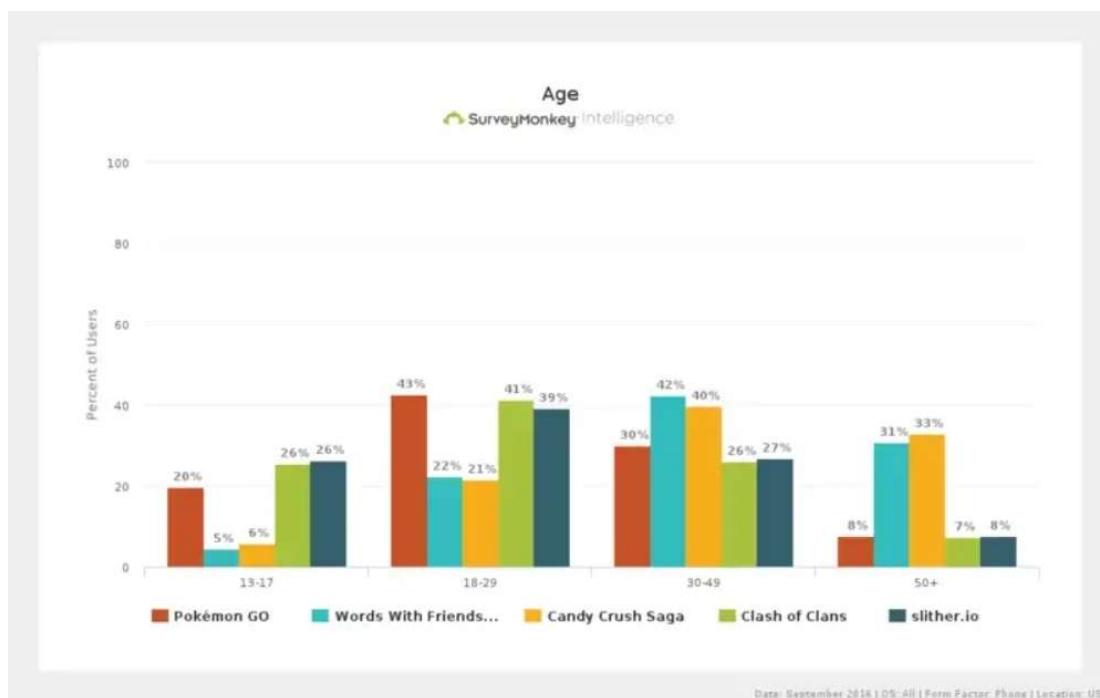


Imagen 30 Estadísticas jugadores de Pokémon

Otro estudio (pero esta vez de 2019) de SpringerLink que remarca que los jugadores de Pokémon Go que más jugaron estaban entre los 21 y 25 años, con un 32,6%. A este grupo lo seguían los usuarios de 26 a 30 años (con un 25,4%), los menores de 20 (con un 18%) y las personas entre 31 y 35 años (con 11,1%). Los datos para este estudio se recopilaron a través de una encuesta basada en la web dirigida a participantes de todo el mundo que eran jugadores de Pokémon Go o que habían jugado en ese momento.

Con todos estos datos podemos decir que el público que juega cualquiera de los títulos de Pokémon es un potencial usuario de nuestra aplicación, por lo que los números que son favorables a la franquicia también son a priori favorables a nuestro proyecto.

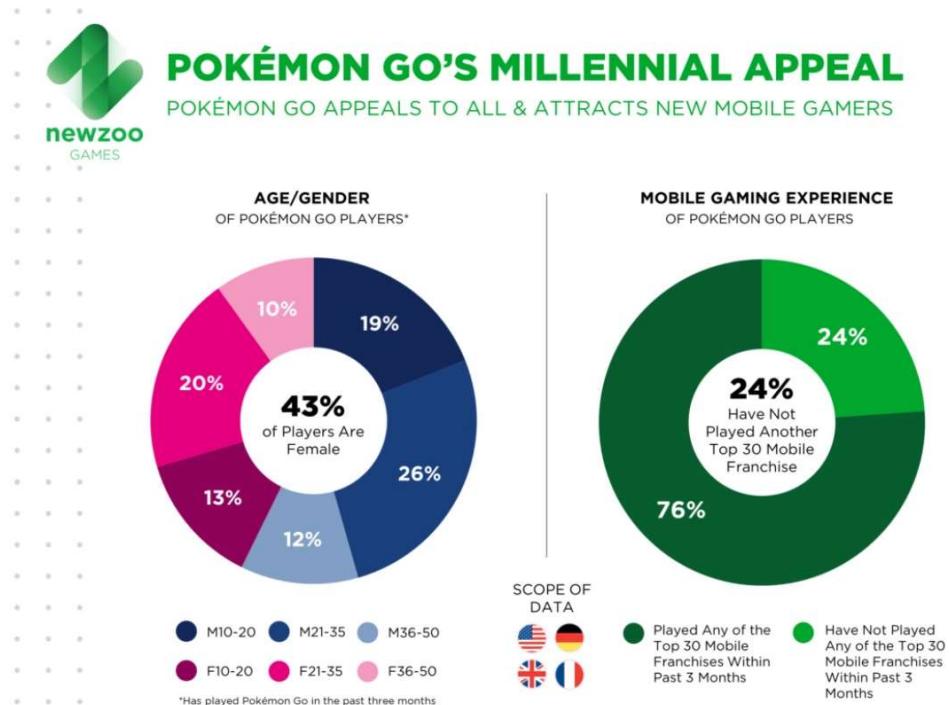


Imagen 31 Estadísticas Edades Jugadores

Los datos actuales vienen a indicarnos que pese a que Pokémon es una saga que comenzó hace muchos años, gran parte de los usuarios que en su lanzamiento eran niños y ahora se encuentran en edades comprendidas entre los 30 y los 40 años y continúan siguiendo los títulos de Pokémon. Sin embargo, pese a que hay muchos usuarios con altos conocimientos en la saga, la mayoría en algún momento requieren consultas tanto a webs, aplicaciones, foros u otras plataformas para poder optimizar su manera de jugar. Aquí es donde aparece Pokelytics para ofrecer una solución alternativa a estos usuarios o nuevos usuarios que desean adentrarse de una forma más inteligente en el modo de juego.

9.2 NECESIDADES DEL CLIENTE Y OPORTUNIDAD DE NEGOCIO

La saga Pokémon ha sido una de las más exitosas de la historia de los videojuegos, con una legión de seguidores que ha ido creciendo a lo largo de los años. Desde el lanzamiento del primer juego en 1996, la franquicia ha evolucionado constantemente, incorporando nuevas características y mecánicas de juego en cada nueva entrega.

Uno de los aspectos más importantes de la saga Pokémon ha sido la Pokédex, una herramienta que permite a los jugadores conocer más sobre cada uno de los Pokémon disponibles en el juego. A lo largo de los años, la Pokédex ha ido evolucionando, incorporando nuevas funcionalidades y características para satisfacer las necesidades de los jugadores.

Con el lanzamiento de Pokémon Go en 2016, la saga Pokémon experimentó un resurgimiento de popularidad sin precedentes. El juego móvil, que permitía a los jugadores capturar Pokémon en el mundo real, se convirtió en un fenómeno mundial, atrayendo a millones de jugadores en todo el mundo.

Uno de los aspectos más populares de Pokémon Go fue la Pokédex, que permitía a los jugadores ver información sobre los Pokémon que habían capturado o que estaban disponibles en su área. Sin embargo, la Pokédex de Pokémon Go tenía algunas limitaciones, lo que llevó a muchos jugadores a buscar alternativas.

Es aquí donde entra en juego Pokelytics, una aplicación que funciona como una Pokédex para los juegos de Pokémon. Pokelytics permite a los jugadores ver información detallada sobre cada uno de los Pokémon disponibles en los diferentes juegos de la saga, incluyendo estadísticas de combate, movimientos, habilidades y más.

Una de las principales necesidades de los clientes de Pokelytics es la capacidad de obtener información precisa y actualizada sobre los Pokémon. Con cada nueva entrega de la saga, se añaden nuevos Pokémon, movimientos y habilidades, lo que hace que la Pokédex necesite actualizaciones constantes. Los clientes de Pokelytics esperan que la aplicación se actualice con regularidad para reflejar los cambios en la saga Pokémon.

Otra necesidad importante de los clientes de Pokelytics es la facilidad de uso de la aplicación. A medida que la saga Pokémon ha evolucionado, los juegos se han vuelto cada vez más complejos, con más Pokémon, movimientos y mecánicas de juego. Los clientes de Pokelytics esperan que la aplicación les permita acceder a la información que necesitan de forma rápida y sencilla, sin tener que buscar a través de múltiples pantallas o menús.

Además, muchos jugadores de Pokémon son competitivos y buscan maximizar el potencial de sus Pokémon en los combates. Para ello, necesitan información detallada sobre las estadísticas de combate de cada Pokémon, así como las habilidades y movimientos que pueden utilizar. Los clientes de Pokelytics esperan que la aplicación les proporcione toda esta información de forma clara y fácil de entender.

Otra necesidad importante de los clientes de Pokelytics es la capacidad de colaborar y conectarse con otros jugadores de Pokémon. A medida que la saga Pokémon ha evolucionado, se ha vuelto cada vez más social, con características como las incursiones, los combates y los intercambios que fomentan la colaboración entre los

jugadores. Los clientes de Pokelytics esperan que la aplicación les permita conectarse con otros jugadores y compartir información y estrategias sobre cómo maximizar el potencial de sus Pokémon. Esta función no está implementada

Por último, los clientes de Pokelytics esperan que la aplicación sea compatible con todos los juegos de la saga Pokémon. A lo largo de los años, la saga ha tenido muchas entregas diferentes, desde los juegos clásicos de Game Boy hasta los juegos más recientes de Nintendo Switch. Los clientes de Pokelytics esperan que la aplicación sea capaz de manejar todas las diferentes versiones de la Pokédex y proporcionar información precisa y actualizada sobre cada uno de los Pokémon disponibles en cada juego.

Es difícil determinar con precisión cuántos jugadores tiene la saga Pokémon y, por tanto, cuantos podrían ser clientes potenciales de Pokelytics, ya que abarca muchos juegos y dispositivos diferentes, así como una gran variedad de regiones y mercados alrededor del mundo. Sin embargo, se estima que la franquicia ha vendido más de 350 millones de copias de sus juegos en todo el mundo desde su lanzamiento en 1996. Si Pokelytics fuera capaz de atraer un 0,1% el resultado de usuarios que habrían descargado y usado nuestra app sería de 350.000 personas. Una cifra realmente significativa si tenemos en cuenta que la inversión en el desarrollo, el marketing, etc. es de 0,00€.

Pese a que es difícil determinar una cifra exacta de jugadores de la saga Pokémon, su popularidad global y su éxito en la venta de juegos y otros productos sugieren que cuenta con una base de jugadores muy amplia y diversa en todo el mundo.

Además de los juegos, la franquicia Pokémon también incluye una serie animada de televisión, películas, juegos de cartas colecciónables, juguetes y otros productos. Todos estos elementos han contribuido a la popularidad global de Pokémon y han ayudado a crear una comunidad de fans leales y apasionados en todo el mundo.

En resumen, Pokelytics es una aplicación que cumple una necesidad importante para los jugadores de Pokémon: una Pokédex detallada y actualizada que les permita obtener información sobre los Pokémon disponibles en la saga. Los clientes de Pokelytics esperan que la aplicación sea precisa, fácil de usar, personalizable, conectada y compatible con todos los juegos de la saga. A medida que la saga Pokémon sigue evolucionando, es probable que los clientes de Pokelytics sigan teniendo nuevas necesidades y expectativas, y la aplicación deberá seguir adaptándose para satisfacerlas. Todo esto, no supondrá un gran esfuerzo a nivel de desarrollo, ya que al estar soportado sobre una API todo se simplifica bastante.

Para poder dar vida a este proyecto Pokelytics ha sido creado desde lo que hoy en día conocemos como “teletrabajo”. Pokelytics no cuenta ni contará con una sede u oficina central, al menos a corto plazo, ya que tratamos de que todas las personas que forman parte del proyecto sientan la libertad de dedicarse en cuerpo y alma en los mejores momentos que ellos estimen en su día. Al final se trata de ser productivo, creativo y tener propuestas, ideas y metas innovadoras. Por ello no estipulamos una jornada de inicio y final para dedicar al proyecto, pero sí una implicación en él notable.

Esta mentalidad creemos que favorecerá que cada miembro del proyecto pueda organizar su día a día con total libertad y así tener toda la facilidad posible para afrontar los retos del proyecto en los momentos más productivos que cada uno tenga.

Dado que es una aplicación y es un formato digital accesible desde cualquier terminal en cualquier parte del mundo, cualquier persona de cualquier rincón del planeta con un terminal capaz de descargar nuestra aplicación es un cliente potencial. Sin embargo, siendo más realistas los principales clientes potenciales son aquellas personas que juegan cualquiera de los títulos que posee la saga Pokémon.

Pokelytics es un proyecto ambicioso, pero sin evadirse de la realidad, por lo que los objetivos a corto plazo son objetivamente modestos: tener un buen posicionamiento en la Play Store y conseguir un número significativo de descargas por parte de la comunidad podrían ser principalmente los objetivos. Tras esto, con una visión más comercial, el objetivo sería implementar en la aplicación algunos módulos de pago y apartados de publicidad (que quedarían eliminados en las cuentas de pago), para todos aquellos usuarios que quieran tener una versión más completa y personalizada de la aplicación. Con ello conseguiríamos ingresos por publicidad para la aplicación para aquellos usuarios que se conforman con una experiencia más limitada e ingresos por pago directo de los usuarios que disfrutan de una versión premium más adaptada a ellos.

10. PREVENCIÓN DE RIESGOS

El trabajo informático desempeñado en oficinas no conlleva riesgos laborales tan graves como en otros sectores, como por ejemplo la construcción, pero eso no quiere decir que no existan peligros que deben ser mitigados.

Los principales riesgos laborales que experimentan los trabajadores informáticos son:

- Fatiga visual o muscular
- Golpes o caídas
- Contacto eléctrico
- Carga mental
- Distintos factores en la organización

Para poder evitar o minimizar estos riesgos hay que tener en cuenta los siguientes factores:

- Adecuada organización en el trabajo.
- Buen diseño de las instalaciones que garantice unas buenas condiciones ambientales.
- Selección adecuada del equipamiento: mesas, sillas, equipos informáticos...
- Formación e información a los trabajadores.

10.1 FATIGA VISUAL

Pueden aparecer molestias oculares por el uso de pantallas de visualización. Hay que tener una colocación ergonómica de la pantalla, es decir, colocarla a un mínimo de 40 centímetros y, además, inclinar ligeramente la parte inferior de modo que el enfoque sea perpendicular a nuestro ángulo de visión. También es importante que el espacio cuente con la luz adecuada: el sistema de iluminación artificial será ambiental para evitar puntos de sombra o un exceso de luz.

10.2 FATIGA MUSCULAR

Es producida por posturas incorrectas al sentarse y la ubicación inadecuada del equipo informático. Además, el mantenimiento prolongado de una determinada posición conduce a la fatiga de la musculatura. Las posturas forzadas o inadecuadas que deberán prevenirse serán aquellas que conlleven posiciones extremas de las diferentes partes del cuerpo (brazos elevados o completamente estirados, espalda inclinada o girada, etc.), así como aquellas que se mantengan de forma prolongada (estar mucho tiempo sentado).

10.3 CAÍDAS DE PERSONAL O GOLPE CONTRA OBJETOS

Los lugares de trabajo deben mantenerse limpios y ordenados, es decir, dejando libres de obstáculos las zonas de paso. El material de trabajo se deberá almacenar en estanterías y armarios.

10.4 CONTACTO ELÉCTRICO

Respetar las normas de seguridad básicas en el uso de los equipos eléctricos y revisar el estado de cada equipo antes de su uso.

10.5 CARGA MENTAL

El estrés en el trabajo o la desmotivación pueden ser un riesgo laboral y, por lo tanto, afectar a la salud de los trabajadores. Por ello, es importante realizar tareas variadas, realizar paradas periódicas para prevenir la fatiga, seguir hábitos de vida saludable y realizar ejercicio de forma habitual.

10.6 FACTORES EN LA ORGANIZACIÓN

Para mejorar la influencia del trabajador sobre su trabajo, es necesario:

- Permitir que la persona pueda organizar y planificar su trabajo.
- Dejar que los empleados aporten ideas a las decisiones y acciones que afecten a sus funciones.
- Proporcionar a cada miembro de la plantilla, en la medida de lo posible, un mayor control sobre sus tareas.

Para prevenir la aparición de estos riesgos es fundamental la adaptación ergonómica del área de trabajo, esto es: adaptar el puesto al mayor número de usuarios y realizar correctamente la distribución y diseño del espacio. Esto contribuirá a la disminución de lesiones, la reducción de fatiga mental y física, así como el aumento de la eficacia de los trabajadores.

Respecto a la ergonomía ambiental —además de una iluminación adecuada sin deslumbramientos o reflejos— es importante controlar o minimizar el ruido para que no dificulte la concentración de los trabajadores y mantener una temperatura adecuada.

Sin duda, el sector tecnológico está en auge desde hace muchos años y cada vez somos más conscientes de las necesidades de prevención de riesgos laborales en sectores como la informática. Por ello es más relevante que nunca la figura del especialista en prevención de riesgos laborales, pues el mercado laboral está cada vez más orientado a este tipo de profesiones, ligadas a la informática y áreas fuertemente tecnologizadas.

BIBLIOGRAFÍA

- DeveloperAndroid. (s.f.). Obtenido de
<https://developer.android.com/reference/androidx/security/crypto/EncryptedSharedPreferences>
- Code, C. (s.f.). Obtenido de <https://japarisivaldes.medium.com/clean-code-solid-y-test-arquitectura-mantenible-462f5efd5b23>
- culturaGeek. (s.f.). Obtenido de <https://culturageek.com.ar/que-edad-tiene-la-audiencia-de-los-videojuegos-de-pokemon/#:~:text=Cambiando%20de%20juego%2C%20una%20encuesta,%22%2C22%25>.
- dependencias, I. d. (s.f.). Obtenido de <https://developer.android.com/training/dependency-injection?hl=es-419>
- Developer android.* (s.f.). Obtenido de
<https://developer.android.com/reference/androidx/security/crypto/EncryptedSharedPreferences>
- Developer, A. (s.f.). Obtenido de
<https://developer.android.com/reference/androidx/security/crypto/EncryptedSharedPreferences>
- DraggerVsKoin. (s.f.). Obtenido de
https://cdn2.hubspot.net/hubfs/5078049/Media_files/whitepaper_Dagger_vs_Koin_spanish.pdf
- Figma. (s.f.). Obtenido de
<https://www.figma.com/file/N3TdKcqAyNXR4ZFwVWUpL/Pokelytics?type=design>
- Firebase, C. (s.f.). Obtenido de https://www.youtube.com/watch?v=KSW5jyWXs_Y&t=247s
- FirebaseAndroid. (s.f.). Obtenido de
https://www.youtube.com/watch?v=KYPC7CAYJOw&list=PLNdFk2_brsRdDzgiJSamO4vC0cAiRXcH4&index=24
- Github. (s.f.). Obtenido de <https://github.com/JoaquinAyG/Pokelytics/tree/develop>
- GoogleFonts. (s.f.). Obtenido de <https://fonts.google.com/?preview.size=24>
- <https://developer.android.com/reference/androidx/security/crypto/EncryptedSharedPreferences>. (s.f.).
<https://developer.android.com/reference/androidx/security/crypto/EncryptedSharedPreferences>. Obtenido de
<https://developer.android.com/reference/androidx/security/crypto/EncryptedSharedPreferences>:
<https://developer.android.com/reference/androidx/security/crypto/EncryptedSharedPreferences>
- Iconos. (s.f.). Obtenido de <https://www.flaticon.com/>

JUnit. (s.f.). Obtenido de <https://es.wikipedia.org/wiki/JUnit>

Logo. (s.f.). Obtenido de

<https://www.freelogodesign.org/manager/signin?r=https%3A%2F%2Fwww.freelogodesign.org%2Fmanager>

Mockito. (s.f.). Obtenido de <https://site.mockito.org/>

MuyInteresante. (s.f.). Obtenido de <https://www.muyinteresante.es/curiosidades/28635.html>

MVVM. (s.f.). Obtenido de <https://programacionymas.com/blog/android-mvc-mvp-mvvm>

pasos, J. p. (s.f.). Obtenido de https://programacion.net/articulo/primeros_pasos_con_junit_265

Retrofit. (s.f.). Obtenido de <https://devexperto.com/retrofit-android-kotlin/>

TDD. (s.f.). Obtenido de <https://www.paradigmadigital.com/dev/tdd-como-metodologia-de-diseno-de-software/>

Trello. (s.f.). Obtenido de <https://trello.com/invite/accept-board>

Unitarias, P. (s.f.). Obtenido de <https://es.parasoft.com/blog/junit-tutorial-setting-up-writing-and-running-java-unit-tests/>

Vandal. (s.f.). Obtenido de <https://vandal.elespanol.com/noticia/1350699142/nintendo-detalla-la-edad-media-de-sus-jugadores/>

Xataka. (s.f.). Obtenido de <https://www.xatakamovil.com/aplicaciones/la-pokedex-de-pokemon-ya-tiene-su-aplicacion-oficial-para-ios>

XatakaPokemon. (s.f.). Obtenido de <https://www.xatakamovil.com/aplicaciones/la-pokedex-de-pokemon-ya-tiene-su-aplicacion-oficial-para-ios>