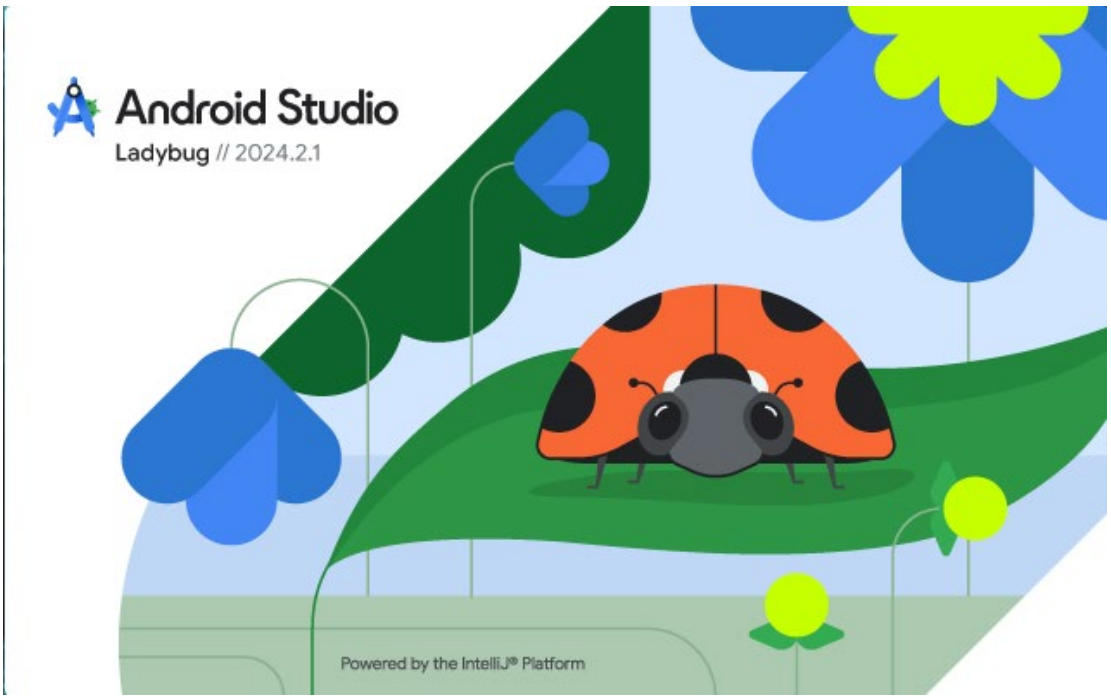
UNIVERSIDAD POLITECNICA DE VALENCIA

DISCA
DEPARTAMENTO DE INFORMÁTICA DE SISTEMAS Y COMPUTADORES

# Desarrollo de aplicaciones para dispositivos móviles

# PRÁCTICA 3: GESTIÓN LOCAL DE LA INFORMACIÓN




Escuela Técnica Superior de Ingeniería **Informática**

etsinf

## Objetivos

- Estructurar la capa de datos en repositorios y fuentes de datos con interfaces definidas.
- Utilizar el mecanismo de *DataStore* para almacenar y recuperar información.
- Utilizar *PreferenceFragment* para mostrar y gestionar opciones de configuración de la aplicación
- Utilizar *Room* para gestionar el almacenamiento de información en bases de datos SQL.

# Gestión local de la información

En esta práctica integraremos la capacidad de almacenar información, de manera local, en la aplicación que ya tenemos desarrollada. Sustituiremos la pantalla actual de configuración por una definida utilizando el mecanismo de *Settings* proporcionado por Android, utilizaremos *DataStore* para acceder a esta información desde diversas pantallas, y haremos uso de *Room* para almacenar las citas favoritas en una base de datos *SQLite*.

El aspecto de la pantalla de configuración de las preferencias de usuario se muestra en la Figura 1.

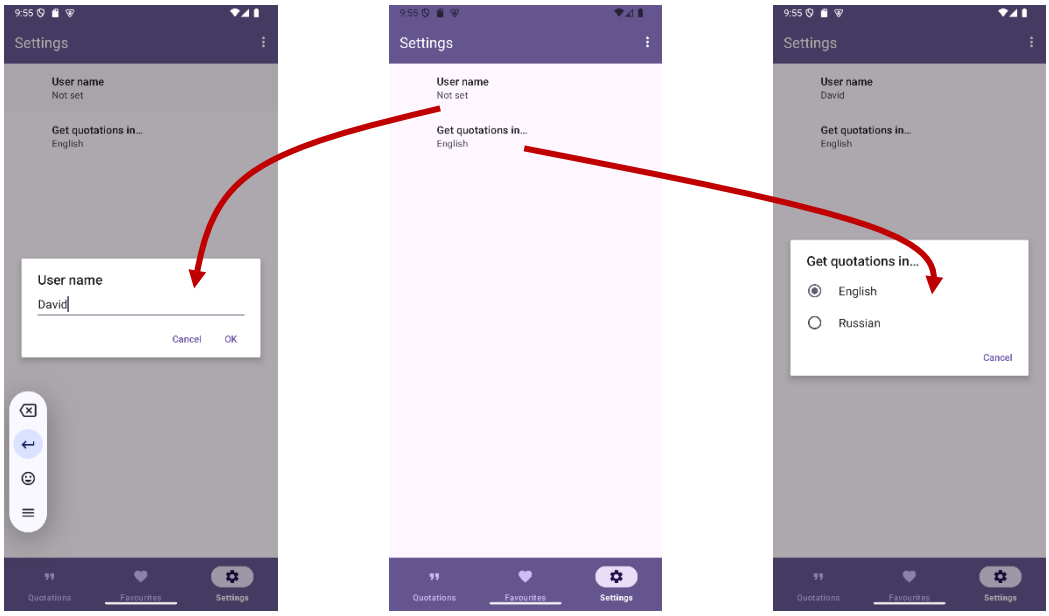



Figura 1. Pantalla de configuración de las preferencias de usuario.

## Ejercicio 01: Crear la pantalla de configuración (*Settings*)

**Descripción:** Para gestionar las preferencias del usuario en la aplicación (configuración) se creará una nueva pantalla que siga el aspecto definido para las pantallas de configuración de Android (*Settings*).

**Pasos a seguir:**

- Edita el fichero *build.gradle (Module:app)* e introduce la dependencia *implementation("androidx.preference:preference-ktx:1.2.1")*. Sincroniza el proyecto para que actualice las dependencias.
- Crea un nuevo recurso, llamado *preferences_settings.xml,* que represente la pantalla de configuración a crear (***File → New → Android resource file***, con *Resource type: XML* y *Root element: PreferenceScreen*).

- Crea un nuevo fichero de recursos de tipo *values* y denomínalo *arrays.xml*. Edita el fichero e incluye dos nuevos recursos de tipo <string-array/>. El primero debe contener dos elementos correspondientes de tipo *String* correspondientes a los idiomas en los que se pueden obtener las citas (*"English"*, *"Russian"* – recuerda definir estas etiquetas en strings.xml y traducirlas). El segundo debe contener otros dos elementos correspondientes al valor interno que utilizará la aplicación cuando el usuario seleccione esas opciones (*"en"* y *"ru"*, que corresponden al código de esos idiomas en el servicio web utilizado).
- Edita el fichero *preferences_settings.xml* para incluir un componente de tipo *EditTextPreference*, para introducir el nombre de usuario, y de tipo *ListPreference*, para seleccionar el idioma en el que se desea recibir las citas. Para este segundo, el asistente del entorno de desarrollo solicitará que indiques, primero, el recurso correspondiente a la información a mostrar en la lista y, después, el recurso asociado a los valores correspondientes. Modifica el título (*android:title*) de ambos componentes, asóciales una clave significativa (*android:key*), indica que debe mostrarse como resumen el valor actual de las preferencias (*app:useSimpleSummaryProvider="true"*), asocia un valor por defecto (*android:defaultValue*) a la lista (*"en"*, para que el idioma por defecto sea inglés), y asocia una pista (*android:hint*) al campo de texto para que indique que debe introducirse un valor (elimina a*ndroid:defaultValue* para el campo de texto).
- Edita el fichero *ui.settings.SettingsFragment* para que extienda *PrefenceFragmentCompat* y no reciba ningún argumento en su constructor. Elimina todo el código existente en la clase y sobrescribe el método *onCreatePreferences()* para crear la vista asociada al recurso XML que representa las preferencias del usuario – *setPreferencesFromResource(R.xml.preferences_settings, rootKey)*.
- Ejecuta la aplicación y comprueba que ahora, al acceder al fragmento de configuración (*Settings*) se muestra la pantalla definida. Comprueba que la información se introduce/selecciona adecuadamente y que, al terminar la aplicación, esta información es persistente.

## Ejercicio 02: Acceder a la preferencia del nombre de usuario

**Descripción:** El acceso a la información almacenada como preferencias de usuario se realizará por medio de *DataStore*. Esta información está almacenada en el espacio de almacenamiento privado de la aplicación, por lo que será necesario utilizar hilos optimizados para el acceso a entrada/salida.

**Pasos a seguir:**

- Edita el fichero *build.gradle (Module:app)* e introduce la dependencia *implementation("androidx.datastore:datastore-preferences:1.0.0")*. Sincroniza el proyecto para que actualice las dependencias.
- Crea una nueva interfaz denominada *SettingsDataSource* en el paquete *data.settings*. Edítala para definir el método *getUserName()* que devolverá como resultado

*Flow<String>,* el método *suspend getUserNameSnapshot()* que devolverá como resultado *String* y el método *suspend setUserName(userName: String)*.

- En el paquete *data.settings* crea una nueva clase *Kotlin* denominada *SettingsDataSourceImpl* que debe implementar la interfaz anteriormente definida. Modifica su constructor para que reciba una instancia de *DataStore<Preferences>*, necesaria para poder acceder a las preferencias del usuario, y etiqueta el constructor con *@Inject*, para que *Hilt* pueda proporcionar esta dependencia.
- Edita esta clase para definir un objeto que contendrá la constante que mantenga la clave de la preferencia correspondiente al nombre del usuario (*stringPreferenceKey("key")*) (debe ser la misma clave que se ha definido en el recurso */res/xml/preferences_settings.xml*).
- Implementa el método de la interfaz que devuelve el valor actual de la preferencia del nombre de usuario a partir de la instancia de *DataStore* recibida en el constructor – *suspend fun getUserNameSnapshot(): String = dataStore.data.first()[USER_NAME].?: "".* El valor por defecto para el nombre de usuario es un *String* vacío.
- Implementa el método de la interfaz que actualiza la preferencia con el nombre del usuario a partir de la instancia de *DataStore* recibida en el constructor –

```
suspend fun setUserName(userName: String) {
    dataStore.edit { preferences ->
      preferences[USER_NAME] = userName
    }
  }
```

- Crea una nueva interfaz denominada *SettingsRepository* en el paquete *data.settings*. Edítala para definir el método *getUserName()* que devolverá como resultado *Flow<String>*, *e*l método *suspend getUserNameSnapshot()* que devolverá como resultado *String* y el método *suspend setUserName(userName: String)*.
- En el paquete *data.settings* crea una nueva clase *Kotlin* denominada *SettingsRepositoryImpl* que debe implementar la interfaz anteriormente definida. Modifica su constructor para que reciba una instancia de *SettingsDataSource* y etiqueta el constructor con *@Inject*.
- Edita esta clase para implementar los métodos de la interfaz utilizando la fuente de datos.
- Crea una nueva clase denominada *SettingsPreferenceDataStore* en el paquete *data.settings* que extienda *PreferenceDataStore* y reciba en su constructor una instancia de *SettingssRepository*. Etiqueta el constructor con @Inject para poder recibir esa dependencia.
- Esta clase es la que actúa de puente entre la interfaz gráfica de *Settings* y el *DataStore* creado. Por ello, es necesario implementar los métodos necesarios para poder recuperar y almacenar la información necesaria. En nuestro caso se trata de String, tanto para el nombre de usuario como para el idioma seleccionado, por lo que deben implementarse los métodos *putString()* y *getString()*. Ambos métodos deben crear una nueva corrutina que utilice un hilo de ejecución especializado en entrada/salida para acceder al método correspondiente del repositorio dependiendo de la clave de la preferencia que se desee leer/escribir. En el caso de *putString()*, crea el hilo con

*CoroutineScope(Dispatcher.IO).launch{}*. En el caso de *getString()* debe bloquearse la ejecución hasta obtener el dato del repositorio, por lo que debe crearse el hilo con *runBlocking(Dispatcher.IO).launch {}*.

- Crea una nueva clase abstracta *Kotlin* en el paquete *di*, denomínala *SettingsBinderModule*, y anótala con *@Module*, para indicar que es un módulo *Hilt*, y *@InstallIn(SingletonComponent::class)*, para indicar que las instancias de las dependencias proporcionadas se mantendrán a lo largo del ciclo de vida de la aplicación.
- En esa clase, crea dos nuevas funciones abstractas denominadas *bindSettingsDataSource()* y *bindSettingsRepository()*. que reciban como parámetro un objeto del tipo de la implementación del origen de datos y del repositorio, respectivamente, y devuelvan como resultado un objeto del tipo de la interfaz del origen de datos y del repositorio, respectivamente. Al anotarla con *@Binds*, esto indica a *Hilt* que debe crear un objeto del tipo de la implementación cuando se requiera un objeto del tipo de la interfaz.
- Crea una nueva clase *Kotlin* en el paquete *di*, denomínala *SettingsProviderModule*, y anótala con *@Module*, para indicar que es un módulo *Hilt*, y *@InstallIn(SingletonComponent::class)*, para indicar que las instancias de las dependencias proporcionadas se mantendrán a lo largo del ciclo de vida de la aplicación.
- En esa clase, crea una nueva función denominada *provideSettingsPreferenceDataStore()* que reciba como parámetro un objeto del tipo *SettingsRepository*, y devuelva como resultado una instancia de *PreferenceDataStore*, obtenida al crear un nuevo objeto *SettingsPreferenceDataStore* por medio de la instancia de *SettingsRepository* recibida como parámetro. Al anotar la función con *@Provides*, se indica a *Hilt* cómo puede crear objetos del tipo proporcionado por la función, y al anotarla *@Singleton* se indica que siempre se devolverá la misma instancia del resultado.

  En esa misma clase, crea una nueva función denominada *provideSettingsDataStore()* que reciba como parámetro el contexto de la aplicación y proporcione una instancia de *DataStore<Preference>*. El propósito de este método es crear el repositorio en el que se almacenarán las preferencias y gestionar su acceso de forma única. Al anotar la función con *@Provides*, se indica a *Hilt* cómo puede crear objetos del tipo proporcionado por la función, y al anotarla *@Singleton* se indica que siempre se devolverá la misma instancia del resultado. Debe utilizarse la factoría existente para crear el fichero, indicando que se devolverá un conjunto de preferencias vacías si hay algún problema al acceder al fichero, que no se van a realizar operaciones para migrar entre versiones de la aplicación (solo tenemos esta versión) y que se accederá por medio de una corrutina que utilice hilos de ejecución especializados en entrada/salida:

```
@Provides
@Singleton
fun provideSettingsDataStore(
  @ApplicationContext context: Context
): DataStore<Preferences> {
  return PreferenceDataStoreFactory.create(
     corruptionHandler = ReplaceFileCorruptionHandler { emptyPreferences() },
     migrations = listOf(),
     scope = CoroutineScope(Dispatchers.IO + SupervisorJob()),
     produceFile = { context.preferencesDataStoreFile(<MyPreferenceFileName>) }
    )
}
```

- Edita la clase *SettingsFragment* del paquete *ui.settings* para que disponga de un atributo variable que almacene una referencia a *SettingsPreferenceDataStore*. Anota el atributo como *@Inject* para permitir que *Hilt* proporcione la dependencia necesaria. Como no dispondrá de valor inicial deberás anotarlo como *lateinit*. Anota la clase con *@AndroidEntryPoint* para permitir la inyección. Edita el método *onCreatePreferences()* para que la primera acción realizada sea asociar esta instancia de *SettinsPreferenceDataStore* al gestor de preferencias (*preferenceManager.preferenceDataStore*).
- Ejecuta la aplicación y accede a la pantalla de configuración. Comprueba que si modificas el nombre del usuario y cierras la aplicación (asegúrate de matarla desde el emulador) se recupera correctamente el valor almacenado al lanzarla nuevamente a ejecución.
- Edita la clase *SettingsDataSourceImpl* del paquete *data.settings* e implementa el método *getUserName()* para obtener un *Flow* que emita el nombre de usuario cada vez que se produzca un cambio. Utiliza la función *catch {}* para capturar los errores que puedan ocurrir. En caso de ser de tipo *IOException* emite unas *Preferences* vacías y, en caso contrario lanza la excepción recibida. Mientras no ocurran excepciones, transforma las *Preferences* recibidas mediante *map {}* para devolver solo el valor almacenado como nombre de usuario.

```
dataStore.data.catch { exception ->
   if (exception is IOException) {
      emit(emptyPreferences())
   } else throw exception
}.map { preferences ->
   preferences[USER_NAME].orEmpty()
}
```

- Edita la clase *SettingsRepositoryImpl* del paquete *data.settings* e implementa el método *getUserName()* para devolver el *Flow* proporcionado por la fuente de datos.
- Edita la clase.*NewQuotationViewModel* del paquete *ui.newquotation* para que reciba en su constructor un objeto de tipo *SettingsRepository*. Elimina la propiedad privada de tipo *MutableStateFlow<String>* que mantiene el nombre del usuario, ya que no será

necesaria. Modifica la propiedad que expone el nombre del usuario para que su valor sea directamente la conversión a *StateFlow* del *Flow* que proporciona el repositorio –

```
settingsRepository.getUsername().stateIn(
    scope = viewModelScope,
    initialValue = "",
    started = SharingStarted.WhileSubscribed()
)
```

- Ejecuta la aplicación y observa el nombre que se muestra en el mensaje de bienvenida. Accede a la pantalla de configuración y modifica el nombre de usuario. Regresa a la pantalla de obtener nuevas citas y el mensaje de bienvenida debería haber cambiado para mostrar el nuevo nombre del usuario.

## Ejercicio 03: Obtener las citas en el idioma seleccionado por el usuario

**Descripción:** El código del idioma seleccionado también se almacena por medio de *DataStore*. Sin embargo, la interfaz de usuario no precisa de esta información, es suficiente con que el repositorio la recupere antes de acceder al origen de datos para obtener una nueva cita.

**Pasos a seguir:**

- Edita las interfaces *SettingsDataSource* y *SettingsRepository del paquete data.settings* para definir métodos de acceso y modificación del lenguaje en el que se desean obtener las citas. Deben ser similares a los utilizados para gestionar el nombre de usuario (el lenguaje también es un *String*).
- Edita la clase *SettingsDataSourceImpl* del paquete *data.settings* para que implemente los nuevos métodos de la interfaz. Para ello, añade la nueva constante necesaria al objeto que mantiene las claves asociadas a las preferencias. La implementación de los nuevos métodos de la interfaz es idéntica a la existente para gestionar el nombre de usuario, simplemente es necesario cambiar la clave utilizada para acceder al dato almacenado en el *DataStore*. Sería interesante factorizar este código para alojarlo en métodos privados que permitan realizar las operaciones suministrando como parámetro la clave a utilizar. El valor por defecto del lenguaje será “en”.
- Edita la clase *SettingsRepositoryImpl* del paquete *data.settings* para que implemente los nuevos métodos definidos en la interfaz.
- Edita la clase *SettingsPreferenceDataStore* del paquete *data.settings* para que los métodos *puString()* y *getString()* gestionen correctamente la clave asociada al lenguaje seleccionado por el usuario.
- Ejecuta la aplicación y accede a la pantalla de configuración. Comprueba que si modificas el idioma seleccionado (debería aparecer por defecto “*English*”) y cierras la aplicación (asegúrate de matarla desde el emulador) se recupera correctamente el valor almacenado al lanzarla nuevamente a ejecución.

- Edita la clase *NewQuotationRepositoryImpl* en el paquete *data.newquotation* para que su constructor reciba una instancia de *SettingsRepositor*.
- Edita esta clase para que disponga de un atributo privado *lateinit* (se inicializará posteriormente) en el que se almacene el código del idioma seleccionado por el usuario. Para obtener este valor debes consumir, mediante el método *collect {}*, el *Flow* que proporciona el repositorio. Para ello lanza una nueva corrutina en el código asociado al constructor de la clase (*init {}*). El código necesario sería:

```
private lateinit var language: String

init {
    CoroutineScope(SupervisorJob()).launch {
        settingsRepository.getLanguage().collect { languageCode ->
            language = languageCode.ifEmpty{ "en" }
        }
    }
}
```

- Modifica el método *getNewQuotation()* para que haga uso de la variable que almacena el lenguaje seleccionado.
- Ejecuta la aplicación y selecciona inglés como idioma de las citas a recibir. Obtén algunas citas y observa cómo se reciben en inglés. Cambia el idioma seleccionado a ruso y obtén nuevas citas, que deberán recibirse en ese otro idioma. Debería funcionar correctamente al rotar el dispositivo, matar la aplicación, etc.

## Ejercicio 04: Gestión de citas favoritas en base de datos *SQLite* por medio de *Room*

**Descripción:** La gestión de bases locales *SQLite* por medio de la biblioteca *Room* es bastante cómoda. Debe definirse una clase de datos *(@Entity*) por cada tabla que contenga la base de datos, una interfaz (*@Dao*) que defina los métodos que se exponen para las operaciones sobre cada tabla de la base de datos, y una clase que cree la base de datos *(@Database*) y dé acceso a los DAO.

**Pasos a seguir:**

- Edita el fichero *build.gradle (Module: app*) e incluye las dependencias de Room y su procesador de anotaciones – *implementation("androidx.room:room-runtime:2.6.1"), implementation("androidx.room:room-ktx:2.6.1")* y *ksp("androidx.room:room-compiler:2.6.1")*.
- Sincroniza el proyecto para que actualice las dependencias.
- Crea un nuevo objeto *Kotlin* denominado *FavouritesContract* en el paquete *data.favourites*. Edítalo para que exponga una constante con el nombre de la base de datos que almacenará las citas favoritas. Crea dentro otro objeto que exponga una

constante con el nombre que tendrá la tabla en la base de datos, y una constante con el nombre de cada una de las columnas que contendrá (identificador, texto y autor).

- Crea un nuevo paquete dentro de *data.favourites* denominado *model*. Crea una clase de datos *Kotlin* denominada *DtaabaseQuotationDto* que disponga de propiedades de solo lectura con el mismo nombre que las propiedades de la clase *domain.model.Quotation* (*id*, *text*, *author*). Anota la clase con *@Entity (tableName = TABLE_NAME)* para que Room pueda crear una tabla en la base de datos a partir de la información proporcionada por esta clase. Anota la propiedad correspondiente al identificador con *@PrimaryKey* para indicar que será la clave primaria de la tabla. Anota cada propiedad con *@ColumnInfo(name = COLUM_NAME)* para indicar el nombre que tendrá la columna correspondiente en la base de datos. Tanto el nombre de la tabla como el nombre de las columnas se obtendrá de las constantes previamente definidas en el objeto *Kotlin*.
- Crea una nueva interfaz *Kotlin* denominada *FavouritesDao* en el paquete *data.favourites* y anótala con *@Dao*. Esta clase define las operaciones de acceso a la base de datos. Crea el método *suspend* que permitirá añadir la cita recibida como parámetro (*DatabaseQuotationDto*) a la base de datos. Deberá anotarse el método con *@Insert(onConflict = OnConflictStrategy.REPLACE)* para que se reemplace la cita si ya existe. Crea el método *suspend* que permitirá eliminar la cita recibida como parámetro (*DatabaseQuotationDto*) de la base de datos. Deberá anotarse el método con *@Delete*. Crea el método que permitirá obtener todas las citas favoritas de la base de datos. Deberá anotarse con la *query* que realiza esta operación – *@Query("SELECT * FROM $TABLE_NAME")* – y deberá devolver *Flow<List<DatabaseQuotationDto>>*. Crea el método que permitirá obtener una cita concreta a partir del identificador recibido como parámetro. Deberá anotarse con la *query* que realiza esta operación – *@Query("SELECT * FROM $TABLE_NAME WHERE $COLUMN_ID = :id")*– y deberá devolver *Flow<DatabaseQuotationDto>*. Crea el método que permitirá borrar todas las citas de la base de datos. Deberá anotarse con la *query* que realiza esta operación – -*@Query("DELETE FROM $TABLE_NAME")*.
- Crea una nueva clase abstracta *Kotlin* denominada *FavouritesDatabase* en el paquete *data.favourites*. Debe extender *RoomDatabase* y debe anotarse con *@Database(entities = [DatabaseQuotationDto::class], version = 1)* para indicar qué clases definen las tablas a crear y el número de versión de la base de datos. Edita la clase para incluir una función abstracta que devuelva una instancia del DAO previamente definido – *abstract fun favouritesDao(): FavoritesDao*. Esta clase proporciona la implementación de la base de datos y de la interfaz DAO.
- Crea una nueva interfaz denominada *FavouritesDataSource* en el paquete *data.favourites*. Edítala para definir los métodos necesarios para poder utilizar aquellos definidos en el DAO.
- En el paquete *data.favourites* crea una nueva clase *Kotlin* denominada *FavouritesDataSourceImpl* que debe implementar la interfaz anteriormente definida. Modifica su constructor para que reciba una instancia de *FavouritesDao*, necesaria para poder acceder a las operaciones sobre base de datos, y etiqueta el constructor con *@Inject*, para que *Hilt* pueda proporcionar esta dependencia. Implementa todos

los métodos de la interfaz, de forma que accedan a las operaciones proporcionadas por el DAO.

- Crea un nuevo fichero *Kotlin* (no clase *Kotlin*, solo fichero) en el paquete *data.favourites.model* denominado *DatabaseQuotationDtoMapper.kt*. Crea una función extendida para añadir el método *toDomain()* a la clase *DatabaseQuotationDto*. Esta función deberá crear una instancia de *Quotation* a partir de los datos existentes en *DatabaseQuotationDto*, de tal forma que se asignarán el identificador, texto y autor a partir de los campos disponibles,
- Crea en este fichero otra función extendida para añadir el método *toDatabaseDto()* a la clase *Quotation*. Esta función deberá crear una instancia de *DatabaseQuotationDto* a partir de los datos existentes en *Quotation*.
- Crea una nueva interfaz denominada *FavouritesRepository* en el paquete *data.favourites*. Edítala para definir los métodos necesarios para poder utilizar aquellos proporcionados por el origen de datos (los mismos métodos, pero utilizando instancias de *Quotation* en lugar de *DatabaseQuotationDto*).
- En el paquete *data.favourites* crea una nueva clase *Kotlin* denominada *FavouritesRepositoryImpl* que debe implementar la interfaz anteriormente definida. Modifica su constructor para que reciba una instancia de *FavoritesDataSource* y etiqueta el constructor con *@Inject*.
- Edita la clase para implementar todos los métodos de la interfaz, de tal forma que invoquen los métodos disponibles en el origen de datos y realice la conversión entre los tipos *Quotation* y *DatabaseQuotationDto* conforme sea necesario. Para convertir el *Flow<List<DatabaseQuotationDto>>* en *Flow<List<Quotation>>* necesitarás dos *.map{}* anidados (uno transforma el dato recibido *List<DatabaseQuotationDto>* y el otro para transformar cada uno de los elementos de la lista a *Quotation*).
- Crea una nueva clase *Kotlin* en el paquete *di*, denomínala *FavouritesProviderModule*, y anótala con *@Module*, para indicar que es un módulo *Hilt*, y *@InstallIn(SingletonComponent::class)*, para indicar que las instancias de las dependencias proporcionadas se mantendrán a lo largo del ciclo de vida de la aplicación.
- En esa clase, crea una nueva función denominada *provideFavouritesDatabase()* que reciba como parámetro un objeto del tipo *Context*, anotado como *@ApplicationContext* para que se inyecte directamente el contexto de la aplicación, y devuelva como resultado una instancia de *FavouritesDatabase*. Esta instancia puede obtenerse con el método estático *Room.databaseBuilder()*, pasándole como parámetros el contexto, la clase de la base de datos que se desea crear (*FavouritesDatabase::class.java*) y el nombre de la base de datos (definido en *FavouritesContract*) e invocando su método *build()*. Al anotar la función con *@Provides*, se indica a *Hilt* cómo puede crear objetos del tipo proporcionado por la función, y al anotarla *@Singleton* se indica que siempre se devolverá la misma instancia del resultado.
- Crea otra función anotada con @Provides, en esta misma clase, que proporcione una instancia de *FavouritesDao* a partir de la instancia de *FavouritesDatabase* recibida como parámetro.

- Crea una nueva clase abstracta *Kotlin* en el paquete *di*, denomínala *FavouritesBinderModule*, y anótala con *@Module*, para indicar que es un módulo *Hilt*, y *@InstallIn(SingletonComponent::class)*, para indicar que las instancias de las dependencias proporcionadas se mantendrán a lo largo del ciclo de vida de la aplicación.
- En esa clase, crea dos nuevas funciones abstractas denominadas *bindFavouritesDataSource()* y *bindFavouritesRepository()*. que reciban como parámetro un objeto del tipo de la implementación del origen de datos y del repositorio, respectivamente, y devuelvan como resultado un objeto del tipo de la interfaz del origen de datos y del repositorio, respectivamente. Al anotarla con *@Binds*, esto indica a *Hilt* que debe crear un objeto del tipo de la implementación cuando se requiera un objeto del tipo de la interfaz.
- Edita la clase *FavouritesViewModel* del paquete *ui.favourites* para que reciba en su constructor una instancia de *FavouritesRepository*. Elimina la propiedad privada de tipo *MutableStateFlow<List<Quotation>>* que mantiene la lista de citas favoritas, ya que no será necesaria. Modifica la propiedad que expone esta lista para que su valor sea directamente la conversión a *StateFlow* del *Flow* que proporciona el repositorio –

  ```
  favouritesRepository.getAllFavouriteQuotations().stateIn(
      scope = viewModelScope,
      initialValue = listOf(),
      started = SharingStarted.WhileSubscribed()
  )
  ```

  Elimina el método que genera la lista de citas aleatorias, puesto que ya no es necesario. Comenta el código de las funciones que permite borrar citas, ya que deben corregirse.

- Ejecuta la aplicación y comprueba que no se produce ningún error y no aparece ninguna cita como favorita, ya que la base de datos está vacía.

## Ejercicio 05: Inserción de citas favoritas en la base de datos

**Descripción:** Una vez dispones de acceso a todas las operaciones sobre la base de datos a través del DAO, la inserción de nuevas citas simplemente requerirá la utilización de estas operaciones a través del repositorio correspondiente.

**Pasos a seguir:**

- Edita la clase *NewQuotationViewModel* del paquete *ui.newquotation* para que reciba una instancia de *FavouritesRepository* en su constructor.
- Modifica el método que añade nuevas citas favoritas para que se ejecute el método apropiado del repositorio dentro de una nueva corrutina (*viewModelScope.launch{}*). El código existente que oculta el botón de añadir citas a favoritos deberá ubicarse dentro de la corrutina.

- Ejecuta la aplicación y comprueba que, al pulsar el botón de añadir citas a favoritas, estas citas aparecen ahora en la lista de citas favoritas. Cierra la aplicación, vuélvela a abrir y comprueba que las citas siguen apareciendo en la lista de favoritas (se han almacenado y recuperado de la base de datos).

## Ejercicio 06: Visibilidad del botón de añadir citas como favoritas

**Descripción:** En lugar de mostrar siempre el botón de añadir la cita a favoritas, deberíamos comprobar si la cita ya existe en la base de datos y, en ese caso, no mostrar el botón. Para ello, se utilizará la operación de buscar citas en la base de datos a través de su identificador.

**Pasos a seguir:**

- Edita la interfaz *data.favourites.FavouritesDao* y todas las clases que lo utilicen para que el método que devuelve una cita de la base de datos a partir de identificador pueda devolver un valor *null* (*Flow<Quotation?>*) o una lista vacía (*Flow<List<Quotation>>*) si la cita no está en la base de datos (elige una de las dos soluciones).
- Edita la clase *ui.newquotation.NewQuotationViewModel* para eliminar la propiedad privada de tipo *MutableStateFlow<Boolean>* que mantiene la visibilidad del botón de añadir citas como favoritas, ya que no será necesaria (y las asignaciones que se realizan al obtener una nueva cita o añadirla a favoritos). Modifica la propiedad que expone este *Boolean* para que su valor se obtenga de la siguiente forma. El método disponible en el repositorio para obtener una cita concreta de la base de datos a partir de su identificador ahora devuelve como resultado un *Flow<Quotation?>* o *Flow<List<Quotation>>*, por lo que su método extendido *map{}* te permitirá convertirlo en *Flow<Boolean?>* dependiendo de si existe esa cita en la base de datos o no. Sin embargo, esta función deberá ejecutarse cada vez que cambie la cita que se ha recibido del servicio web, por lo que deberías poder observar cambios en la propiedad existente asociada. Si tenemos en cuenta que cada llamada a la función para recuperar la cita de la base de datos devolverá un nuevo *Flow*, deberás utilizar la transformación *flatMapLatest{}* para cambiar de *Flow* cada vez que se emita un nuevo valor. Finalmente, como lo que deseamos obtener es un *StateFlow*, deberás obtenerlo mediante *stateIn()*.

```
val isAddToFavouritesVisible = quotation.flatMapLatest { currentQuotation ->
    if (currentQuotation == null) flowOf(false)
    else favouritesRepository.getFavouriteQuotationById(currentQuotation.id)
        .map { quotationInDatabase ->
            quotationInDatabase == null
        }
}.stateIn(
    scope = viewModelScope,
    initialValue = false,
    started = SharingStarted.WhileSubscribed()
)
```

- Para poder comprobar que funciona correctamente, modifica el método *getNewQuotation()* de la clase *NewQuotationViewModel del paquete ui.newquotation* de tal forma que el 50% de las veces acceda al servidor para obtener una cita y el otro 50% de las veces obtenga siempre una cita que habrás creado manualmente (siempre la misma).
- Ejecuta la aplicación y comprueba que el botón de añadir citas a favoritos no se muestra por defecto. Obtén una nueva cita y comprueba que el botón sí aparece ahora. Obtén citas hasta que aparezca la cita que has definido manualmente. Pulsa en el botón para añadirla a favoritos, con lo que debería ocultarse. Solicita varias citas y observa como el botón se muestra para las citas proporcionadas por el servidor (a menos que tengas mala suerte y recuperes una que ya tengas en la base de datos) y se oculta para la cita que has creado manualmente y ya está en la base de datos. Borra el código adicional de prueba para que el método *getNewQuotation()* recupere siempre las citas del servidor.

## Ejercicio 07: Eliminación de citas favoritas

**Descripción:** Una vez dispones de acceso a todas las operaciones sobre la base de datos a través del DAO, la eliminación de citas favoritas simplemente requerirá la utilización de estas operaciones a través del repositorio correspondiente.

**Pasos a seguir:**

- Edita la clase *FavouritesViewModel* del paquete *ui.favourites* para que el método que borra una cita a partir de su posición en la lista ejecute el método apropiado del repositorio dentro de una nueva corrutina (*viewModelScope.launch{}*).
- Ejecuta la aplicación y comprueba que, al hacer *swipe right* sobre alguna de las citas favoritas, se elimina de la base de datos.
- En el misma clase, modifica el método que borra todas las citas para que ejecute el método apropiado del repositorio dentro de una nueva corrutina (*viewModelScope.launch{}*).
- Ejecuta la aplicación y comprueba que, al seleccionar la opción del menú de opciones y aceptar el borrado en el diálogo resultante, se eliminan todas las citas de la base de datos.

Con esto tendrás una aplicación simple que cubre todos los aspectos básicos del desarrollo de aplicaciones Android: interfaz gráfica, comunicaciones HTTP y almacenamiento local de la información, siguiendo la arquitectura propuesta por Google.