Capitulo 3

UD03. FRAGMENTOS Y NAVEGACIÓN

Resultados de avaliación

RA1. Aplica tecnoloxías de desenvolvemento para dispositivos móbiles, e avalía as súas características e as súas capacidades.

RA2. Desenvolve aplicacións para dispositivos móbiles, para o que analiza e emprega as tecnoloxías e as librarías específicas.

Criterios de avaliación

- CA1.6 Describíronse os perfís que establecen a relación entre o dispositivo e a aplicación.
- CA1.7 Analizouse a estrutura de aplicacións existentes para dispositivos móbiles, e identificáronse as clases utilizadas.
- CA1.8 Realizáronse modificacións sobre aplicacións existentes.
- CA1.9 Utilizáronse emuladores para comprobar o funcionamento das aplicacións.
- CA2.1 Xerouse a estrutura de clases necesaria para a aplicación.
- CA2.4 Utilizáronse as clases necesarias para a conexión e a comunicación con dispositivos sen fíos.
- CA2.5 Utilizáronse as clases necesarias para o intercambio de mensaxes de texto e multimedia.
- CA2.8 Realizáronse probas de interacción entre o usuario e a aplicación para mellorar as aplicacións desenvolvidas a partir de emuladores.
- CA2.9 Empaquetáronse e despregáronse as aplicacións desenvolvidas en dispositivos móbiles reais.
- CA2.10 Documentáronse os procesos necesarios para o desenvolvemento das aplicacións

BC1. Análise de tecnoloxías para desenvolvemento de aplicacións en dispositivos móbiles.

- Estrutura dunha aplicación para dispositivo móbil.
- Modificación de aplicacións existentes.
- Uso do contorno de execución do administrador de aplicacións.
- Contornos integrados de traballo.
- Módulos para o desenvolvemento de aplicacións móbiles.
- Emuladores.
- Perfís: características, arquitectura e requisitos. Dispositivos soportados.
- Ciclo de vida dunha aplicación: descubrimento, instalación, execución, actualización e borrado.
- Eventos da interface.
- Descubrimento de servizos
- Empaquetaxe e distribución.
- Documentación do desenvolvemento das aplicacións.

- Contexto gráfico. Imaxes.
- Comunicacións: clases asociadas. Tipos de conexións

Última actualización: 08.02.2024

SUBSECCIONES DE UD03. FRAGMENTOS Y NAVEGACIÓN

Capitulo 1

FRAGMENTOS

Un fragmento en Android es una porción **modular** y reutilizable de la interfaz de usuario de una aplicación. Puede considerarse como un "subcomponente" de una actividad, y su principal propósito es representar una parte específica de la interfaz de usuario o una funcionalidad de la aplicación dentro de una pantalla. Los fragmentos son utilizados principalmente para crear interfaces de usuario flexibles y adaptables en dispositivos con diferentes tamaños de pantalla, como teléfonos y tabletas.

Aquí hay algunas características clave de los fragmentos en Android:

1. **Reutilización:** Los fragmentos pueden ser reutilizados en diferentes partes de la aplicación y en múltiples actividades. Esto permite un desarrollo más modular y eficiente.
2. **Ciclo de vida propio:** Los fragmentos tienen su propio ciclo de vida, similar al ciclo de vida de una actividad. Esto significa que pueden responder a eventos y gestionar su propio estado.
3. **Interacción:** Los fragmentos pueden comunicarse entre sí y con la actividad que los contiene a través de la actividad anfitriona. Esto facilita la construcción de interfaces de usuario complejas con componentes interactivos.
4. **Diseño adaptable:** Los fragmentos se utilizan comúnmente en aplicaciones que necesitan adaptarse a diferentes tamaños de pantalla. Puedes cargar y organizar fragmentos de manera dinámica según el espacio disponible en la pantalla.
5. **Transacciones de fragmentos:** Los fragmentos se agregan, reemplazan o eliminan dinámicamente en tiempo de ejecución utilizando transacciones de fragmentos, lo que permite una gestión flexible de la interfaz de usuario.
6. **Compatibilidad con back stack:** Los fragmentos pueden ser gestionados en una pila de retroceso (back stack), lo que permite una navegación fluida hacia atrás y hacia adelante dentro de una actividad.

En resumen, un fragmento en Android es una herramienta poderosa que permite dividir y organizar la interfaz de usuario y la funcionalidad de una aplicación en componentes más pequeños y reutilizables, lo que facilita el desarrollo de aplicaciones flexibles y adaptables a una variedad de dispositivos y escenarios.

Autor/a: Sabela Sobrino Última actualización: 08.02.2024

SUBSECCIONES DE FRAGMENTOS

CREAR PROYECTO

Para ver cómo funcionan los ciclos de vida en Android vamos a generar un proyecto sobre el que veremos los distintos estados de forma práctica. Así, lo primero que haremos será generar nuestro protecyo en Android Studio siguiendo los siguientes pasos:

1. **Abrir Android Studio**: Abre Android Studio en tu ordenador.
2. **Crear un Nuevo Proyecto**:
    - Selecciona “Start a new Android Studio project” en la pantalla de inicio.
    - Elige “Phone and Tablet” como tipo de dispositivo.
    - Selecciona “Empty Activity” como plantilla para comenzar con una actividad vacía.
3. **Configuración del Proyecto**:
    - En la siguiente pantalla, completa la información básica sobre el proyecto:
        - **Name**: Ingresa “UD03_1_Fragmentos”.
        - **Package name**: Deja el nombre de paquete predeterminado o personalízalo según tus necesidades.
        - **Save location**: Elige la ubicación donde deseas guardar el proyecto en tu sistema.
        - **Language**: Selecciona “Kotlin” como lenguaje de programación.
        - **Minimum API level**: Selecciona “API 35: Android 7.0 (Nougat)” como SDK mínimo.
4. **Finalizar Configuración**:
    - Revisa la configuración y ajusta cualquier otra opción según tus preferencias.
    - Haz clic en “Finish” para crear el proyecto.

Android Studio generará automáticamente la estructura básica del proyecto, incluyendo los archivos necesarios para la actividad principal que has creado. Puedes comenzar a desarrollar tu aplicación agregando código a la actividad `MainActivity.kt` y diseñando la interfaz de usuario en el archivo de diseño correspondiente.

Autor/a: Sabela Sobrino Última actualización: 20.01.2025

RECURSOS

Crea los diferentes recursos para los distintos textos que aparecen en nuestra aplicación, tanto en inglés como en castellano:

Inglés

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
    <string name="app_name">UD1_UD03_SecretApp</string>
    <string name="welcome_text">Welcome to the Secret Message app! Use this app to encrypt a secret message.</string>
    <string name="start">Start</string>
```

```
    <string name="next">Next</string>
    <string name="encrypt_text">Here is your encrypted message!</string>
    <string name="message_text">Please, enter your message</string>
</resources>
```

Castellano

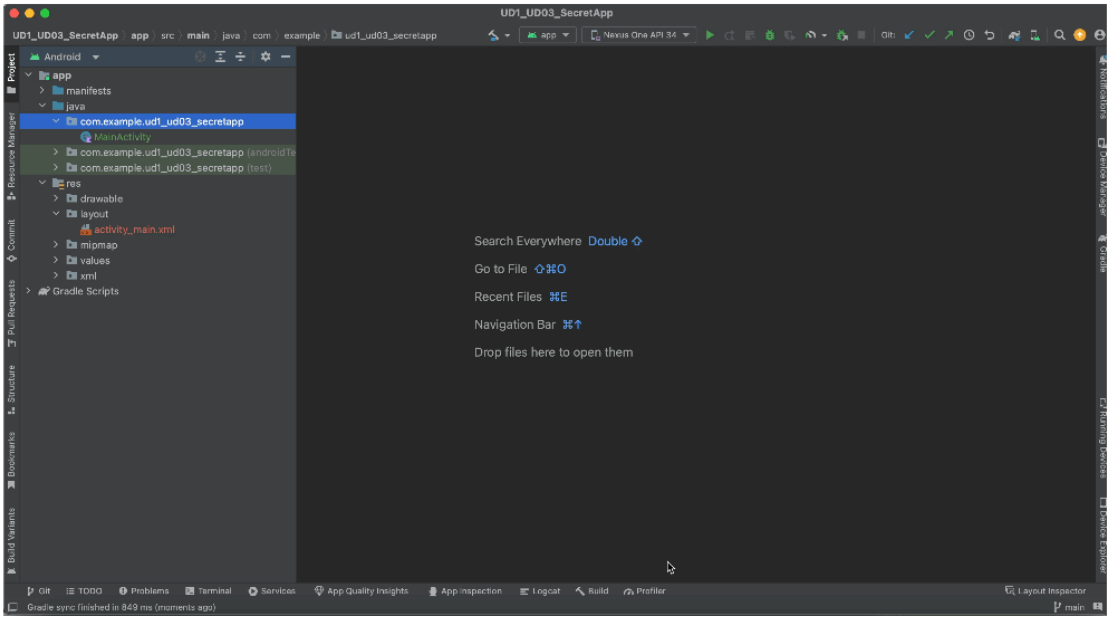
```
<resources>
    <string name="app_name">UD1_UD03_SecretApp</string>
    <string name="welcome_text">¡Bienvenido a la aplicación Secret Message! Usa esta app para encriptar un mensaje secreto.</string>
    <string name="start">Inicio</string>
    <string name="next">Siguiente</string>
    <string name="encrypt_text">¡Aquí está tu mensaje encriptado!</string>
    <string name="message_text">Por favor, introduce tu mensaje</string>
</resources>
```

Autor/a: Sabela Sobrino Última actualización: 08.02.2024

CREAR FRAGMENTO

Para crear un fragmento, se deben seguir estos pasos:

1. Ve al menú “File” y selecciona “Fragment”.
2. En esta sección, encontrarás varias opciones. Algunos fragmentos ya están predefinidos y tienen comportamientos específicos. En este caso, seleccionaremos la opción en **blanco**.
3. Aparecerá un asistente en el que deberás especificar el nombre del fragmento. En este ejemplo, utilizaremos “WelcomeFragment”. Automáticamente, se generará el nombre del archivo XML para el diseño del fragmento (siguiendo la convención de invertir el nombre y usar guiones bajos, similar a cómo se hace en los layouts).
4. Haz clic en “Finalizar” y verifica que se haya creado un archivo Kotlin y un archivo XML asociados a nuestro fragmento, de manera similar a lo que sucede al crear la actividad principal.



FUNCIÓN ONCREATEVIEW

Dirígete al archivo Kotlin correspondiente y elimina el código generado que, por el momento, no es necesario. Mantendremos únicamente lo esencial para este fragmento:

```
class WelcomeFragment : Fragment() {

    override fun onCreateView(
        inflater: LayoutInflater, container: ViewGroup?,
        savedInstanceState: Bundle?
    ): View? {
        // Inflamos el diseño de este fragmento
        return inflater.inflate(R.layout.fragment_welcome, container, false)
    }
}
```

Este código define la clase `WelcomeFragment`, que hereda de la clase `Fragment`, de manera similar a lo que hicimos en nuestro `MainActivity`. Al igual que teníamos un método `onCreate`, aquí también tenemos un método que se sobrescribe y se ejecuta en el momento en que se lanza este fragmento.

En este método, utilizamos el objeto `inflater` para inflar (crear) la interfaz de usuario asociada a este fragmento a partir del archivo XML que se generó simultáneamente con este archivo. Este método devuelve una **vista** que representa la interfaz de usuario del fragmento.

Los fragmentos son componentes adicionales en nuestra interfaz de usuario, y este código se encarga de inflar la interfaz de usuario definida en el archivo XML asociado al fragmento.

Autor/a: Sabela Sobrino Última actualización: 08.02.2024

FRAGMENT WELCOME

Vamos a diseñar el primer fragmento que constituirá la pantalla inicial. Queremos lograr la siguiente apariencia:

Para construir esta pantalla, podemos utilizar un `LinearLayout` con las siguientes especificaciones:

Fragmento: Welcome

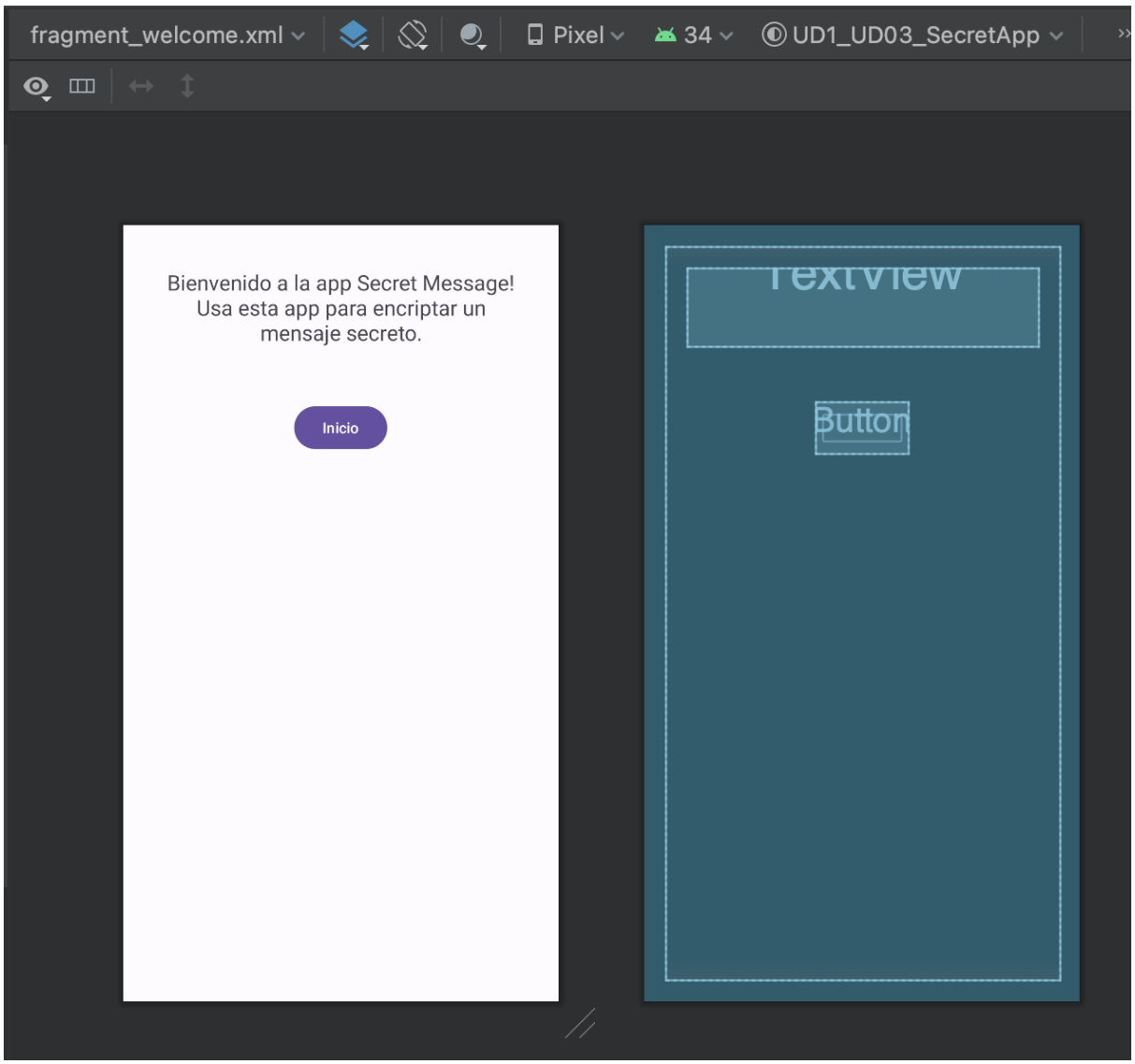
- **Diseño:**
  - Centrado horizontalmente.
- **TextView:**
  - Centrado horizontalmente.
  - Margen de 20dp.
  - Tamaño de texto de 20sp.
- **Botón:**
  - Margen superior de 32dp.

Para lograr esto, abrimos el archivo `fragment_welcome.xml` y creamos un `LinearLayout` de la siguiente manera:

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:layout_margin="20dp"
    android:orientation="vertical"
    android:gravity="center_horizontal" >

    <TextView
```

```
        android:id="@+id/welcome_text"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:gravity="center_horizontal"
        android:textSize="20sp"
        android:layout_margin="20dp"
        android:text="@string/welcome_text" />

    <Button
        android:id="@+id/button_start"
        android:layout_width="wrap_content"
        android:layout_marginTop="32dp"
        android:layout_height="wrap_content"
        android:text="@string/start" />

</LinearLayout>
```

El resultado final se verá así:



FRAGMENT CONTAINER VIEW

El fragmento que acabamos de crear se mostrará dentro de la actividad principal, que, a su vez, cuenta con su propio diseño (layout). En el archivo activity_mail.xml, donde se define el diseño de la actividad principal, crearemos un contenedor de fragmentos llamado `FragmentContainerLayout`. Este contenedor nos permite cargar fragmentos dinámicamente.

Para agregar esta funcionalidad, debemos modificar el archivo build.gradle y añadir la siguiente línea de dependencia:

```
implementation 'androidx.fragment:fragment-ktx:1.6.1'
```

En nuestro caso, dado que solo tendremos un fragmento que ocupará toda la pantalla, podemos utilizar directamente un `FragmentContainerView`. Sin embargo, debemos tener en cuenta que no podemos usarlo como elemento raíz de nuestro diseño. Para configurar los elementos básicos del contenedor y asignarle un identificador para futuras referencias, podemos definirlo de la siguiente manera en el archivo XML:

```
<?xml version="1.0" encoding="utf-8"?>
<androidx.fragment.app.FragmentContainerView
    xmlns:tools="http://schemas.android.com/tools"
    xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:padding="16dp"
    android:id="@+id/fragment_container_view">
```

A través de la propiedad “name”, podemos especificar qué fragmento se cargará en este contenedor. De esta manera, al modificar esta propiedad, podemos cambiar dinámicamente el fragmento que se muestra en función de las acciones realizadas en nuestra aplicación:

```
android:name="com.example.ud1_ud03_secretapp.WelcomeFragment"
```

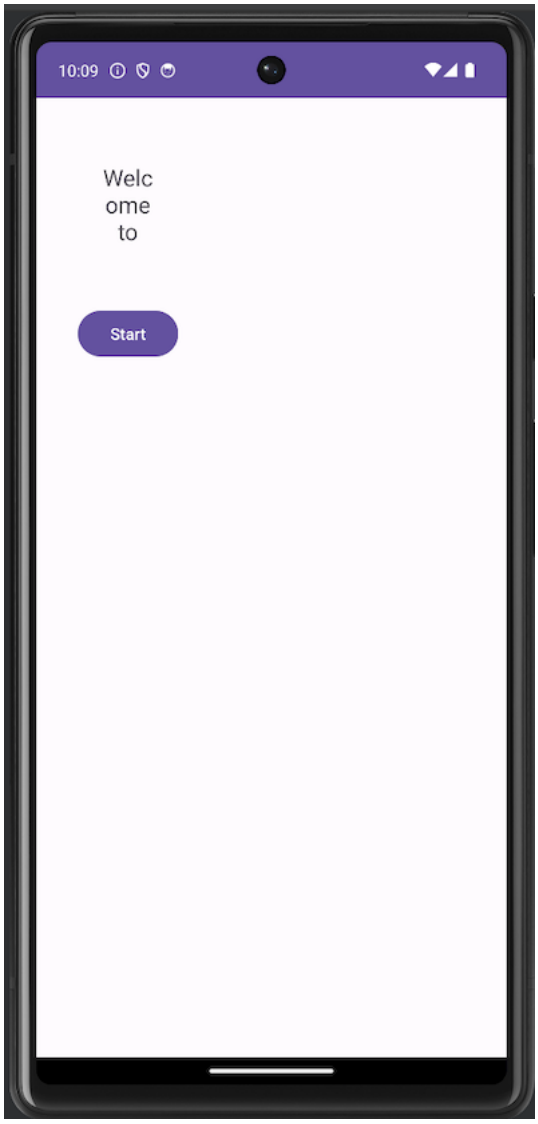
Autor/a: Sabela Sobrino Última actualización: 08.02.2024

FLUJO DE EJECUCIÓN

Actualmente, contamos con una actividad principal en nuestro archivo `MainActivity.kt`. Cuando esta actividad se crea, su principal tarea es construir la interfaz de usuario definida en el archivo XML asociado a `activity_main.xml`.

En este momento, nuestro diseño se basa en un **“contenedor de fragmentos”**. Inicialmente, este contenedor de fragmentos carga un fragmento específico. Este fragmento opera de manera similar a una actividad, en el sentido de que tiene su propio archivo Kotlin con código que se ejecuta al lanzarse. Además, el fragmento crea su propia interfaz de usuario a partir de un archivo XML, siguiendo el mismo proceso que una actividad.

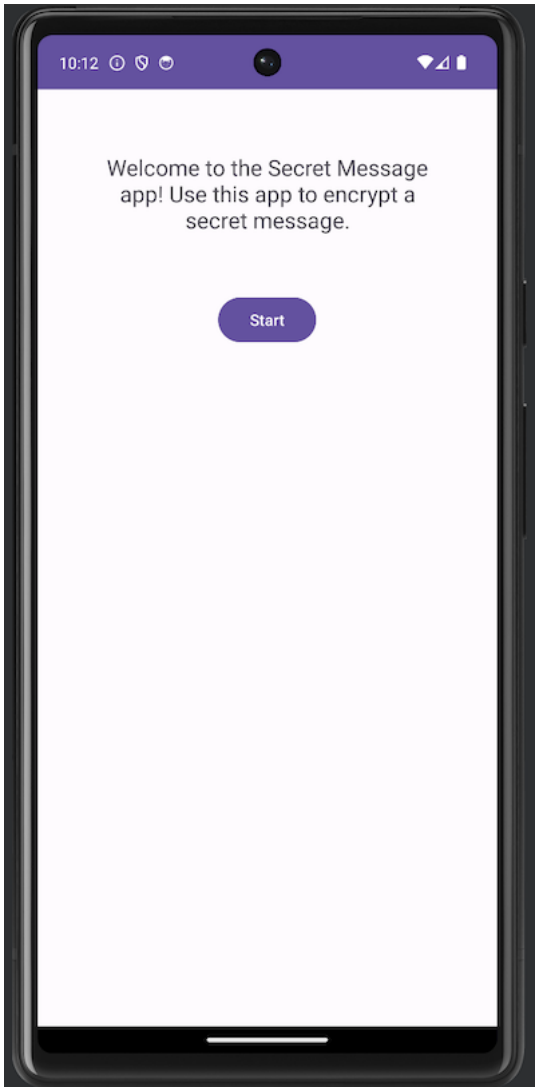
Sin embargo, al ejecutar la aplicación, nos encontramos con un problema visual, que se muestra en la siguiente imagen:



Este problema se debe a que el ancho y el alto del contenedor están configurados para ajustarse automáticamente al contenido (en este caso, el fragmento). Para resolverlo, debemos ajustar el ancho y el alto del contenedor para que coincidan con las dimensiones de la pantalla:

```
android:layout_width="match_parent"
android:layout_height="match_parent"
```

Al ejecutar la aplicación nuevamente, podemos comprobar que la interfaz se muestra correctamente:



Idioma
Comprobar la configuración del idioma. Cambiar idioma del móvil al inglés para comprobar que funciona.
Autor/a: Sabela Sobrino Última actualización: 08.02.2024

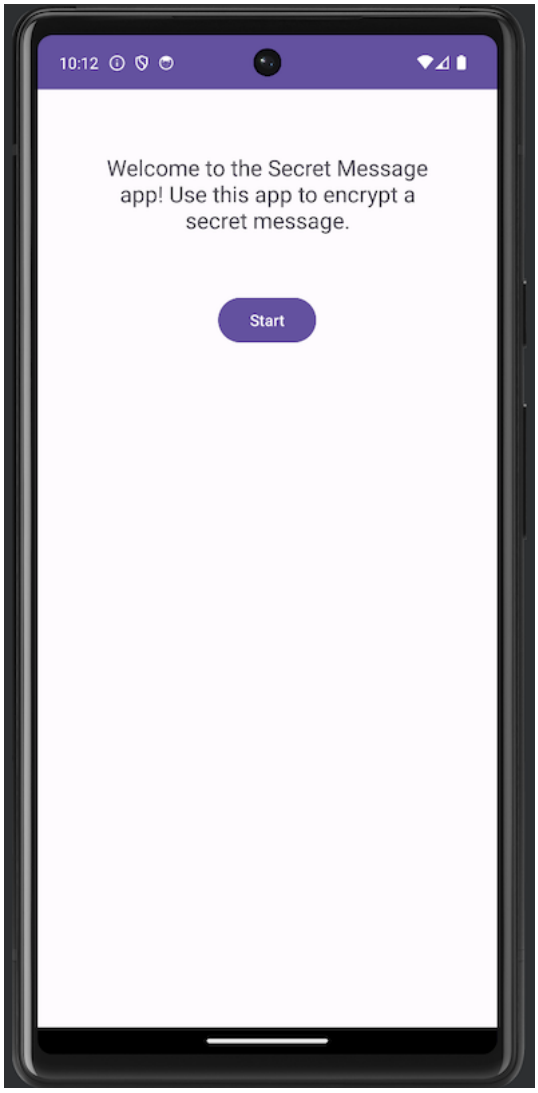
MÚLTIPLES FRAGMENTOS

Es posible incluir **más de un fragmento** en una misma pantalla. Para esto, vamos a crear un LinearLayout que contendrá dos contenedores de fragmentos:

```
<?xml version="1.0" encoding="utf-8"?>

<LinearLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:orientation="vertical"
    >

    <androidx.fragment.app.FragmentContainerView
```

```
        xmlns:tools="http://schemas.android.com/tools"
        xmlns:android="http://schemas.android.com/apk/res/android"
        android:layout_width="match_parent"
        android:layout_height="match_parent"
        android:padding="16dp"
        android:id="@+id/fragment_container_view"
        android:name="com.example.ud1_ud03_secretapp.WelcomeFragment">

    </androidx.fragment.app.FragmentContainerView>
    <androidx.fragment.app.FragmentContainerView
        xmlns:tools="http://schemas.android.com/tools"
        xmlns:android="http://schemas.android.com/apk/res/android"
        android:layout_width="match_parent"
        android:layout_height="match_parent"
        android:padding="16dp"
        android:id="@+id/fragment_container_view_2"
        android:name="com.example.ud1_ud03_secretapp.WelcomeFragment">

    </androidx.fragment.app.FragmentContainerView>

</LinearLayout>
```

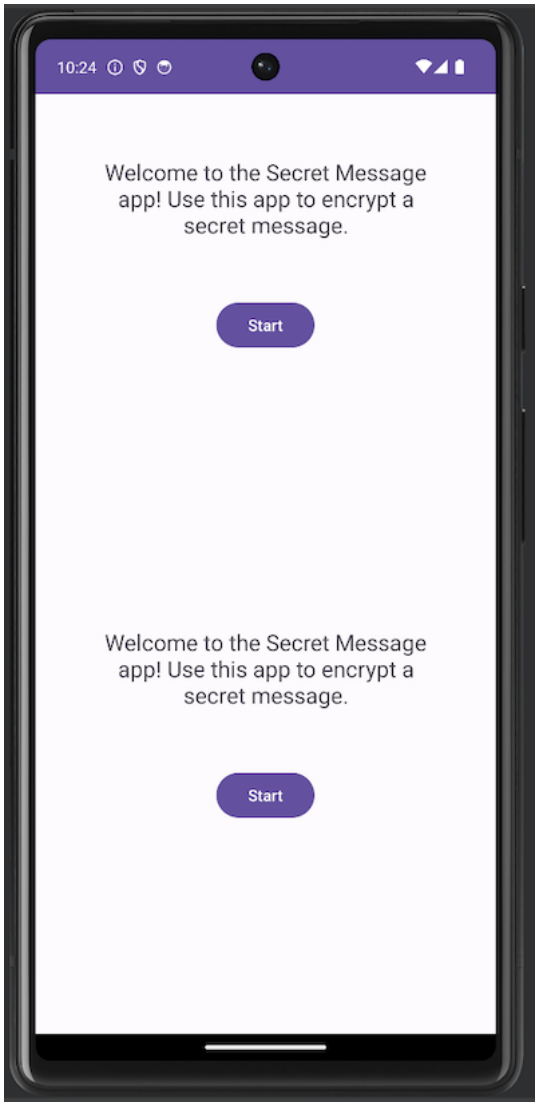
Al ejecutar este diseño, observamos que solo se muestra un fragmento:



La razón de esto es que las propiedades de ancho (layout_width) y alto (layout_height) no están configuradas correctamente, ya que ambos fragmentos están tratando de ocupar todo el espacio del contenedor padre, que es el mismo para ambos. Para solucionarlo, simplemente debemos añadir la propiedad “layout_weight” para que el espacio se divida de manera equitativa entre ambos fragmentos, asignándoles un 50% a cada uno:

```
<androidx.fragment.app.FragmentContainerView>

    android:layout_weight="1"

</androidx.fragment.app.FragmentContainerView>
```

Con esta configuración, ahora mostramos ambos fragmentos en la misma pantalla:



Es importante entender que cada uno de estos fragmentos se comporta como una **actividad independiente**, con su propio ciclo de vida y gestión de eventos, similar a tener múltiples pantallas que pueden funcionar de manera independiente en la misma interfaz.

Alerta
Revertir los cambios de este apartado para tener un único contenedor.
Autor/a: Sabela Sobrino Última actualización: 08.02.2024

FRAGMENT MESSAGE

Crea un nuevo fragmento con la apariencia y características deseadas, sigue estas especificaciones:

Layout:

- Debe estar centrado horizontalmente en la pantalla.

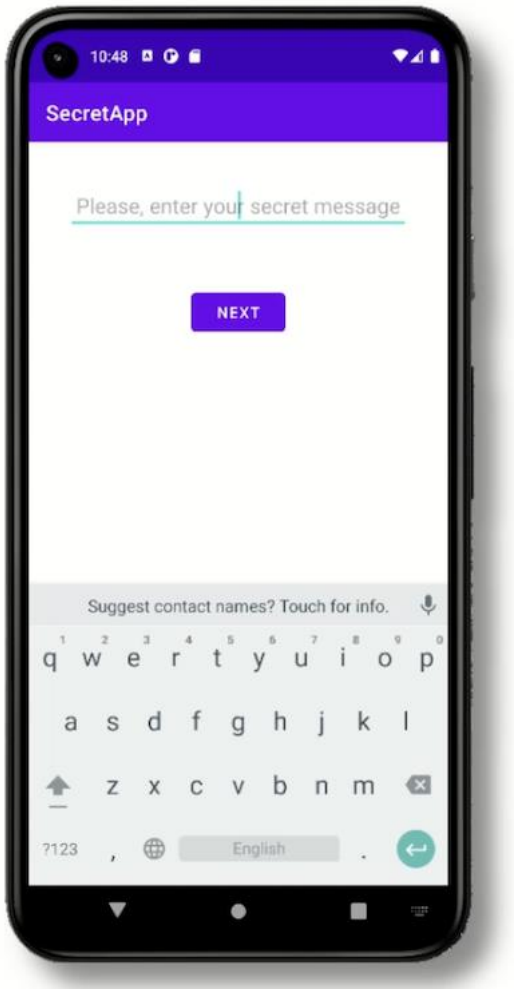
EditText:

- Debe ser de varias líneas.
- Debe tener un margen de 20dp.
- El tamaño del texto debe ser de 20sp.
- Debe mostrar un hint.

Button:

- Debe tener un margen superior (top) de 32dp.

El resultado final debe coincidir con la siguiente apariencia:



Solución

De forma similar que en el caso anterior, creamos de nuevo un fragmento con el nombre MessageFragment. En este fragmento añadimos una caja de texto que nos permita escribir varias líneas y con un botón que nos indique “Siguiente”, como el diseño es muy sencillo podemos utilizar un LinearLayout:

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:layout_margin="20dp"
    android:orientation="vertical"
```

```
    android:gravity="center_horizontal" >

    <EditText
        android:id="@+id/edit_text"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:textSize="20sp"
        android:layout_margin="20dp"
        android:hint="@string/mensage_text"
        android:inputType="textMultiLine" />

    <Button
        android:id="@+id/button_next"
        android:layout_width="wrap_content"
        android:layout_marginTop="32dp"
        android:layout_height="wrap_content"
        android:text="@string/next" />

</LinearLayout>
```

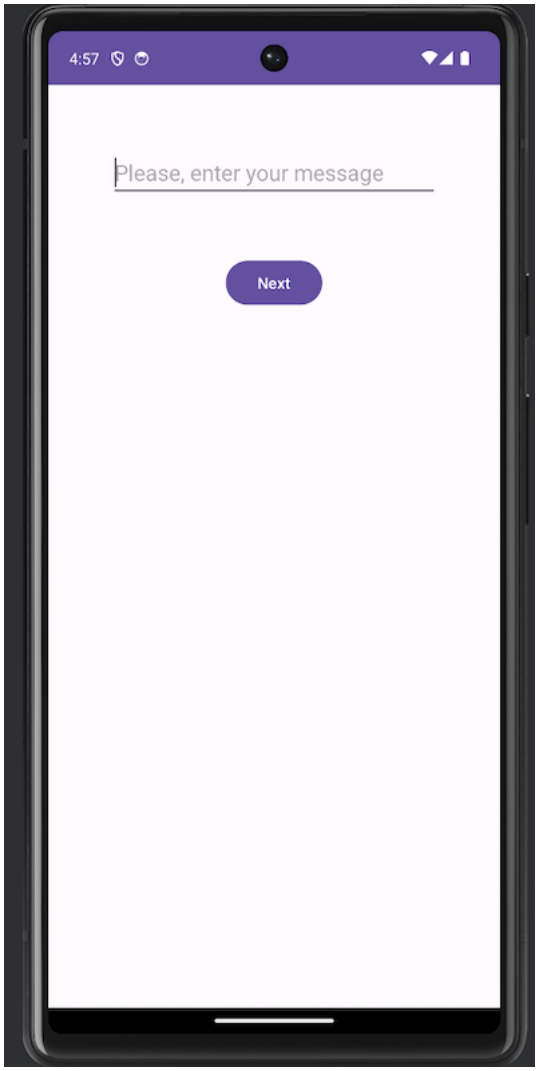
Ahora vamos a editar el archivo Kotlin asociado al fragmento y nos quedamos únicamente con el método `onCreateView`.

```
class MessageFragment : Fragment() {

    override fun onCreateView(
        inflater: LayoutInflater, container: ViewGroup?,
        savedInstanceState: Bundle?
    ): View? {
        // Inflar el diseño (layout) para este fragmento
        return inflater.inflate(R.layout.fragment_message, container, false)
    }
}
```

Para verificar que este fragmento funciona correctamente, vamos a realizar cambios en el archivo activity_main.xml. Reemplazaremos el fragmento anterior con el nuevo fragmento que hemos creado:

```
<androidx.fragment.app.FragmentContainerView
    xmlns:tools="http://schemas.android.com/tools"
    xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:padding="16dp"
    android:layout_weight="1"
    android:id="@+id/fragment_container_view"
    android:name="com.example.ud1_ud03_secretapp.MessageFragment">
</androidx.fragment.app.FragmentContainerView>
```

Esto generará la siguiente apariencia en la pantalla:



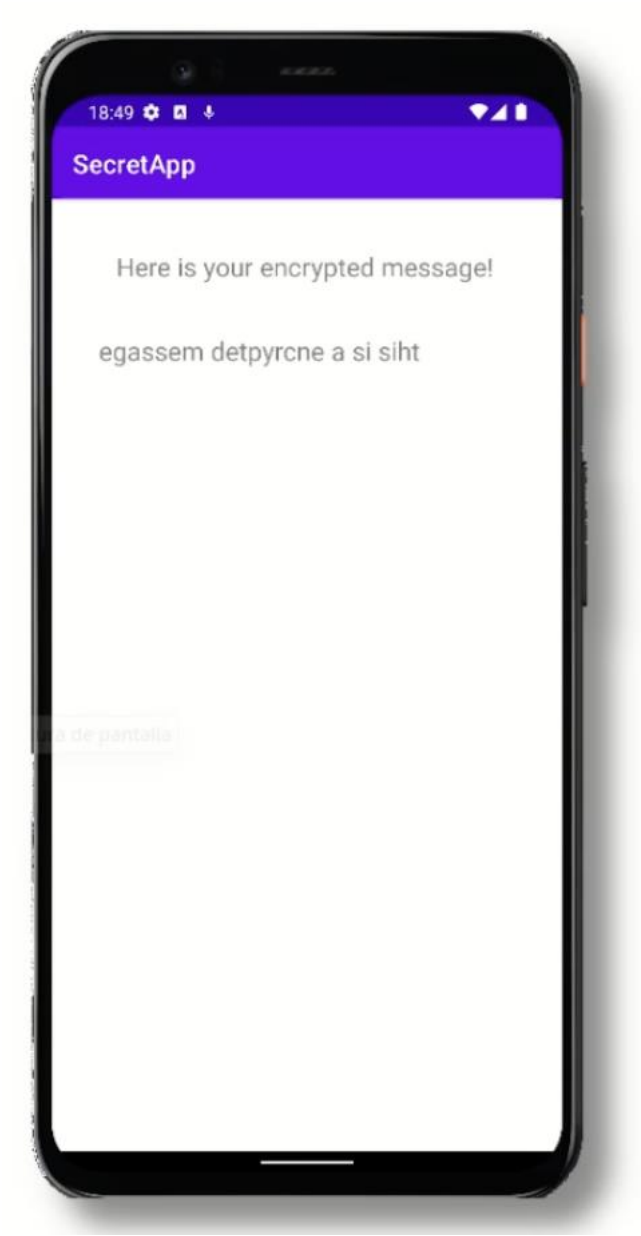
Alerta

Para volver a cargar el fragmento “WelcomeFragment”, debes revertir los cambios realizados en esta sección del código en el archivo.

Autor/a: Sabela Sobrino Última actualización: 08.02.2024

FRAGMENT ENCRYPT

Vamos a diseñar el tercer fragmento, que constituirá la pantalla donde se muestra el mensaje cifrado. En este caso, no tendremos un botón, ya que haremos uso del botón de Android para volver atrás y permitir que el usuario regrese a la pantalla inicial. Queremos lograr la siguiente apariencia:



Para construir esta pantalla, utilizaremos un `LinearLayout` con las siguientes especificaciones:

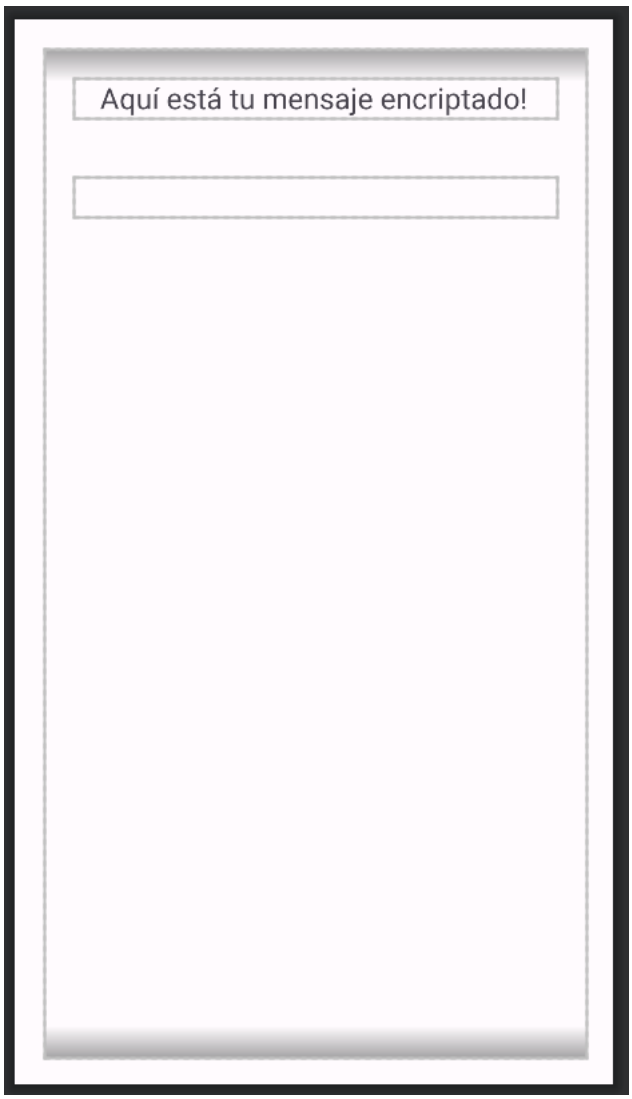
- Centrado horizontal.
- `TextView` con un margen de 20dp, centrado horizontal y un tamaño de texto de 20sp.
- Otro `TextView` con un margen de 20dp, un tamaño de texto de 20sp y capacidad de mostrar múltiples líneas.

Para lograr esto, primero debemos crear un nuevo fragmento en blanco al que llamaremos “EncryptFragment”. Luego, abrimos el archivo `fragment_encrypt.xml` y creamos un `LinearLayout` con las siguientes propiedades:

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:layout_margin="20dp"
    android:orientation="vertical"
```

```
    android:gravity="center_horizontal">

    <TextView
        android:id="@+id/encrypt_text"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:gravity="center_horizontal"
        android:textSize="20sp"
        android:layout_margin="20dp"
        android:text="@string/encrypt_text" />

    <TextView
        android:id="@+id/encrypt_textView"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:gravity="center_horizontal"
        android:textSize="20sp"
        android:layout_margin="20dp" />
</LinearLayout>
```

El resultado final se verá de la siguiente manera:



Luego, debemos modificar la lógica del fragmento y quedarnos únicamente con el método `onCreateView`:

```
class EncryptFragment : Fragment() {
    override fun onCreateView(
        inflater: LayoutInflater, container: ViewGroup?,
        savedInstanceState: Bundle?
    ): View? {
        // Inflamos el diseño de este fragmento
        return inflater.inflate(R.layout.fragment_encrypt, container, false)
    }
}
```

Autor/a: Sabela Sobrino Última actualización: 08.02.2024

Capitulo 2

NAVEGACIÓN

Para la navegación entre fragmentos, utilizaremos el componente **Navigation** de Jetpack, conocido como el controlador de navegación, el cual administra qué fragmento se muestra en el host de navegación a medida que el usuario se desplaza por la aplicación.

La navegación entre fragmentos implica tres elementos clave:

- **Grafo de navegación**: Este grafo describe las posibles “rutas” que el usuario puede seguir al navegar por la aplicación. Aunque se define como un recurso XML, por lo general se edita en el editor de diseño visual.
- **Host de navegación**: Este componente actúa como el contenedor en el que se presenta el fragmento al que estamos navegando. Se encuentra dentro del diseño de la actividad.
- **Controlador de navegación**: El controlador de navegación se encarga de determinar qué fragmento se mostrará en el host de navegación a medida que el usuario interactúa con la aplicación. Este controlador se manipula desde el código de la aplicación.

Referencias

- Inicio con Navigation
- Navigation

Autor/a: Sabela Sobrino Última actualización: 20.01.2025

SUBSECCIONES DE NAVEGACIÓN

DEPENDENCIA

Dado que este componente es una adición adicional, necesitamos incorporarlo a la configuración de nuestra aplicación. Para hacerlo, vamos a realizar modificaciones en el archivo de configuración build.gradle de nuestra aplicación y añadir la nueva dependencia.

Aquí hay dos archivos disponibles, uno relacionado con la aplicación y otro con el proyecto en general. Elegiremos el archivo asociado a la aplicación, como se muestra a continuación:

Luego, incluiremos la siguiente línea en la sección de dependencias:

```
dependencies {
    // ... Otras dependencias ...
    // Kotlin
    implementation("androidx.navigation:navigation-fragment-ktx:2.7.7")
    // ... Otras dependencias ...
}
```

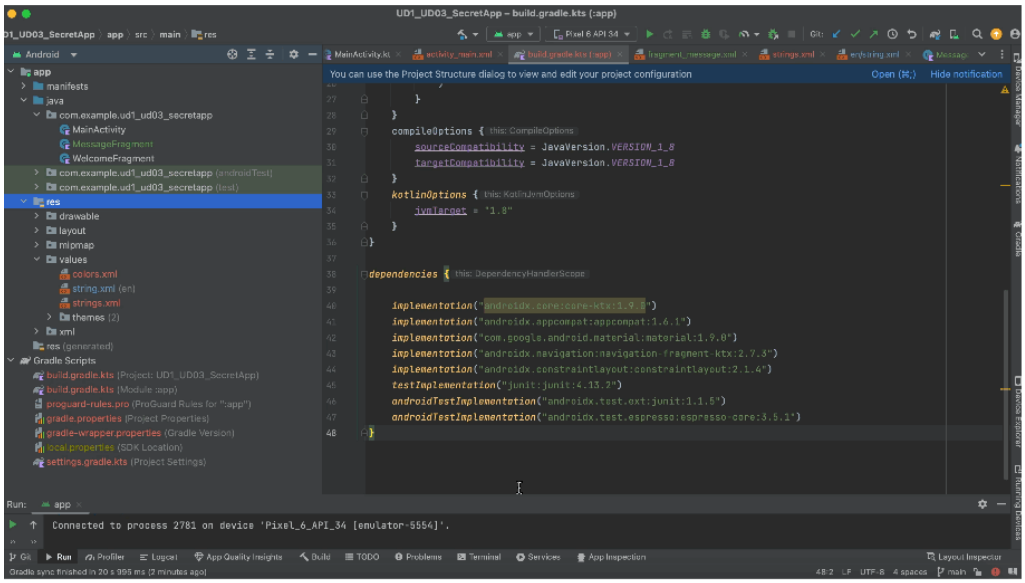
Una vez que hayamos agregado esta línea, deberemos hacer clic en el botón “Sync Now” para actualizar las dependencias. De esta forma, tendremos las siguientes dependencias:

```
dependencies {
    implementation("androidx.core:core-ktx:1.9.0")
    implementation("androidx.appcompat:appcompat:1.6.1")
    implementation("com.google.android.material:material:1.9.0")
    implementation("androidx.navigation:navigation-fragment-ktx:2.7.7")
    implementation("androidx.constraintlayout:constraintlayout:2.1.4")
    testImplementation("junit:junit:4.13.2")
    androidTestImplementation("androidx.test.ext:junit:1.1.5")
    androidTestImplementation("androidx.test.espresso:espresso-core:3.5.1")
```

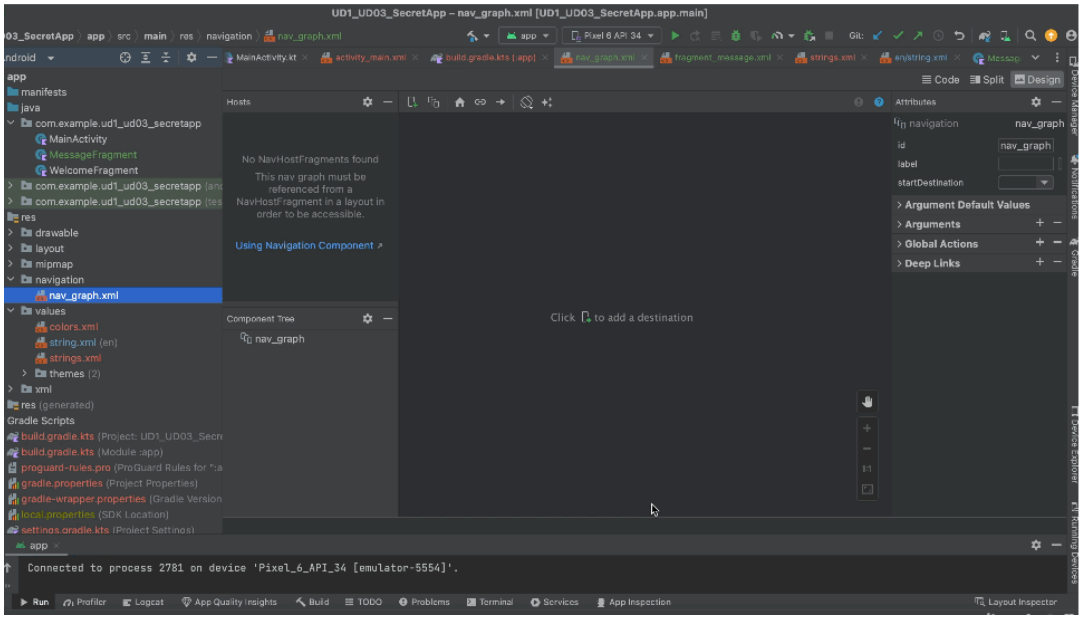
Autor/a: Sabela Sobrino Última actualización: 20.01.2025

GRAFO DE NAVEGACIÓN

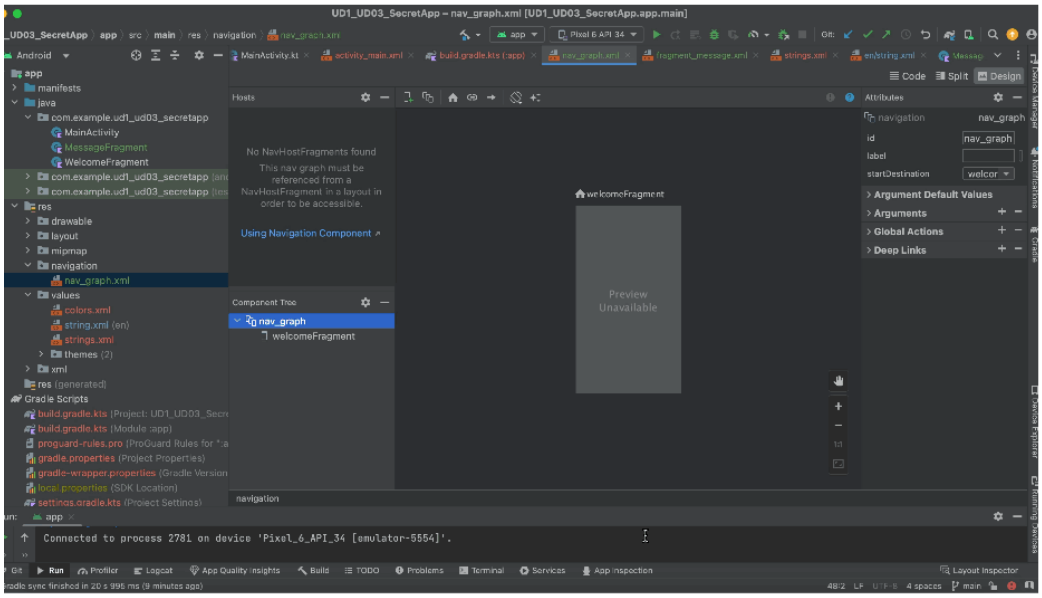
Como hemos visto, el grafo de navegación es un recurso XML que podemos agregar de manera sencilla. Para hacerlo, simplemente creamos un nuevo “Archivo de Recurso de Android” desde la carpeta de Recursos y seleccionamos el tipo “Navigation”.



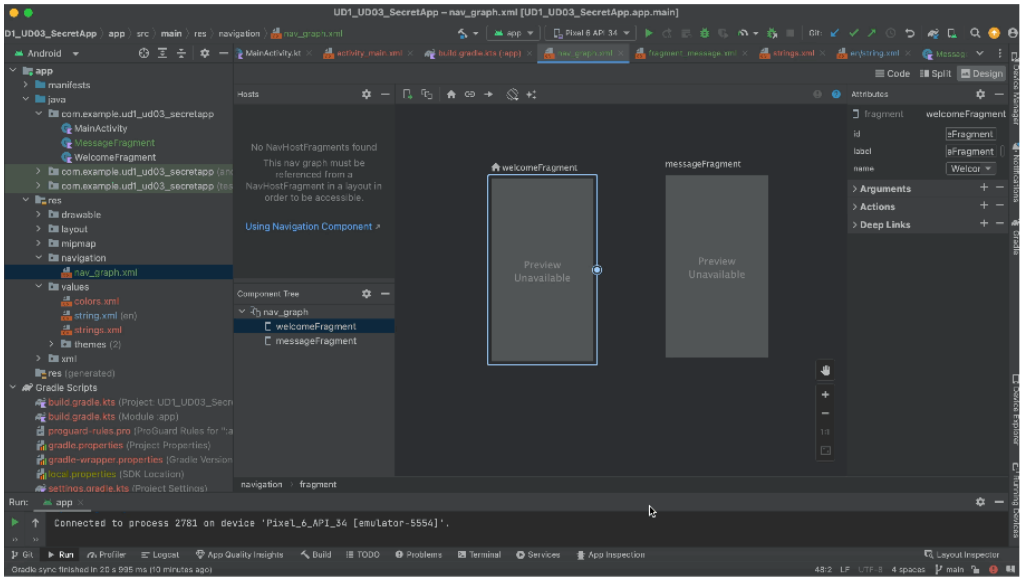
Una vez creado, se abrirá automáticamente un editor visual que nos permitirá añadir nuestro grafo. Al hacer clic en el botón para agregar un nuevo destino, veremos una lista de los diferentes fragmentos que hemos creado.



Al seleccionar el botón para agregar otro fragmento, veremos que se añaden ambos fragmentos al editor visual.



Si hacemos clic en cualquiera de estos fragmentos, notaremos que se muestra un círculo que nos permite definir acciones, es decir, rutas que nos llevarán de un fragmento a otro.



Al seleccionar una de esas flechas, podemos ver sus atributos y confirmar que se trata de una acción que tiene un ID por defecto que utilizaremos más adelante.

Como mencionamos previamente, estos grafos son archivos XML, y podemos ver su código en la pestaña “Code”:

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

```
    xmlns:app="http://schemas.android.com/apk/res-auto"
    android:id="@+id/nav_graph"
    app:startDestination="@id/welcomeFragment">

    <fragment
        android:id="@+id/welcomeFragment"
        android:name="com.example.ud1_ud03_secretapp.WelcomeFragment"
        android:label="WelcomeFragment">
        <action
            android:id="@+id/action_welcomeFragment_to_messageFragment"
            app:destination="@id/messageFragment" />
    </fragment>
    <fragment
        android:id="@+id/messageFragment"
        android:name="com.example.ud1_ud03_secretapp.MessageFragment"
        android:label="MessageFragment" />
</navigation>
```

En el código anterior, es relevante destacar la propiedad `app:startDestination="@id/welcomeFragment"`, que indica cuál es el primer fragmento que se carga. También podemos observar las definiciones de ambos fragmentos, así como la acción que conecta el primer fragmento con el segundo.

Autor/a: Sabela Sobrino Última actualización: 08.02.2024

NAVHOSTFRAGMENT

Vamos a aprender cómo agregar el ***container*** que contendrá nuestro sistema de navegación. Si nos dirigimos al diseño de la actividad principal, encontramos un contenedor de fragmentos, donde ahora hemos añadido el fragmento “WelcomeFragment”.

Lo que necesitamos hacer es insertar un *“host de navegación”* en ese contenedor, es decir, un contenedor para nuestro sistema de navegación que, a su vez, contendrá los distintos fragmentos. Afortunadamente, la biblioteca de navegación de Android nos proporciona un componente de este tipo llamado `NavHostFragment`, que es un fragmento en sí mismo y, además, implementa toda la lógica relacionada con la navegación. Por lo tanto, podemos incluir directamente este fragmento de navegación en nuestro contenedor.

Para lograrlo, simplemente debemos realizar algunos cambios:

1. Primero, vamos a reemplazar el **fragmento** que tenemos actualmente:

   ```
   android:name="com.example.ud1_ud03_secretapp.MessageFragment"
   ```

   con este componente:

   ```
   android:name="androidx.navigation.fragment.NavHostFragment"
   ```

   `NavHostFragment` será nuestro “host de navegación” que contendrá los fragmentos por los que navegaremos. Como se puede observar en la documentación oficial de este componente, `NavHostFragment` hereda de `Fragment`.

```
open class NavHostFragment : Fragment, NavHost
```

por eso lo podemos meter dentro de nuestro contenedor. Además, impmeneta una interface (NavHost) donde está la lógica relativa que nos permite desplazarnos entre fragmentos.

2. Debemos proporcionar el **grafo de navegación**. Para hacerlo, agregaremos un nuevo espacio de nombres (`namespace`), específico para la biblioteca de navegación:

```
xmlns:app="http://schemas.android.com/apk/res-auto"
```

Aprovechando este espacio de nombres, incluiremos dos nuevas propiedades:

```
app:navGraph="@navigation/nav_graph"
app:defaultNavHost="true"
```

La primera propiedad hace referencia al grafo de navegación que creamos. La segunda propiedad, aunque no la utilizaremos por el momento, está relacionada con el botón “Atrás” en las aplicaciones; con esta opción podemos controlar la navegación a través de él.

Antes de continuar, verifiquemos que, tras los cambios realizados, la aplicación se carga correctamente y muestra el primer fragmento.

Autor/a: Sabela Sobrino Última actualización: 08.02.2024

NAVEGACIÓN

El siguiente paso es habilitar la transición a la otra pantalla al presionar el botón “Start”. Para lograrlo, vamos a realizar cambios en el código del `WelcomeFragment` para obtener una referencia al botón y agregar un listener que defina el comportamiento de navegación hacia el otro fragmento.

Notamos que en este caso, el fragmento no hereda de la clase `Activity`, por lo que no podemos usar el método `findViewByID` directamente. En su lugar, obtendremos una referencia al botón de otra manera. Como este método `onCreateView` crea una **vista**, utilizaremos esa vista para encontrar el botón utilizando `findViewByID`. En lugar de devolver la vista de inmediato, la almacenaremos en una variable:

```
val view = inflater.inflate(R.layout.fragment_welcome, container, false)
```

Ahora, dentro de esta variable `view`, tenemos acceso al método `findViewByID`, lo que nos permite obtener una referencia al botón “Start”:

```
val buttonStart = view.findViewById<Button>(R.id.button_start)
```

Luego, podemos agregar un listener al botón y activar la acción definida en nuestro grafo de navegación. En el grafo, tenemos una acción que se ve así:

```
<action
    android:id="@+id/action_welcomeFragment_to_messageFragment"
    app:destination="@id/messageFragment" />
```

Para lograr esta transición, utilizaremos el **controlador de navegación**, que es el tercer elemento que habíamos definido al principio:

- Grafo de navegación: representa el mapa de navegación.
- Host de navegación: integra el sistema de navegación.
- Controlador de navegación: a través de este, podemos activar las acciones.

Para obtener una referencia al controlador de navegación, lo hacemos a través de la vista utilizando el método `findNavController()`. Dentro de este controlador, tenemos un método llamado `navigate`, al cual simplemente le pasamos la acción que deseamos que se ejecute:

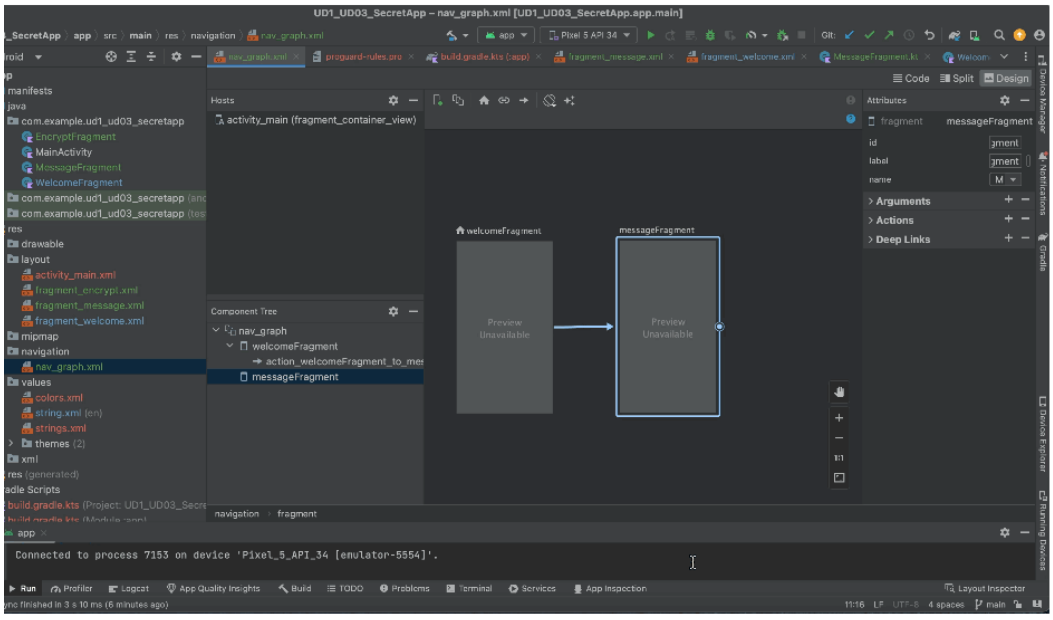
```
buttonStart.setOnClickListener {
    view.findNavController().navigate(R.id.action_welcomeFragment_to_messageFragment)
}
```

Si ejecutamos nuestra aplicación, verificamos que al presionar el botón “Start”, cambiamos de pantalla.

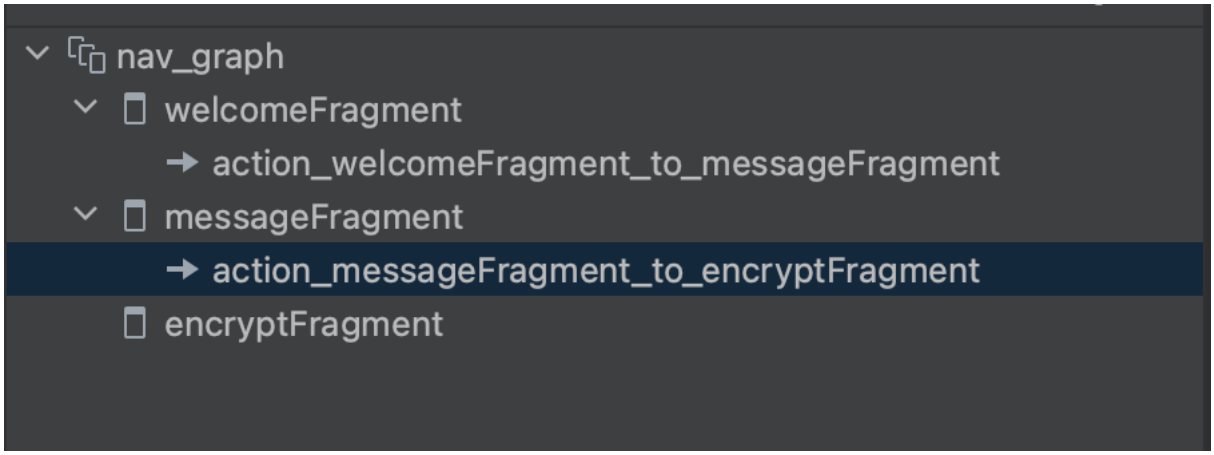
Autor/a: Sabela Sobrino Última actualización: 08.02.2024

GRADO DE NAVEGACIÓN COMPLETO

Vams a vovler al grafo de navegación para completarlo. En este momento tenemos dos fragmentos: `WelcomeFragment` y `MessageFragment`. Ahora, vamos a añadir nuestro tercer fragmento y creamos una acción que va desde la segunda pantalla a este fragmento:



Recuerda que esta acción tiene un identificador que nos permitirá hacer referencia a ella:

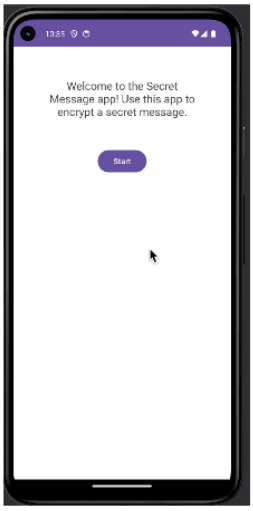


Con estos pasos, habrás añadido el tercer fragmento a tu aplicación y establecido la navegación desde la segunda pantalla a la pantalla de cifrado.

A continuación, deberemos agregar el **listener** asociado al botón para poder cambiar de pantalla. Para hacerlo, nos dirigimos al Fragmento de Mensajes (MessageFragment) y, de manera similar a lo que hicimos anteriormente, creamos una referencia al botón y definimos la acción a ejecutar:

```
val view = inflater.inflate(R.layout.fragment_message, container, false)
val buttonNext = view.findViewById<Button>(R.id.button_next)
buttonNext.setOnClickListener {
    view.findNavController().navigate(R.id.action_messageFragment_to_encryptFragment)
}
return view
```

Si ejecutamos nuestra aplicación en este punto, veremos que la navegación de la aplicación funciona de manera completa.



Autor/a: Sabela Sobrino Última actualización: 08.02.2024

Capitulo 3

ARGUMENTOS

Para intercambiar información entre diferentes fragmentos, utilizaremos un plugin de Gradle llamado Safe Args, el cual se integra con el componente de navegación. Este componente asegura la integridad de los tipos, por lo tanto, se recomienda su uso de manera general. El funcionamiento de este componente es bastante simple.

En primer lugar, debemos crear los **argumentos** en el **fragmento** que recibirá la información. Este componente generará dos clases en todos los fragmentos que tengan acciones definidas, es decir, fragmentos que se dirijan hacia cualquier otro destino dentro de nuestra aplicación.

Se generará una clase llamada "**Directions**". El nombre por defecto de esta clase será el nombre del propio fragmento, seguido de "Directions". En nuestro ejemplo, con el fragmento "Message", se generará "MessageDirections".

En cuanto a la recepción de datos en los fragmentos que tienen argumentos definidos y, por lo tanto, recibirán datos, se creará una clase llamada **"Args"** siguiendo la convención mencionada anteriormente, es decir, el nombre del fragmento seguido de "Args". Esta clase tendrá un método llamado **"fromBundle()"** que básicamente nos permitirá definir un Bundle en el cual se almacenarán los datos que se pasen como argumento al fragmento. Podremos recuperar estos datos desde ese Bundle.

Este mecanismo es bastante **sencillo** y, gracias a la funcionalidad de Safe Args, se construirán automáticamente estas clases a través de las cuales podemos enviar la información y la clase desde la cual la recibiremos. Veremos esto en detalle en nuestro proyecto.

Autor/a: Sabela Sobrino Última actualización: 20.01.2025

SUBSECCIONES DE ARGUMENTOS

DEPENDENCIA

Para utilizar este componente, debemos incorporar un plugin en nuestro proyecto, lo que implica realizar ajustes en dos archivos Gradle, tanto en el nivel del **proyecto** como en la **aplicación**.

En el archivo de configuración de nivel superior (el del proyecto), donde podemos agregar opciones de configuración comunes para todos los subproyectos/módulos, se deben incluir las siguientes líneas:

```
plugins {
    alias(libs.plugins.android.application) apply false
    alias(libs.plugins.jetbrains.kotlin.android) apply false
    id("androidx.navigation.safeargs") version "2.5.3" apply false
}
```

Una vez que hayamos realizado estas modificaciones en el archivo, es necesario sincronizar las dependencias haciendo clic en el botón **“Sync Gradle”** o **“Sincronizar dependencias”,** según la interfaz de desarrollo que estemos utilizando.

Además, en el archivo de configuración Gradle de la **aplicación**, debemos agregar el plugin de la siguiente manera:

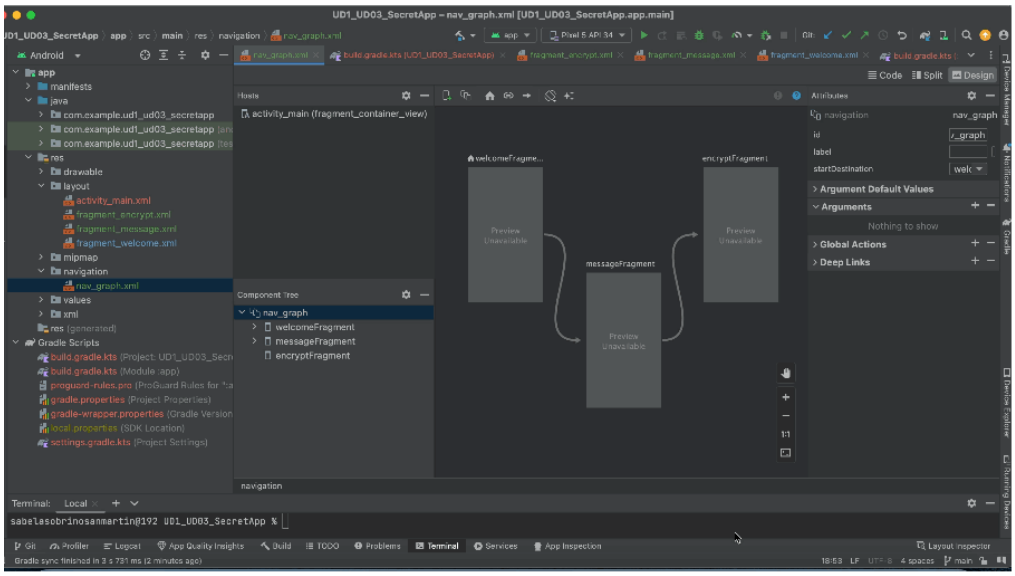
```
plugins {
    alias(libs.plugins.android.application)
    alias(libs.plugins.jetbrains.kotlin.android)
    id("androidx.navigation.safeargs")
}
```

Una vez hecho esto, habremos añadido el plugin de Gradle a nuestro proyecto. Nuevamente, será necesario **sincronizar** las dependencias haciendo clic en el botón “Sync Gradle” para asegurarnos de que todas las configuraciones se apliquen correctamente.

Autor/a: Sabela Sobrino Última actualización: 20.01.2025

DEFINIR ARGUMENTOS

Vamos a dirigirnos a nuestro grafo y, en el fragmento final, llamado **“EncryptFragment”**, que será el **destinatario final** de nuestro texto, vamos a incorporar un argumento. Llamaremos a este argumento “Message” y le asignaremos el tipo “String”.



En el proceso, cada fragmento que tenga una acción definida en **SafeArgs** generará una clase llamada “Directions” y creará un método para cada una de las acciones (flechas) que hayamos definido. Los parámetros de estos métodos serán la lista de argumentos que hemos creado. En este caso, solamente tenemos un argumento llamado “Message”.

Autor/a: Sabela Sobrino Última actualización: 08.02.2024

PASAR ARGUMENTOS

Ahora, vamos a cambiar el código en el fragmento **“MessageFragment”** para transmitir el mensaje al fragmento siguiente. En lugar de invocar directamente la siguiente pantalla al generar el Listener para el botón “Siguiente”, vamos a crear una **acción** a partir de la clase generada automáticamente llamada “MessageFragmentDirections”. Observa que esta clase tiene un método con el mismo nombre que la acción que hemos definido en el grafo, y como parámetros, tiene un argumento con el mismo nombre que el que definimos en el grafo.

```
// Crear una acción para la navegación
val actions = MessageFragmentDirections.actionMessageFragmentToEncryptFragment()

// Para obtener el texto ingresado por el usuario desde la caja de texto
val message = view.findViewById<EditText>(R.id.edit_text).text.toString()

// Pasar el mensaje como argumento a la acción
val actions = MessageFragmentDirections.actionMessageFragmentToEncryptFragment(message)

// Finalmente, transmitir esta acción al controlador de navegación
view.findNavController().navigate(actions)
```

En realidad, podríamos simplificar aún más este proceso, pasando todo como argumento directamente:

```
view.findNavController().navigate(
    MessageFragmentDirections.actionMessageFragmentToEncryptFragment(
        view.findViewById<EditText>(R.id.edit_text).text.toString()
    )
)
```

Este enfoque más conciso facilita la navegación y asegura que el mensaje se transmita correctamente al fragmento de destino.

Autor/a: Sabela Sobrino Última actualización: 08.02.2024

RECUPERAR ARGUMENTOS

Para recuperar los valores transmitidos entre fragmentos, debemos dirigirnos al archivo Kotlin correspondiente, en este caso, “EncryptMessage.kt”. Antes que nada, definiremos una **variable** para almacenar la **vista** que se construirá en este fragmento, lo que nos permitirá acceder a sus diferentes **elementos**. Lo que nos interesa especialmente en este momento es acceder al campo de texto que contendrá el texto cifrado.

```
val view = inflater.inflate(R.layout.fragment_encrypt, container, false)
return view
```

Para recuperar los elementos, aprovecharemos la clase “Args” que el plugin crea automáticamente. Esta clase ofrece varios métodos, pero el que nos interesa

es **“fromBundle”**, el cual nos permitirá acceder al bundle donde se almacenan los datos. La clase “Fragment” también cuenta con un método llamado **“requireArguments”** que nos proporcionará un bundle con los diferentes **valores**.

A partir del bundle, tendremos automáticamente generados una serie de atributos que coincidirán con el argumento que definimos en el grafo. En este caso, para acceder al mensaje transmitido, podemos utilizar el siguiente código:

```
val mensajeCifrado = EncryptFragmentArgs.fromBundle(requireArguments()).message
```

Luego, podemos guardar este valor en una constante y posteriormente pasar este valor a una función que se encargue de cifrarlo y mostrarlo en la interfaz.

Autor/a: Sabela Sobrino Última actualización: 08.02.2024

PRÁCTICA OPTATIVA. CREACIÓN DE MAPA CAMINO DE SANTIAGO

Desarrolla una aplicación llamada “UD03_3_MiCamino” que integre un **mapa interactivo** utilizando la **API de Google Maps** en Android. La aplicación mostrará los puntos marcados a lo largo del Camino Francés de Santiago de Compostela. Los requisitos son los siguientes:

1. **Mapa Interactivo**: Al abrir la aplicación, se mostrará un mapa centrado en la ruta del Camino Francés, indicando las paradas existentes en el mismo.
2. **Optativo. Descripción de los Puntos**: Cada marcador debe tener una breve descripción que se muestre al hacer clic en él, así como la opción de obtener direcciones desde la ubicación actual del usuario hasta el punto de interés.
3. **Optativo. Marcadores de Puntos de Interés**: Implementa marcadores en el mapa que representen puntos clave a lo largo del Camino Francés, como albergues, iglesias, y otros lugares de interés. Optativo: Al hacer clic en un marcador, se mostrará un `InfoWindow` con información adicional sobre el lugar.

Requisitos adicionales:

- Utiliza `Kotlin`.
- Implementa una interfaz de usuario intuitiva que facilite la navegación entre el mapa y la lista de puntos de interés.

Recursos:

- Sigue el codelab de Google como guía para la implementación de la funcionalidad del mapa y los marcadores.

Autor/a: Sabela Sobrino Última actualización: 20.01.2025

PRÁCTICA. CREACIÓN DE CUENTOS DE HALLOWEEN PERSONALIZADOS

Descripción de la Práctica:

En esta práctica, crearás una aplicación de Android UD03_2_Halloween que permitirá a los usuarios generar cuentos de Halloween personalizados. La aplicación constará de tres pantallas, cada una implementada como un fragmento.

Pantalla 1: Introducción de Nombre

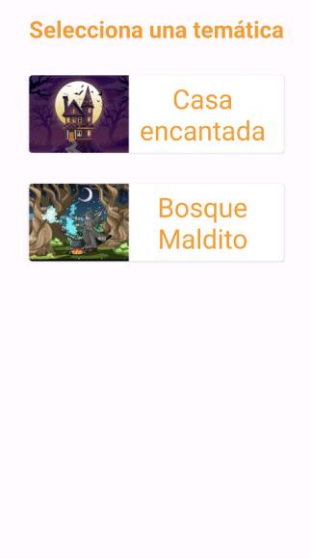
- En esta pantalla, los usuarios podrán introducir su nombre en un cuadro de texto.
- Deberás incluir una etiqueta para indicar que deben escribir su nombre.
- Un botón de “Continuar” les llevará a la siguiente pantalla una vez que hayan ingresado su nombre.

Tendrá el siguiente aspecto:

Pantalla 2: Selección de Temática

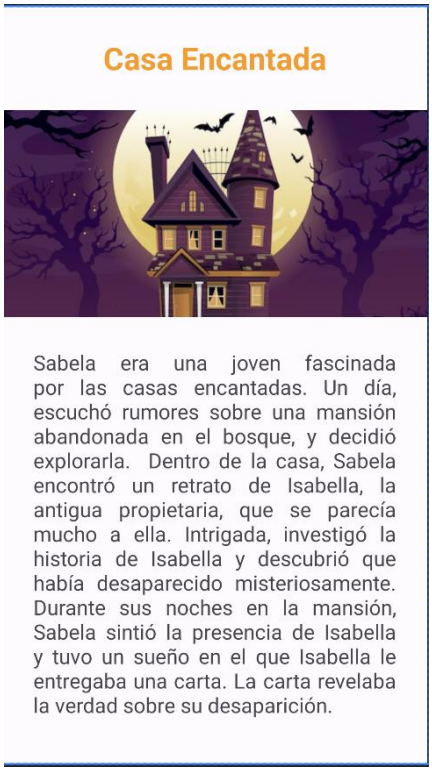
- En esta pantalla, los usuarios podrán elegir una temática para su cuento de Halloween. Debes proporcionar al menos tres opciones, como “Casa Encantada”, “Bosque Oscuro”, “Mansión Misteriosa”, entre otras.
- Cada opción debe presentarse como una `CardView` donde incluirás el texto y una imangen asociada.
- Al pulsar en alguna de las categorías, les llevará a la tercera pantalla después de hacer su elección.

Tendrá un aspecto similar a este:



Pantalla 3: Cuento Personalizado

- En esta pantalla, se generará un cuento personalizado basado en el nombre del usuario y la temática seleccionada.
- Deberás incluir una vista de texto para mostrar el cuento completo.
- Los detalles del cuento deben cambiar según el nombre ingresado y la temática seleccionada.
- Puedes agregar la imagen relacionada con la temática.
- Un botón de “Volver” en la parte inferior permitirá a los usuarios regresar a la pantalla de inicio para crear otro cuento si lo desean.



Requisitos Técnicos:

- Deberás utilizar fragmentos para implementar cada una de las pantallas.

- La interacción entre las pantallas debe gestionarse adecuadamente, de modo que la información ingresada por el usuario se utilice para generar el cuento personalizado.
- La aplicación debe tener un diseño atractivo y temático de Halloween.

Consideraciones Adicionales:

- Puedes utilizar recursos como imágenes o descripciones para cada temática de cuento.
- Asegúrate de que la aplicación sea fácil de usar y que los usuarios puedan seguir el flujo lógico de introducir su nombre, seleccionar una temática y leer su cuento personalizado.