

DESARROLLO AVANZADO DE SOFTWARE

4º curso, grupo 46

Segundo cuatrimestre

PRIMER PROYECTO INDIVIDUAL: BUDGET BUDDY

Maitane Urruela López de Echazarreta

Bilbao, 17 de marzo de 2024

ÍNDICE

Índice

1. BudgetBuddy	3
1.1. Interfaz del usuario	3
1.2. Pantallas	4
1.2.1. Pantalla principal y pantallas auxiliares	4
1.2.2. Facturas	5
1.2.3. Dashboards	5
1.3. Datos	6
2. Link al repositorio	6
3. Herramientas utilizadas en el proyecto	7
3.1. Kotlin	7
3.1.1. Interoperabilidad con Java	7
3.1.2. Orientación a objetos	8
3.1.3. Corrutinas y Flow	8
3.1.4. Otros tipos de clase	8
3.2. Jetpack Compose	9
3.3. Preferences Datastore	10
3.4. Room Database	10
3.5. Android ViewModels	11
3.6. Hilt	11
3.7. Navigation	12
4. Arquitectura del proyecto y sus clases	13
4.1. DI	13
4.2. Utils	13
4.3. UI Theme	14
4.4. VM	14
4.5. Preferences	15
4.6. Data	15
4.6.1. Enumeration	16
4.7. Navigation	16
4.8. Screens	16
4.9. Shared	17
5. Requisitos llevados a cabo	18
5.1. Requisitos mínimos	18
5.1.1. Listados de elementos	18
5.1.2. Base de datos local	18
5.1.3. Uso de diálogos	18

ÍNDICE DE FIGURAS

5.1.4. Notificaciones locales	19
5.1.5. Control del funcionamiento de la aplicación	19
5.2. Requisitos extra	20
5.2.1. Fragments	20
5.2.2. Multiidioma	21
5.2.3. Uso de ficheros de texto	21
5.2.4. Uso de preferencias	21
5.2.5. Estilos y temas propios	21
5.2.6. Uso de intents implícitos	22
5.2.7. ActionBar	22

Índice de figuras

1. Pantallas “Factura”, “Home” y “Dashboards”, respectivamente.	3
2. Pantallas auxiliares.	5
3. Entidades involucradas en los flujos de datos: consumidor, intermediarios opcionales y productor [4].	8
4. Arquitectura MVVM [9].	9
5. Arquitectura de Room Database [3].	11
6. Funcionamiento de <i>Navigation</i> [8].	12
7. Posibles temas de colores en la aplicación.	14
8. Ejemplo de <i>Scaffold</i> [7].	17
9. Diferentes diálogos en la aplicación.	18
10. Notificación de descarga.	19
11. Diferentes comportamientos en la visualización de la aplicación.	20
12. Correo electrónico predeterminado.	22

1. BudgetBuddy

BudgetBuddy es una aplicación Android diseñada para facilitar el registro, visualización, descarga y compartición eficiente e intuitiva de los gastos personales. La aplicación se centra en mantener un seguimiento diario de los gastos, permitiendo a los usuarios visualizarlos según su cuantía y categoría.

Además, ofrece tres opciones de temas de colores, siendo el predeterminado el verde (Figura 7), y admite la interacción en tres idiomas: español (predeterminado), inglés y euskera. La interfaz común de la aplicación se compone de un marco que abarca todas las pantallas, incluyendo las principales como “Home”, “Factura” y “Dashboards” (Figura 1), así como dos ventanas auxiliares denominadas “Agregar” y “Editar” (Figura 2).

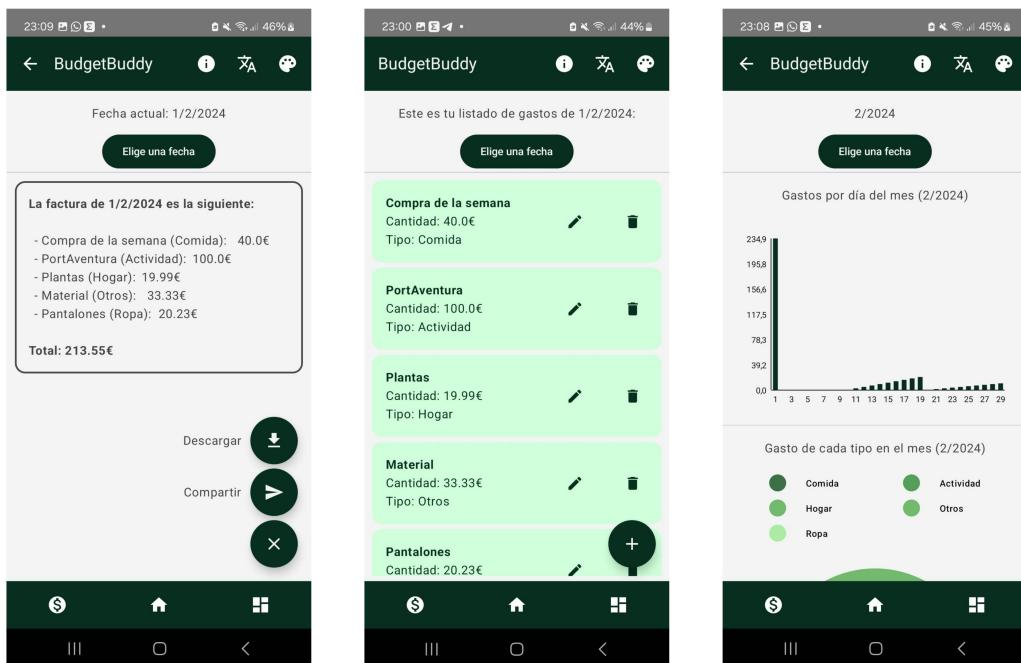


Figura 1: Pantallas “Factura”, “Home” y “Dashboards”, respectivamente.

1.1. Interfaz del usuario

El marco común de todas las pantallas, o *Scaffold* (Figura 8), consta de una barra superior y otra inferior, que en caso de estar en horizontal la pantalla, se convierten en una barra superior y una lateral izquierda, respectivamente. La barra superior presenta el nombre de la aplicación y una flecha para regresar a la pantalla principal (a menos que el usuario ya esté en ella). Además, cuenta con tres botones a la derecha:

- **Botón de información:** Al pulsar este botón, aparece un diálogo en el que se da información general de la APP, con la opción de realizar dos acciones: “Compartir” y “Email”. La primera opción permite enviar una invitación a otro usuario a través del canal de chat de elección del

usuario, mientras que la segunda abre el correo electrónico predeterminado del usuario con una plantilla dirigida al correo de BudgetBuddy (budgetbuddy46@gmail.com). Si el usuario no desea realizar ninguna de estas acciones, simplemente puede pulsar el botón “OK”.

- **Botón de idiomas:** Al hacer clic, aparecerá otro diálogo con la opción de cambiar el idioma entre inglés, euskera y castellano. Al seleccionar cualquiera de estos, la aplicación cambiará automáticamente de idioma y este se guardará en las preferencias del usuario. En caso de no querer cambiar, una vez más se pulsará “OK”.
- **Botón de tema:** Similar al anterior, abrirá una selección entre los temas de azul, verde y morado. Al elegir uno, este también se guardará en las preferencias y en caso de pulsar “OK” no se producirá ningún cambio.

En la barra inferior (o lateral en su defecto), aparecerán los iconos de las tres pantallas principales de “Home”, “Factura” y “Dashboards”, de forma que al seleccionar uno de ellos, se muestra la pantalla correspondiente al usuario. En el modo horizontal, se agrega un cuarto ícono de “Aregar” a la barra de navegación, haciendo desaparecer el botón flotante con la misma función de la pantalla inicial.

Además de estas secciones, estas tres pantallas cuentan con una parte superior con un texto relativo a su contenido particular y un botón para seleccionar una fecha. Este abre un calendario con la fecha por defecto del día actual (al no guardarla en las preferencias, al iniciar la aplicación siempre sale el día de la fecha actual). Más tarde el usuario podrá cambiarla para realizar gestiones de otro día, o visualizar los datos referentes a este.

1.2. Pantallas

1.2.1. Pantalla principal y pantallas auxiliares

La primera pantalla principal, conocida como “Home” (Figura 1), muestra todos los gastos realizados en un día a elección del usuario. Cada gasto se muestra en un recuadro a parte, visualizando su nombre, la cantidad asociada a este y el tipo de gasto al que pertenece. Además, cada uno de ellos cuenta con sus respectivos botones de borrado y edición. Este último, dará la opción a cambiar cualquier campo del gasto, incluida la fecha en la que se produjo. Tanto al borrar como al editar un elemento con éxito, aparecerá un mensaje en la parte inferior.

Como característica especial, esta pantalla cuenta con un botón flotante con el símbolo de “Añadir”, que al ser pulsado lleva al usuario a un formulario donde meter los datos de un nuevo gasto:

- **Nombre:** El nombre del gasto o una explicación de este en su defecto.
- **Tipo:** El tipo de gasto que es (por defecto a “Otros”).
- **Cantidad:** La cantidad en euros que ha supuesto ese gasto (en caso de querer añadir decimales se separa por puntos).
- **Fecha:** Da la opción de escoger la fecha del gasto mediante un calendario desplegable (por defecto con la fecha que venía seleccionada en la pantalla principal).

1 BUDGETBUDDY

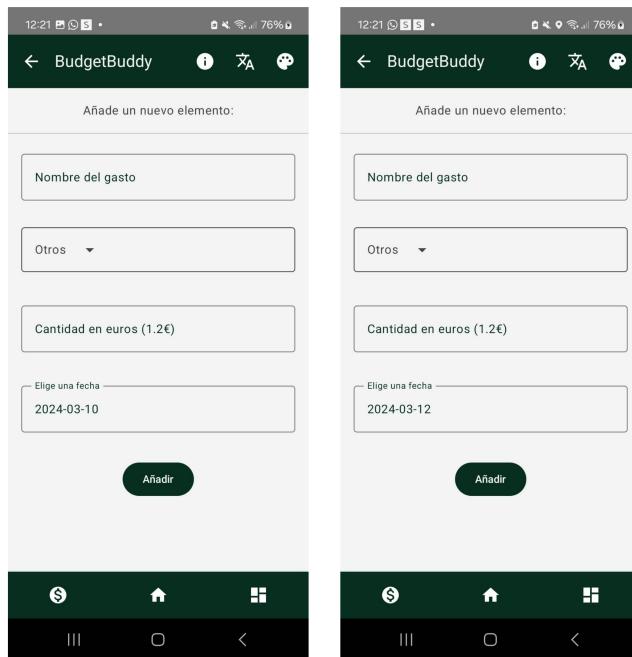


Figura 2: Pantallas auxiliares.

En caso de no introducir algo en todos los campos (salvo en los que tienen un valor por defecto), aparecerá una alerta, del mismo modo que al introducir el campo de la cantidad con una coma (u otro signo de puntuación) en vez de un punto. Esta pantalla sigue la misma estructura que la de edición (Figura 2).

1.2.2. Facturas

En esta segunda pantalla (Figura 1), aparece la factura asociada al día seleccionado. La factura cuenta con un encabezado con la fecha de la factura, un listado de todos los gastos, y la suma de todos estos al final. Como punto destacable de la pantalla y al igual que la anterior, cuenta con un botón flotante, en este caso desplegable, dando lugar a dos acciones posibles: "Descargar" y "Compartir".

Al descargar el fichero, se generará una notificación en el dispositivo, además de un fichero TXT en el directorio de descargas del teléfono. En cambio, al compartir, se enviará el contenido de la factura por el canal de mensajería de elección del usuario, al contacto seleccionado.

1.2.3. Dashboards

En esta última pantalla (Figura 1), al igual que en las dos anteriores, se permite al usuario seleccionar una fecha. Sin embargo, en este caso, se muestra información relativa a todo el mes seleccionado. Un gráfico de barras ilustra el gasto diario del mes, acompañado por una gráfica de donut que representa el porcentaje asociado a cada tipo de gasto en el mes completo (Figura 7).

2 LINK AL REPOSITORIO

1.3. Datos

Los datos por defecto se encuentran concentrados en los meses de marzo y abril. En caso de querer un conjunto de datos suficiente para desbloquear la función de *scroll* en la pantalla de “Home”, se puede acceder al día 12 de marzo de 2024.

2. Link al repositorio

La aplicación de BudgetBuddy se encuentra en el siguiente link:

https://github.com/Maits27/DAS_BudgetBuddy.git

3 HERRAMIENTAS UTILIZADAS EN EL PROYECTO

3. Herramientas utilizadas en el proyecto

En esta asignatura se utilizan las herramientas de Java, Android View, Shared Preferences y SQLite. Sin embargo, en este proyecto se han utilizado otras herramientas en busca de una alternativa más moderna y eficiente (con el previo consentimiento del profesor). Dichas herramientas han sido:

- **Kotlin.** Una alternativa a Java siendo un lenguaje más conciso y expresivo, facilitando así el desarrollo y mejorando la legibilidad del código.
- **Jetpack Compose.** Ha sido utilizado en lugar de Android View, puesto que proporciona una forma más declarativa y flexible de construir interfaces de usuario, simplificando la creación y manipulación de componentes visuales.
- **Datastore.** Esta alternativa a Shared Preferences, es una solución más robusta y segura de almacenar datos en formato de clave-valor de manera asíncrona, mejorando la persistencia de la información.
- **Room database.** Mediante este framework, se genera una capa de abstracción de la base de datos que permite el manejo de las mismas operaciones relacionadas con esta de manera más eficiente.

Esta transición no solo se alinea con las tendencias y recomendaciones actuales de desarrollo de aplicaciones Android, sino que también optimiza la productividad y calidad del código. Además, para poder implementar la aplicación, han sido necesarias las herramientas de ViewModels de Android y Hilt.

3.1. Kotlin

Kotlin es un lenguaje de programación utilizado por *Google* en desarrollo de aplicaciones Android, que hace que la codificación sea concisa y multiplataforma [13]. Este lenguaje creado por *JetBrains* es un lenguaje de código abierto y de tipo estático que cuenta, entre otras, con las características explicadas a continuación.

3.1.1. Interoperabilidad con Java

La interoperabilidad con Java es una característica fundamental de Kotlin. Esto se logra gracias a que Kotlin está diseñado para ser completamente compatible con la sintaxis y las estructuras de código de Java, permitiendo a los desarrolladores integrar módulos de ambos en el mismo proyecto [1].

La interoperabilidad bidireccional entre Kotlin y Java es posible gracias a que Kotlin compila a bytecode de Java, lo que significa que ambos lenguajes comparten el mismo entorno de ejecución en la máquina virtual de Java (JVM). Esta integración fluida facilita la transición gradual de proyectos existentes de Java a Kotlin, pudiendo adoptar Kotlin de manera progresiva sin tener que reescribir todo el código existente.

3 HERRAMIENTAS UTILIZADAS EN EL PROYECTO

3.1.2. Orientación a objetos

Aunque en el desarrollo de aplicaciones móviles lo habitual sea utilizar un paradigma orientado a objetos, este lenguaje también proporciona funcionalidades que permiten una programación más expresiva y concisa, integrando conceptos de programación funcional.

En Kotlin, no todas las funciones necesariamente son métodos de clases y no todas las propiedades deben ser atributos de objetos, por lo que permite la definición de funciones que no están vinculadas a una clase específica. Estas funciones de orden superior ofrecen modularidad al código, pudiendo ser usadas en más de un lugar. Además, admite funciones anónimas y lambdas.

3.1.3. Corrutinas y Flow

También posee corrutinas para optimizar la programación asíncrona, simplificando así el acceso a la base de datos por ejemplo, y evitando bloqueos de la aplicación en este tipo de operaciones [1].

Para ello, implementa los *Flows*, un concepto que permite emitir varios valores de manera secuencial, en lugar de funciones que suspenden el proceso para mostrar un único valor. Un flujo se puede usar, por ejemplo, para recibir actualizaciones en vivo de una base de datos.

Un *Flow* es muy similar a un *Iterator* que produce una secuencia de valores. Hay tres entidades involucradas en esta transmisión de datos [4]:

- **El productor** que produce datos que se agregan al flujo. Gracias a las corrutinas, los flujos también pueden producir datos de forma asíncrona.
- **Los intermediarios (opcional)** que pueden modificar cada valor emitido en el flujo, o bien el flujo mismo.
- **El consumidor** que consume los valores del flujo.

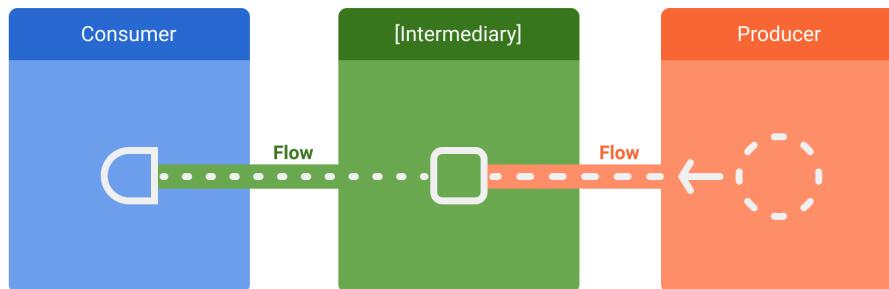


Figura 3: Entidades involucradas en los flujos de datos: consumidor, intermediarios opcionales y productor [4].

3.1.4. Otros tipos de clase

Por un lado, implementa clases tipo *singleton*, que implica que estas puedan instanciarse una única vez, evitando duplicados de la misma.

3 HERRAMIENTAS UTILIZADAS EN EL PROYECTO

También utiliza ***data class***, que se utilizan principalmente para el almacenamiento de datos [12]. Para cada una de estas, el compilador genera automáticamente funciones tales como: *copy*, *equals*, *toString*, etc.

Por último, utiliza las ***enum class***, para generar instancias con un número de valores predeterminado. Se utilizan en situaciones en las que se espera un conjunto específico de opciones, como los días de la semana, los meses del año, o cualquier otro conjunto discreto de valores.

3.2. Jetpack Compose

Jetpack Compose es un kit de herramientas diseñado para simplificar y agilizar el desarrollo de interfaces de usuario (IU) nativas en aplicaciones Android. Este *framework* se basa en un concepto totalmente declarativo, lo que significa que se describe la IU (de elementos jerárquicos, conteniendo unos a otros) en un estado en concreto. En el momento en el que ese estado cambia (debido a un cambio en los datos o en uno de los elementos), el *framework* se vuelve a ejecutar, únicamente en las zonas donde ese cambio afecte, actualizando así la jerarquía de la IU [2].

Las funciones marcadas con `@Composable`, son lo que se conoce como funciones de componibilidad. Estas son las únicas que pueden llamar a las de su mismo tipo y son lo único necesario para generar un componente IU. Es la anotación la que indica a Compose que agregue la compatibilidad necesaria a la función para actualizar y mantener la IU con el tiempo.

Resumiendo, la función de la capa de la IU (capa de presentación) consiste en mostrar los datos de la aplicación en la pantalla. De modo que, cuando los datos cambian debido a una interacción del usuario, la IU debe actualizarse para reflejar los cambios. Esta capa consta de los siguientes componentes:

- Elementos IU: Componentes que renderizan los datos en la pantalla (composables).
- Contenedores de estado: Componentes que contienen los datos, los exponen a la IU y controlan la lógica de la app, por ejemplo un ViewModel.

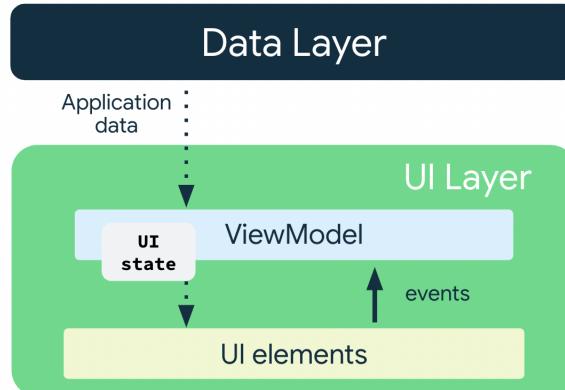


Figura 4: Arquitectura MVVM [9].

3 HERRAMIENTAS UTILIZADAS EN EL PROYECTO

Además al integrar Jetpack Compose con la arquitectura MVVM (*Model View ViewModel*) en Android, se consigue una arquitectura sólida en el desarrollo de la aplicación, ya que ofrece una clara separación entre la parte de la interfaz del usuario (*View*) de la parte lógica (*Model*), logrando así que la parte visual sea totalmente independiente [9].

En este proyecto, el *View* correspondería con los elementos de la IU, mientras que el *Model* lo constituirían la base de datos Room Database y el Datastore Preferences, tomando como intermedio el *ViewModel* (*ViewModels* de Android). Además se ha implementando el patrón *Repository* para mejorar la interacción entre el *ViewModel* y las partes lógicas.

3.3. Preferences Datastore

A partir de Android Jetpack 11, se recomienda el uso de Preferences Datastore sobre Shared Preferences puesto que lo supera en términos de rendimiento, especialmente cuando se trabaja con grandes cantidades de datos. Además, utiliza una estrategia I/O de disco más eficiente y aprovecha las rutinas y flujos de Kotlin para manejar las actualizaciones de datos de forma asíncrona.

Por otro lado, SharedPreferences solo admite tipos de datos primitivos, lo que requiere conversiones de tipos manuales cuando se trabaja con objetos más complejos. DataStore también proporciona APIs con seguridad de tipos, lo que facilita el trabajo con diferentes tipos de datos [11].

Preferences DataStore es ideal para conjuntos de datos pequeños y simples, como el almacenamiento de datos de inicio de sesión, la paleta de colores de la APP, etc. Sin embargo, no es adecuado para conjuntos de datos complejos. La biblioteca Jetpack DataStore ofrece dos implementaciones diferentes: Preferences DataStore y Proto DataStore.

La diferencia entre ellas reside en que la primera utiliza la estructura clave-valor, sin definir un esquema (modelo de base de datos) por adelantado, mientras que Proto DataStore define el esquema mediante búferes de protocolo [6]. En este caso se ha implementado el DataStore, debido a que no se requieren estructuras complejas en las preferencias de usuario.

3.4. Room Database

La biblioteca de persistencias Room del grupo de librerías de Jetpack Compose ofrece una capa de abstracción para SQLite que permite hacer el mismo uso de una base de datos, sin necesidad de escribir las consultas SQL a mano. Teniendo en cuenta la facilidad de uso y seguridad que supone al mismo tiempo, Android recomienda el uso de Room frente a las APIs de SQLite [3]. En particular, Room brinda los siguientes beneficios:

- Verifica el tiempo de compilación de las consultas y advierte al desarrollador en caso de posible bloqueo, forzando el uso de corrutinas.
- Anotaciones de conveniencia obviando código estándar repetido que de lugar a errores.
- Rutas de migración de bases de datos optimizadas.

3 HERRAMIENTAS UTILIZADAS EN EL PROYECTO

Los componentes principales de este *framework* son tres:

1. **La clase de la base de datos**, que contiene la base de datos y sirve como punto de acceso.
2. **Las entidades de datos**, anotadas con `@Entity`, lo que sería el equivalente a las tablas de SQLite y sus respectivos atributos y funciones.
3. **Los objetos de acceso a datos (DAOs)** que proporcionan los métodos de inserción, borrado, actualización y consulta (anotados con `@Insert`, `@Delete`, `@Update` y `@Query`, respectivamente).

La clase de base de datos proporciona los DAOs asociados a esta al resto de la aplicación. A su vez, la APP puede usar los DAOs para recuperar datos como instancias. La aplicación también puede hacer uso de las entidades de datos definidas para actualizar filas de las tablas o insertar elementos nuevos en estas (Figura 5).

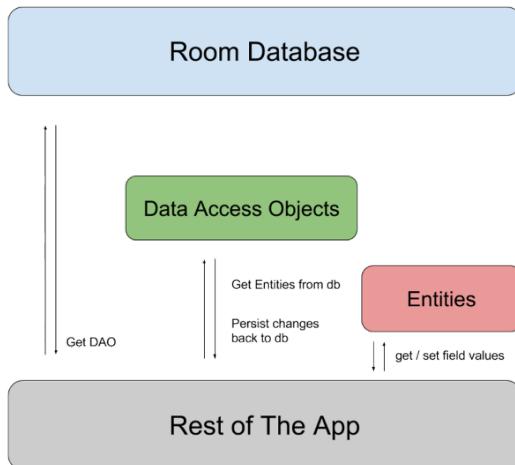


Figura 5: Arquitectura de Room Database [3].

3.5. Android ViewModels

Esta librería de Jetpack facilita la implementación del patrón *ViewModel*. Este almacena los datos relativos a la APP, evitando que se destruyan cuando el *framework* de Android recrea la actividad, puesto que, al ser independiente la actividad, los objetos *ViewModel* no se destruyen [9].

3.6. Hilt

Hilt es una biblioteca de inserción de dependencias para Android que evita tener que construir cada clase y sus dependencias de forma manual, y usar contenedores para reutilizar y administrar las dependencias.

Este framework basado en Dagger proporciona una forma estándar de usar la inserción de dependencias en la aplicación, proporcionando contenedores para cada clase de Android en el proyecto y administrando su ciclo de vida [5].

3.7. Navigation

En Compose, las múltiples actividades no son necesarias, pudiendo implementar el cambio entre pantallas y entre diferentes fragmentos, mediante el componente *Navigation*. Este componente tiene tres partes principales:

- **NavController:** Responsable de navegar entre los destinos, es decir, las pantallas en la APP.
- **NavGraph:** Realiza la asignación de los destinos componibles a los que se navegará.
- **NavHost:** Es el elemento componible que funciona como contenedor para mostrar el destino actual del *NavGraph*.

Uno de los conceptos fundamentales de la navegación en una aplicación de Compose es la ruta, un *string* que se corresponde con un destino (o pantalla). Esta idea es similar al concepto de una URL, ya que, como una URL diferente se asigna a una página diferente en un sitio web, una ruta es una *string* que se asigna a un destino y sirve como su identificador único [10]. Estas rutas se definen en el AppScreens, como se explica en el apartado 4.7.

En caso de querer acceder a una de esas rutas, haciendo uso del método *navigate* del *NavController*, se le pasará la ruta a la que se quiere acceder. Este componente también lleva un registro de las pantallas visitadas en una pila, lo que permite volver a la anterior haciendo uso del método de *navigateUp*.

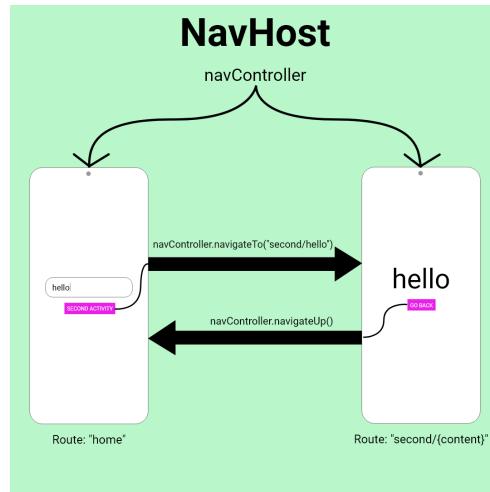


Figura 6: Funcionamiento de *Navigation* [8].

4. Arquitectura del proyecto y sus clases

El proyecto está dividido en 30 ficheros de tipo Kotlin. La mayoría de ellos solo tienen componibles que se utilizan en otro u otros ficheros o funciones de uso general en la aplicación.

En el directorio raíz, solo se encuentran dos archivos que son *BudgetBuddyApp.kt* y *MainActivity.kt*, puesto que, al solo tener una actividad el proyecto, se ha mantenido aquí.

El primero de esos dos ficheros, contiene la clase **BudgetBuddyApp**, que hereda de **Application**. A simple vista, el fichero es irrelevante para el funcionamiento de la aplicación, debido a que se encuentra prácticamente vacío. Sin embargo, es totalmente necesario para el correcto funcionamiento de la librería Hilt.

Por otro lado, en la clase **MainActivity** del segundo fichero (que hereda de **AppCompatActivity**), se define la actividad principal que carga la interfaz. Aquí se crean los *ViewModel* de la aplicación y preferencias, se solicitan los permisos de notificación (en caso de no estar ya dados) y se crea el canal de notificaciones correspondiente. Además, se ha añadido la función para descargar ficheros, que también requiere de solicitar permisos de descarga y almacenamiento.

Los demás ficheros se han almacenado en diferentes carpetas.

4.1. DI

Esta carpeta de *Dependency Injection*, contiene el único fichero de *AppModule.kt* con el objeto (*singleton*) de **AppModule**. Al igual que la clase de tipo **Application**, este objeto lo utiliza Hilt para definir los objetos (interfaces, DAOs...) que se deben inyectar en otras clases y cómo se deben inyectar.

Todos estos irán etiquetados como *@Singleton*, ya que solo debe existir una instancia de cada uno. Al estar a nivel de aplicación, no se destruirán hasta que la aplicación se cierre por completo.

4.2. Utils

Esta carpeta contiene dos ficheros: *TypeChange.kt* y *LanguageConfigUtils.kt*.

El primero se utiliza para implementar funciones sobre tipos de datos ya existentes. Más concretamente, se ha implementado la conversión de *LocalDate* a *Long* y viceversa (no confundir con los *TypeConverter* que son utilizados por Room Database).

En el segundo fichero, se ha creado la clase singleton de **LanguageManager** que se encarga de cambiar el idioma de la aplicación Android. Gracias a las nuevas implementaciones de las APIs de AppCompat, ha sido posible realizar el cambio utilizando directamente el método de *setApplicationLocales* que delega a AppCompat el cambio de idioma en la aplicación local y sin necesidad de reiniciar la actividad a mano.

4.3. UI Theme

Compose está diseñado para admitir los principios de Material Design, implementando la mayor parte del diseño de sus elementos directamente desde aquí [2]. En esta carpeta es donde se define el tema a utilizar en toda la aplicación, concretamente en los ficheros *Color.kt* y *Theme.kt* (en este caso no se ha hecho cambios en el tipo de letra).

Mientras que en el primer fichero se definen los colores de la aplicación, dándoles un nombre a su forma hexadecimal, el segundo los utiliza para definir dónde y en qué condiciones se usan. Más concretamente, se ha definido un composable llamado *BudgetBuddyTheme*, que se aplica desde el **MainActivity** a toda la aplicación, pasándole como parámetro cuál de las tres paletas de colores debe proporcionar. Las paletas de colores son las que aparecen en la Figura 7.

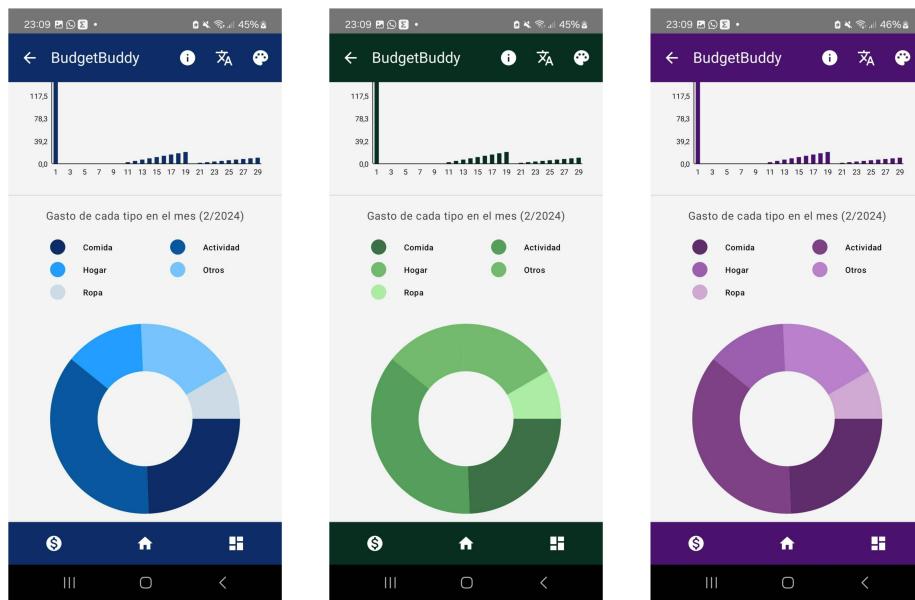


Figura 7: Posibles temas de colores en la aplicación.

4.4. VM

Esta carpeta contiene los *ViewModel* tanto de los datos de la aplicación como de las preferencias, mediante las clases **AppViewModel** y **PreferencesViewModel**, ambas heredando de **ViewModel** y siendo etiquetadas como `@HiltViewModel`, para que el *framework* sea capaz de inyectar las interfaces de los repositorios que conectan estas clases con Room y Datastore.

El *ViewModel* de la APP, contiene todos los *Flows* de los datos necesarios para el correcto funcionamiento de la APP. Estos *Flow* (como se ha explicado en la sección 3.1.3), proporcionan secuencias de valores con los resultados a las consultas definidas en el DAO. Estas secuencias se actualizan en tiempo real a medida que se realizan cambios en la base de datos y posteriormente se recoge el último valor a la hora de imprimirlos en pantalla en el formato definido.

De la misma manera, el **PreferencesViewModel** recoge los flujos de los valores del idioma (tipo *AppLanguage*) y el tema (tipo *Int*) y a la hora de utilizarlos, recoge el último valor registrado en el flujo. En caso de suceder alguna alteración, el cambio sería inmediato.

Estos hacen de conexión entre sus respectivos repositorios (a través de sus interfaces) y los elementos de la UI siguiendo el patrón MVVM. Todas las dependencias que usan son inyectadas por Hilt.

4.5. Preferences

Esta carpeta guarda los ficheros de *IGeneralPreferences.kt*, que es la interfaz antes mencionada del *Repository* utilizado para la conexión con el Datastore y *PreferencesRepository.kt*, el fichero que recoge la implementación del repositorio.

En este último, se crea el Datastore dentro del contexto de la aplicación y después se implementa la clase *singleton PreferencesRepository*, que en una primera instancia define las claves que se usarán en el Datastore y más adelante los métodos *getter* y *setter* para cada una de estas.

4.6. Data

En esta carpeta se guardan, por un lado todos los ficheros necesarios para la implementación de la base de datos de Room y por otro una sub-carpeta con las clases *enumerate*.

Entre los ficheros de Room se encuentran, como aparece en la sección 3.4 la clase de la base de datos que hereda de **RoomDatabase (Database)** y recoge como entidad la clase **Gasto**. Esta entidad se encuentra en el fichero de *DataTables.kt* marcada con *@Entity*. Este fichero también recoge los *data class* de **GastoDia**, **GastoTipo** y **Diseño**, que se usan en los flujos de los composable de *MainView* y *Dashboards* (sección 4.8).

Dentro de **Gasto**, se utiliza un tipo de dato *LocalDate*, que la base de datos no reconoce. Es por esto que en el fichero *Converter.kt* se implementa la clase **Converters** con dos funciones etiquetadas con *@TypeConverter* para pasar este tipo de dato a *Long* y viceversa de forma automática. Esta clase también se le pasa a la base de datos.

Con esta entidad interactúa la interfaz del DAO (**GastoDao**) que se recoge en el fichero *Dao.kt* de la misma carpeta. En esta interfaz se definen todas las consultas que se pueden realizar a la base de datos (no se podrá realizar ningún tipo de consulta que no aparezca aquí recogida).

En este tipo de interfaz no es necesario definir las consultas SQL explícitamente, por ejemplo, en caso de querer hacer una inserción, basta con introducir un elemento del tipo recogido en las entidades y el *framework* se encarga de mapearlo (sección 3.4). La única excepción es con las *@Query*.

Finalmente, este DAO se llama desde la clase *singleton GastoRepository* del fichero *Repository.kt* (donde se inyecta mediante Hilt la interfaz correspondiente al repositorio, que ha sido definida en el mismo fichero), completando así la estructura de la Figura 5.

4.6.1. Enumeration

En la sub-carpetas mencionadas anteriormente se recogen tres ficheros *AppLanguage*, *Tema.kt* y *TipoGasto.kt* que implementan cada uno la clase **Enumeration** con el nombre del fichero y un método que devuelve el nombre del objeto en el idioma de preferencia. La excepción a esto es **AppLanguage**, que recoge estos idiomas y devuelve el tipo en base al código de idioma (string) proporcionado.

4.7. Navigation

Aquí se encuentra un único fichero llamado *AppScreens* que contiene una **sealed class** con el mismo nombre. Dentro se recogen todas las pantallas a las que se puede acceder desde el *NavHost* y sus nombres (sección 4.8). Una **sealed class** implica que los objetos de esta son inmutables, por lo que no se pueden editar, añadir o borrar y todos los elementos en esta se definen como *Object*.

A la clase se le pasa como parámetro un *string* con la ruta (en este caso el nombre del objeto y nombre de la pantalla).

4.8. Screens

En esta carpeta se guardan todos los ficheros con las pantallas a las que se puede acceder desde el *NavHost* (“Home”, “Dashboards”, “Facturas”, “Add” y “Edit”), además del *MainView*.

El **MainView** es el composable principal de la APP, que recoge el *Scaffold* que define la estructura principal de la APP (Figura 8). Este elemento proporciona una API sencilla para ensamblar la estructura general de la aplicación rápidamente. *Scaffold* acepta varios **@Composable** como parámetro:

- **TopBar**: Es la barra de la APP en la parte superior de la pantalla (en este caso utilizada como *ActionBar*).
- **BottomBar**: Barra de la aplicación que se muestra en la parte inferior de la pantalla. En este caso siendo utilizada como *Navigation Bar* en caso de estar en modo vertical (al estar en horizontal esta desaparecería, dando lugar a un menú lateral).
- **FloatingActionButton**: Un botón que se desplaza sobre la esquina inferior derecha de la pantalla, utilizado para exponer acciones clave. En esta aplicación, su función y presencia cambia en base a la pantalla visualizada (sección 1.1).

Por último, en la *trailing lambda*, el *Scaffold* define su contenido. Aquí se ha decidido añadir directamente el *NavHost* de forma que, dependiendo de la ruta que se le proporcione, se visualice el composable correspondiente a esta.

Los demás ficheros de la carpeta tienen los composable relativos a la pantalla que representan. En caso de tener alguno de estos elementos en común entre las pantallas, este se encontraría en la carpeta “Shared”.

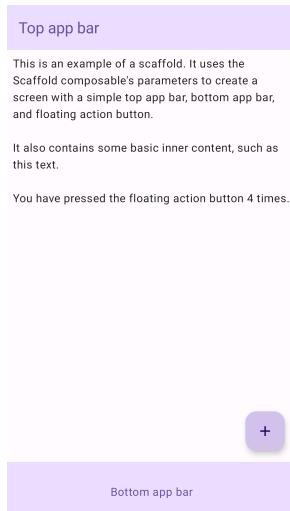


Figura 8: Ejemplo de *Scaffold* [7].

4.9. Shared

En esta carpeta se guardan todos los composable y alertas comunes entre las pantallas, además de las notificaciones e *intents* implícitos.

Por un lado, en el fichero *Comunes.kt* se implementan las funciones como **NoData** (pantalla que aparece ante la falta de datos), **Calendario** o **Header** (recuadro superior del contenido con la explicación de la pantalla y el botón para cambiar la fecha), que aparecen en al menos todas las pantallas principales.

Los **AlertDialog** como la información de la APP, los diálogos para cambiar las preferencias o las alertas de errores, se definen en *Alerts.kt*. También guarda el **Toast** que se imprime en caso de edición o borrado de un elemento. Estos composable se han separado de los anteriores por su naturaleza de aviso, pero son usados de la misma manera.

Por último, en *Notifications.kt* se guarda la implementación de la notificación en caso de descarga. Además de una función que sirve para compartir contenido mediante *intents* implícitos. La naturaleza de estos *intents* cambiará en base a los parámetros recibidos (correo electrónico o mensaje general).

5. Requisitos llevados a cabo

5.1. Requisitos mínimos

5.1.1. Listados de elementos

Como se puede ver en la pantalla de “Home”, este requisito se ha completado haciendo uso de una *LazyColumn* (siendo el equivalente en Compose a las *CardView*).

Como se ha explicado anteriormente, en esta lista se muestra un listado de los gastos realizados por el usuario en el día escogido, siendo cada uno de estos elementos representado por un *Card* con las características del elemento y dos botones: de borrar y editar.

Una *LazyColumn* (como sucede en las *RecyclerView*) es eficiente a la hora de manejar listas grandes de elementos, ya que solo renderiza los elementos visibles en la pantalla. Esto mejora significativamente el rendimiento al trabajar con conjuntos de datos extensos. Con respecto a los componentes en Java, este composable permite personalizar la lista de manera rápida y eficiente.

5.1.2. Base de datos local

En este proyecto se emplea la base de datos local Room como aparece en el apartado 3.4. En la sección de 4.6 aparece el proceso seguido a la hora de realizar las consultas de inserción, borrado, edición y selección de los gastos guardados en la base de datos.

Como uso avanzado cabe destacar que el empleo de Room Database no ha sido dado en clase, además de que se han usado los flujos para acceder a los datos de esta, optimizando el tiempo de recarga al realizar cambios en los datos.

5.1.3. Uso de diálogos

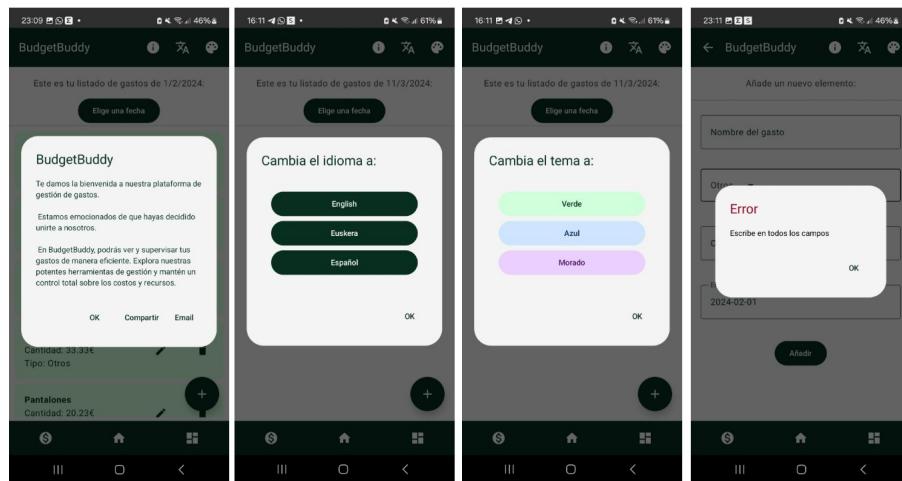


Figura 9: Diferentes diálogos en la aplicación.

5 REQUISITOS LLEVADOS A CABO

Como aparece en la sección 4.9 se han utilizado varios diálogos en toda la aplicación (Figura 9), que se guardan en el fichero *Alerts.kt* de la carpeta *Shared*. Los diálogos son los siguientes:

- **Panel de información** de la aplicación, desplegado al pulsar en el icono de información de la *ActionBar* superior de la APP.
- **Menú de selección de idioma** que aparece al pulsar el icono de idiomas en la misma barra.
- De igual manera que los dos anteriores, el **diálogo que se crea para la elección de temas** de colores al pulsar el icono de paleta de colores en la parte superior derecha.
- **Mensajes de error** al hacer un mal uso de los formularios o al haber errores de ejecución.

5.1.4. Notificaciones locales

Se ha implementado una notificación local en caso de descarga de un fichero factura. Esta avisa de que el fichero ha sido almacenado correctamente y el directorio del dispositivo donde ha sido almacenado.

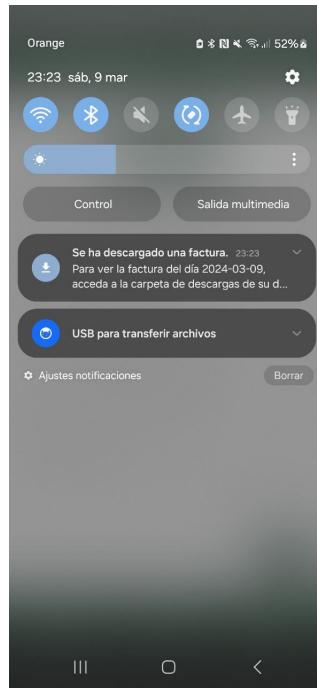


Figura 10: Notificación de descarga.

5.1.5. Control del funcionamiento de la aplicación

Se ha asegurado el mantenimiento del estado de la aplicación ante cualquier tipo de interrupción. Esto se ha logrado haciendo uso de variables que utilizan tanto *remember* como *rememberSaveable* (el segundo mantiene los datos aunque se interrumpa el funcionamiento de la aplicación momentáneamente). El estado se mantiene al:

5 REQUISITOS LLEVADOS A CABO

- Girar la pantalla del dispositivo, manteniendo los datos de cualquier ventana.
- Cambiar el *Navigation Bar* inferior por el lateral junto con el giro de pantalla.
- Cualquier actualización de la base de datos.
- Cambio en las variables del *ViewModel*. Los cambios en estas se visualizarán al momento, sin afectar a los demás datos.

El único caso donde no se mantiene el estado (hecho a propósito), son las alertas de error. Que al girar el dispositivo o realizar otra acción desaparecen. El resto de diálogos (información, idiomas y temas), se mantienen ante cualquier interrupción.

Además, se ha controlado la pila de ventanas haciendo uso del *NavHost*. Se ha implementado de tal forma que como máximo hayan dos pantallas en la pila: “Home” (la pantalla principal) y cualquiera de las otras pantallas o pantallas auxiliares. De esta forma, el espacio en memoria de la pila no se acumula a medida que se navega por las diferentes pantallas.

El uso de este componente se puede considerar avanzado, puesto que no se ha visto en la asignatura y es una alternativa más eficiente a las diferentes actividades.

5.2. Requisitos extra

5.2.1. Fragments

De por si, en Compose los composable se encargan de cambiar la forma de visualizar los elementos al girar la pantalla en el teléfono. Sin embargo, para cumplir con este requisito se ha forzado un cambio más visual a la hora de poner la pantalla en horizontal.

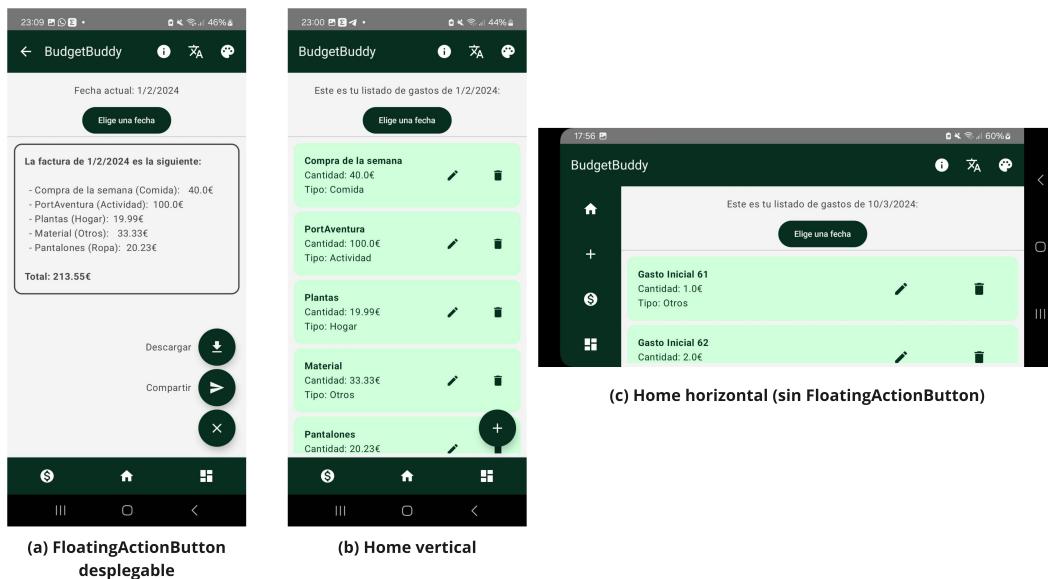


Figura 11: Diferentes comportamientos en la visualización de la aplicación.

5 REQUISITOS LLEVADOS A CABO

Como ya se ha mencionado varias veces, a la hora de girar la pantalla se ha establecido que desaparezca el *BottomBar* del *Scaffold*, apareciendo un menú lateral para sustituirlo. Además, en ese menú lateral también aparece la opción de acceder a la pantalla de “Add” que en vertical se quedaría como *FloatingActionButton*, únicamente en la pantalla “Home”.

En cuanto a este *FloatingActionButton*, también tiene un comportamiento diferente según en qué pantalla se encuentre. Es decir, en “Home” y vertical será el botón de “Add”, en la pantalla de “Facturas” se vuelve un botón desplegable con dos opciones (descarga y envío) y en caso de estar en “Dashboards” o en “Home” en modo horizontal desaparecería. Si no hay datos en la pantalla de “Factura” también desaparece.

5.2.2. Multiidioma

Como se ve en la Figura 9, existe la opción de cambiar el idioma de la visualización de la aplicación, las notificaciones y los *intents* implícitos. Por ejemplo, al enviar el email al correo de la aplicación (en la pestaña de información mediante el botón “Email”), el asunto y contenido por defecto también cambiarán de idioma.

El idioma por defecto de la APP es el inglés (o el idioma por defecto del teléfono en caso de que sea uno de los tres disponibles), pero también existe la posibilidad de cambiar el idioma a euskera y castellano.

5.2.3. Uso de ficheros de texto

En la pantalla de facturas, existe la posibilidad de descargar el contenido de la factura que se visualiza en formato TXT en el directorio de descargas del dispositivo Android.

5.2.4. Uso de preferencias

Para guardar las preferencias se hace uso de Preferences DataStore. El usuario puede guardar en sus preferencias tanto el idioma como la paleta de colores que prefiera, de forma que al volver a abrir la aplicación se carguen automáticamente.

5.2.5. Estilos y temas propios

Como se menciona en el apartado 4.3 Compose admite los principios de Material Design de forma que se aplique el diseño establecido en estos ficheros en toda la aplicación.

Se han implementado tres paletas de colores distintas de manera global para toda la aplicación, además de definir colores en vez de usarlos de manera hexadecimal. De manera puntual, también se ha cambiado el diseño de textos (como el de *NoData*) o la visualización de diferentes componentes (los elementos de los gráficos en “Dashboards”).

5 REQUISITOS LLEVADOS A CABO

5.2.6. Uso de intents implícitos

Se ha hecho uso de dos tipos de *intents* implícitos. El primero consta del envío de texto plano por el canal de mensajería de preferencia del usuario en cada momento. Este se implementa tanto en el envío de la factura, como en el diálogo de información al poder compartir información de la aplicación con otras personas.

Por otro lado, se hace uso del *intent* implícito que hace uso del correo electrónico de preferencia del usuario. Este envía el contenido establecido a una URI (en este caso un email al correo de budgetbuddy46@gmail.com). Se ha modificado este *intent* de forma que el correo (que se puede enviar a través del panel de información en la barra superior de la APP), venga con un asunto y contenidos predeterminados, Y no solo abriendo la aplicación del correo electrónico.



Figura 12: Correo electrónico predeterminado.

5.2.7. ActionBar

Como ya ha aparecido en otras secciones, se ha implementado una *ActionBar* que se encuentra en la parte superior de la pantalla, dando la opción a realizar tres acciones diferentes, además de la de ir hacia atrás en pantallas que no sean la inicial (sección 1.1).

Además, se ha implementado la *Navigation Bar* inferior de forma que el usuario navegue entre las diferentes pantallas.

Referencias

- [1] MyTaskPanel Consulting. *El lenguaje de programación Kotlin: qué es y para qué sirve.* URL: <https://www.mytaskpanel.com/lenguaje-de-programacion-kotlin/#:~:text=%C2%BFQu%C3%A9%20es%20Kotlin%3F,utilizar%20para%20desarrollar%20aplicaciones%20Android..>
- [2] Android Developers. *Aspectos básicos de Jetpack Compose.* URL: <https://developer.android.com/codelabs/jetpack-compose-basics?hl=es-419#0>.
- [3] Android Developers. *Cómo guardar datos en una base de datos local usando Room.* URL: <https://developer.android.com/training/data-storage/room?hl=es-419>.
- [4] Android Developers. *Flujos de Kotlin en Android.* URL: <https://developer.android.com/kotlin/flow?hl=es-419>.
- [5] Android Developers. *Inserción de dependencias con Hilt.* URL: <https://developer.android.com/training/dependency-injection/hilt-android?hl=es-419>.
- [6] Android Developers. *Preferences DataStore.* URL: <https://developer.android.com/codelabs/basic-android-kotlin-training-preferences-datastore?hl=es-419#2>.
- [7] Android Developers. *Scaffold.* URL: <https://developer.android.com/jetpack/compose/components/scaffold?hl=es-419>.
- [8] Android Developers. *Understand Jetpack Compose Navigation in 3 Minutes.* URL: <https://medium.com/@cybercoder.naj/compose-navigation-in-3-minutes-5cff3c57c34e>.
- [9] Android Developers. *ViewModel y el estado en Compose.* URL: <https://developer.android.com/codelabs/basic-android-kotlin-compose-viewmodel-and-state?hl=es-419#3>.
- [10] Nishant Aanjaney Jalan. *Cómo definir rutas y crear un NavController.* URL: <https://developer.android.com/codelabs/basic-android-kotlin-compose-navigation?hl=es-419#3>.
- [11] Muhammad Humza Khan. *Simplifying Data Storage in Android: SharedPreferences and Preferences DataStore.* URL: <https://medium.com/@humzakhalid94/simplifying-data-storage-in-android-sharedpreferences-and-preferences-datastore-e775825d8d0c>.
- [12] Kotlin. *Data classes.* URL: <https://kotlinlang.org/docs/data-classes.html>.
- [13] Kotlin. *Kotlin for Android.* URL: <https://kotlinlang.org/docs/android-overview.html>.