Capitulo 7

UD07. PERSISTENCIA DE DATOS

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

- CA1.8 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.

- Contidos
- Contornos integrados de traballo.
- Módulos para o desenvolvemento de aplicacións móbiles.
- Emuladores.
- Estrutura dunha aplicación para dispositivo móbil.
- Modificación de aplicacións existentes.
- Uso do contorno de execución do administrador de aplicacións.
- Ferramentas e fases de construción.
- Técnicas de animación e son.
- Comunicacións: clases asociadas. Tipos de conexións.
- Xestión da comunicación sen fíos.
- Envío e recepción de mensaxes de texto: seguridade e permisos.
- Envío e recepción de mensaxaría multimedia: sincronización de contido; seguridade e permisos.

Autor/a: Sabela Sobrino Última actualización: 08.02.2024

SUBSECCIONES DE UD07. PERSISTENCIA DE DATOS

Capitulo 1

COMPOSE

En el contexto de Android, “Compose” generalmente se refiere a Jetpack Compose, que es un moderno kit de herramientas de UI (interfaz de usuario) declarativo y totalmente nativo para el desarrollo de aplicaciones Android. Jetpack Compose permite a los desarrolladores construir interfaces de usuario de manera más sencilla y más eficiente mediante la creación de interfaces de usuario mediante código Kotlin.

Para poder realizar un proyecto con Compose debemos seleccionar la plantilla “Empty View”.

Autor/a: Sabela Sobrino Última actualización: 08.02.2024

SUBSECCIONES DE COMPOSE

PREVISUALIZACIÓN

En el código proporcionado, la opción de “Previsualización” se refiere a la capacidad de Jetpack Compose para proporcionar una vista previa visual en tiempo de diseño de cómo se verá un componente en la interfaz de usuario antes de que se ejecute la aplicación. La función `@Preview` se utiliza para definir vistas previas en Compose.

En el código que compartiste:

```
@Preview(showBackground = true)
@Composable
fun GreetingPreview() {
    Prueba_composeTheme {
        Greeting("Android")
    }
}
```

Aquí, `@Preview` se utiliza para crear una vista previa de la función `Greeting`. La función `GreetingPreview` se compone de la función `Greeting` y se le proporciona un nombre (`"Android"`) como argumento para simular cómo se vería la interfaz de usuario cuando se utiliza la función `Greeting` con ese argumento específico.

La opción `showBackground = true` indica que se debe mostrar un fondo en la vista previa, lo que puede ser útil para visualizar cómo se integrará el componente en el diseño general de la interfaz de usuario.

Autor/a: Sabela Sobrino Última actualización: 08.02.2024

SURFACE

En el contexto de Jetpack Compose, una `Surface` es un componente que actúa como un contenedor visual y define un área rectangular en la interfaz de usuario. La función principal de la `Surface` es proporcionar un fondo para otros componentes y establecer características visuales para ese área específica.

Algunas características clave de `Surface` incluyen:

1. **Definición de Fondo:** Puedes especificar un color de fondo para la `Surface`, que afectará a la apariencia visual de cualquier componente contenido en ella.
2. **Gestión de Propiedades Visuales:** La `Surface` puede gestionar propiedades visuales como sombras, bordes y más, lo que permite personalizar la apariencia del área que cubre.
3. **Contenedor para Otros Componentes:** Puedes anidar otros componentes dentro de una `Surface`, lo que permite componer interfaces de usuario más complejas y estructuradas.

En el contexto de nuestro código:

```
@Composable
fun Greeting(name: String, modifier: Modifier = Modifier) {
    // La Surface toma un color, en este caso, el color primary del tema.
    Surface(color = MaterialTheme.colorScheme.primary) {
        // Los componentes anidados dentro de Surface se dibujarán sobre ese color de fondo.
        Text(
            text = "Hello $name!",
            modifier = modifier
        )
    }
}
```

Aquí, la `Surface` se utiliza para definir el color de fondo (en este caso, el color primario del tema) sobre el cual se dibujará el componente `Text`.

Autor/a: Sabela Sobrino Última actualización: 08.02.2024

MODIFICADORES

En Compose, la mayoría de los elementos de la interfaz de usuario, como `Surface` y `Text`, incorporan un parámetro opcional denominado `modifier`. Estos modificadores permiten especificar cómo debe presentarse o comportarse un elemento en el diseño general.

```
Text(
    text = "Hello $name!",
    modifier = modifier.padding(Dp(15f))
)
```

En el ejemplo anterior, se aplica un modificador para establecer un relleno (padding) alrededor del componente `Text`, proporcionando así un espacio adicional alrededor del texto.

Consulta la lista completa de modificadores para explorar las diversas opciones disponibles para personalizar la apariencia y el comportamiento de los elementos de la interfaz de usuario en Jetpack Compose.

Autor/a: Sabela Sobrino Última actualización: 08.02.2024

COMPONIBLES

Con el objetivo de promover la **reutilización de código**, se propone la creación de un elemento **componible** llamado `MyApp` que encapsule la funcionalidad de mostrar un saludo. Como una práctica recomendada, se sugiere incluir un parámetro de modificador en la función, asignándole un modificador vacío de forma predeterminada.

```
@Composable
private fun MyApp(modifier: Modifier = Modifier) {
    Surface(
        modifier = modifier,
        color = MaterialTheme.colorScheme.background
    ) {
        Greeting("Android")
    }
}
```

Esta implementación posibilita la simplificación de la devolución de llamada `onCreate` y la vista previa, ya que ahora es posible **reutilizar** el elemento componible `MyApp`, evitando la duplicación de código.

En la clase `MainActivity`, el uso de `MyApp` se integra de la siguiente manera:

```
class MainActivity : ComponentActivity() {
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContent {
            // En lugar de un archivo XML, se invoca a una función de Compose
            Prueba_composeTheme {
                // Elementos componibles
                MyApp(modifier = Modifier.fillMaxSize())
            }
        }
    }
}
```

Asimismo, en la vista previa, se puede observar la reutilización de `MyApp` de la siguiente manera:

```
@Preview(showBackground = true)
@Composable
fun GreetingPreview() {
    Prueba_composeTheme {
        // Elementos componibles
```

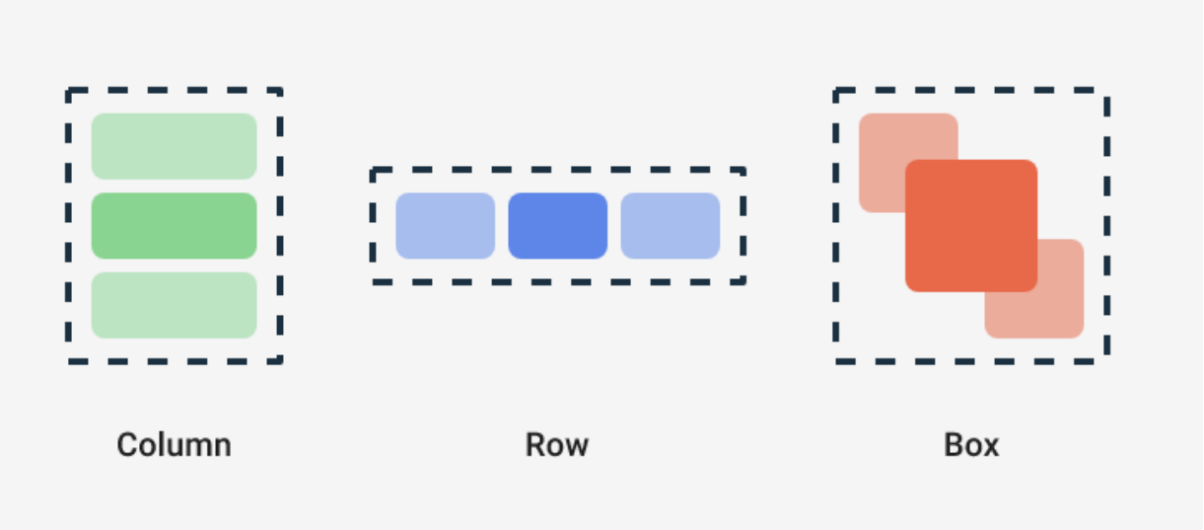
```
        MyApp()
    }
}
```

Esta estructura fomenta la coherencia y facilita la gestión de componentes componibles en la aplicación, mejorando la legibilidad y la mantenibilidad del código.

Autor/a: Sabela Sobrino Última actualización: 08.02.2024

ELEMENTOS BÁSICOS

Los componentes fundamentales en Compose son **Column**, **Row** y **Box**, representando estructuras de diseño estándar.



Pueden combinarse para lograr disposiciones más complejas y flexibles. En el siguiente código, ilustramos cómo crear una columna dentro de la función Greeting:

```
@Composable
fun Greeting(name: String, modifier: Modifier = Modifier) {
    Surface(color = MaterialTheme.colorScheme.primary) {
        Column(modifier = Modifier.padding(Dp(24f))) {
            Text(text = "Hello,")
            Text(text = name)
        }
    }
}
```

En este ejemplo, hemos encapsulado dos elementos de texto (`Text`) dentro de una columna (`Column`). La columna se ha configurado con un modificador para agregar un relleno alrededor de los elementos. La superficie (`Surface`) establece el color de fondo de la composición.

Autor/a: Sabela Sobrino Última actualización: 08.02.2024

ROOM

En el contexto de desarrollo de aplicaciones Android, “Room” se refiere a una biblioteca de persistencia que proporciona una capa de abstracción sobre SQLite, que es una base de datos relacional incorporada en Android. Room simplifica el manejo de la base de datos y ofrece una forma más robusta y eficiente de realizar operaciones de base de datos en comparación con el uso directo de SQLite.

Las principales características de Room incluyen:

1. **Entity:** Representa una tabla en la base de datos. Cada instancia de la entidad representa una fila en esa tabla.
2. **DAO (Data Access Object):** Define métodos que acceden a la base de datos. Estos métodos pueden incluir consultas SQL personalizadas.
3. **Database:** Es una clase que sirve como punto de acceso principal para la base de datos. Se anota con la anotación `@Database` y debe ser una extensión de la clase `RoomDatabase`. La clase `Database` generalmente incluye una lista de entidades y proporciona un método abstracto que devuelve la instancia de la interfaz DAO asociada.

Room simplifica muchas tareas comunes en el manejo de bases de datos, como la creación de tablas, la ejecución de consultas, la gestión de transacciones y la actualización de esquemas de base de datos. También proporciona una capa de abstracción sólida que facilita el cambio del proveedor de la base de datos subyacente (por ejemplo, de SQLite a otro).

Para utilizar Room en un proyecto Android, debes agregar las dependencias necesarias en tu archivo `build.gradle` y luego crear las entidades, DAO y la clase de base de datos según las necesidades de tu aplicación.

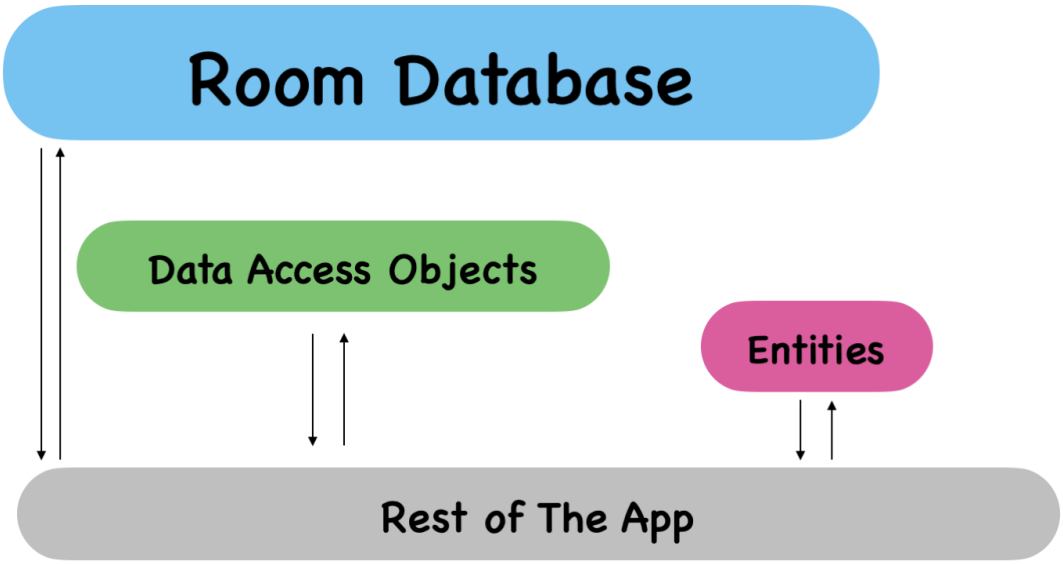
Autor/a: Sabela Sobrino Última actualización: 08.02.2024

SUBSECCIONES DE ROOM

COMPONENTES PRINCIPALES

Kotlin ofrece una forma sencilla de gestionar datos mediante la introducción de clases de datos. Puedes acceder y modificar estos datos fácilmente utilizando llamadas a funciones. Sin embargo, en el contexto de las bases de datos, la manipulación de datos requiere la interacción con tablas y consultas. Room, una biblioteca de persistencia en Android, simplifica estos flujos de trabajo, y sus principales componentes son cruciales para ello.

Los tres componentes principales de Room son los siguientes:



- **Entidades:** Estas representan las tablas de la base de datos de tu aplicación. Se utilizan para actualizar los datos almacenados en filas dentro de las tablas y para crear nuevas filas que se insertarán.
- **Objetos de Acceso a Datos (DAO):** Los DAO proporcionan métodos que tu aplicación utiliza para recuperar, actualizar, insertar y eliminar datos en la base de datos. Son una interfaz entre tu código y la lógica de persistencia.
- **Clase de Base de Datos:** Esta clase contiene la base de datos y sirve como el principal punto de acceso para la conexión subyacente a la base de datos de la aplicación. Además, la clase de base de datos proporciona instancias de los DAO asociados, facilitando la ejecución de operaciones en la base de datos de manera organizada y eficiente.

Autor/a: Sabela Sobrino Última actualización: 08.02.2024

DEPENDENCIA

En el bloque de configuración de Gradle de tu proyecto, debes incluir las bibliotecas necesarias para los componentes de Room. Aquí se muestra cómo integrar las bibliotecas de Room en tus archivos de Gradle:

Para el bloque de configuración de Gradle del proyecto:

```
plugins {
    id("com.google.devtools.ksp") version "1.8.10-1.0.9" apply false
}

// Otras configuraciones de tu proyecto...

dependencies {
    // Otras dependencias de tu proyecto...
}
```

Para el bloque de configuración de Gradle de la aplicación:

```
plugins {
    id("com.google.devtools.ksp")
}

// Otras configuraciones de tu aplicación...

dependencies {
    // Room libraries
    implementation("androidx.room:room-runtime:2.6.0")
    ksp("androidx.room:room-compiler:2.6.0")
    implementation("androidx.room:room-ktx:2.6.0")
    implementation("androidx.lifecycle:lifecycle-livedata-ktx:2.6.2")

    // Otras dependencias de tu aplicación...
}
```

Estas configuraciones aseguran que las bibliotecas de Room necesarias estén incluidas correctamente en tu proyecto o aplicación, según la naturaleza del bloque de configuración. Asegúrate de ajustar las versiones de las bibliotecas según tus requisitos específicos.

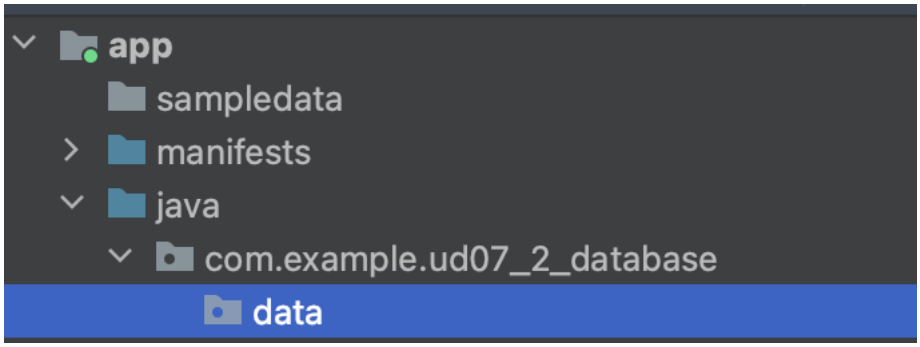
Autor/a: Sabela Sobrino Última actualización: 08.02.2024

ENTITY

La clase `Entity` sirve para definir una tabla en la base de datos, donde cada instancia de esta clase representa una fila en dicha tabla. Además, cuenta con anotaciones que le indican a Room cómo debe presentar y manipular la información en la base de datos. En el contexto de tu aplicación, esta entidad almacenará datos relacionados con los elementos del inventario, tales como el nombre, el precio y la cantidad disponible.

A continuación, se describen los pasos para implementar esta entidad llamada `Item`:

1. Crea un paquete llamado `data` dentro del paquete base de tu aplicación:



1. Dentro del paquete `data`, crea una nueva clase Kotlin llamada `Item`, la cual representará una entidad en tu base de datos.

2. Actualiza la definición de la clase `Item` con el siguiente código, declarando `id` como Int, `itemName` como String, `itemPrice` como Double, y `quantityInStock` como Int, asignando valores predeterminados según sea necesario:

```
class Item {
    val id: Int = 0
    val itemName: String = ""
    val itemPrice: Double = 0.0
    val quantityInStock: Int = 0
}
```

8. Convierte la clase `Item` en una clase de datos añadiendo la palabra clave `data` al principio de su definición:

```
data class Item(
    val id: Int = 0,
    val itemName: String,
    val itemPrice: Double,
    val quantityInStock: Int
)
```

14. Anota la clase de datos con `@Entity` sobre la declaración de la clase. Utiliza el argumento `tableName` para designar “item” como el nombre de la tabla en SQLite:

```
@Entity(tableName = "item")
data class Item(
    val id: Int = 0,
    val itemName: String,
    val itemPrice: Double,
    val quantityInStock: Int
)
```

21. Identifica `id` como la clave primaria anotándola con `@PrimaryKey`. Configura el parámetro `autoGenerate` en `true` para permitir que Room genere automáticamente el ID de cada entidad:

```
@Entity(tableName = "item")
data class Item(
    @PrimaryKey(autoGenerate = true)
    val id: Int = 0,
    val itemName: String,
    val itemPrice: Double,
    val quantityInStock: Int
)
```

29. Anota las propiedades restantes con `@ColumnInfo`. Esta anotación se utiliza para personalizar el nombre de la columna asociada con cada campo:

```
@Entity(tableName = "item")
data class Item(
    @PrimaryKey(autoGenerate = true)
    val id: Int = 0,
    @ColumnInfo(name = "name")
    val itemName: String,
    @ColumnInfo(name = "price")
    val itemPrice: Double,
    @ColumnInfo(name = "quantity")
    val quantityInStock: Int
)
```

Estos pasos establecen la estructura de la entidad `Item` para tu base de datos Room, definiendo las propiedades, la clave primaria y las personalizaciones de las columnas según sea necesario.

Autor/a: Sabela Sobrino Última actualización: 08.02.2024

DAO

Para realizar operaciones comunes en la base de datos, Room ofrece convenientes anotaciones como @Insert, @Delete y @Update. Para casos más específicos, se utiliza la anotación @Query, que permite escribir consultas compatibles con SQLite.

Ahora, procede a implementar el Data Access Object (DAO) en tu aplicación. Sigue estos pasos:

En el paquete `data`, crea la clase Kotlin `ItemDao.kt`. Transforma la definición de la clase en una interfaz y anótala con @Dao para indicar que es un DAO.

```
@Dao
interface ItemDao {
}
```

Introduce las funciones que realizarán las operaciones en la base de datos. Usa la anotación @Query para las consultas personalizadas. Aquí tienes un ejemplo:

```
@Dao
interface ItemDao {

    @Query("SELECT * from item ORDER BY name ASC")
    fun getItems(): Flow<List<Item>>

    @Query("SELECT * from item WHERE id = :id")
    fun getItem(id: Int): Flow<Item>

    // Especifica la estrategia de conflicto como IGNORE. Cuando el usuario intenta agregar un
    // Item existente a la base de datos, Room ignora el conflicto.
    @Insert(onConflict = OnConflictStrategy.IGNORE)
    suspend fun insert(item: Item)

    @Update
    suspend fun update(item: Item)

    @Delete
    suspend fun delete(item: Item)
}
```

En estas funciones:

- `getItems`: Recupera todos los elementos ordenados por nombre en forma de flujo (`Flow`).
- `getItem`: Recupera un elemento específico por su ID en forma de flujo (`Flow`).
- `insert`: Inserta un nuevo elemento, ignorando conflictos si el elemento ya existe.

- `update`: Actualiza la información de un elemento existente.
- `delete`: Elimina un elemento de la base de datos.

La utilización de `suspend` indica que estas funciones pueden pausar su ejecución y reanudarse más tarde, lo cual es esencial para operaciones en la base de datos que pueden bloquear el hilo principal.

Autor/a: Sabela Sobrino Última actualización: 08.02.2024

INSTANCIA BASE DE DATOS

En el paquete `data`, crea la clase Kotlin `InventoryDatabase.kt`. Luego, en el archivo `InventoryDatabase.kt`, convierte la clase `InventoryDatabase` en una clase abstracta que extienda `RoomDatabase` y anótala con `@Database`. A pesar de la presencia de un error de parámetros faltantes, se corregirá en el siguiente paso:

```
@Database(entities = [Item::class], version = 1, exportSchema = false)
abstract class InventoryDatabase : RoomDatabase() {

    abstract fun itemDao(): ItemDao

    companion object {
        @Volatile
        private var instance: InventoryDatabase? = null

        fun getDatabase(context: Context): InventoryDatabase {
            // Aún no se ha implementado
        }
    }
}
```

La anotación `@Database` requiere varios argumentos para que Room pueda compilar la base de datos:

- Especifica `Item` como la única clase con la lista de entidades.
- Establece `version` en 1. Debes incrementar el número de versión cada vez que cambies el esquema de la tabla de la base de datos.
- Establece `exportSchema` como false para que no se conserven copias de seguridad del historial de versiones de esquemas.

Dentro del cuerpo de la clase, se declara una función abstracta que proporciona el `ItemDao`, permitiendo que la base de datos conozca el DAO asociado con la entidad `Item`.

El objeto `companion` se utiliza para acceder a métodos para crear u obtener la base de datos, utilizando el nombre de la clase como calificador. Se define una variable anulable privada `instance` para la base de datos dentro del objeto `companion`, inicializándola en null. La palabra clave `volatile` se utiliza para garantizar que la instancia de la base de datos sea siempre visible para otros subprocesos de manera consistente.

Debajo de `instance`, dentro del objeto `companion`, se define un método `getDatabase()` con un parámetro `Context` necesario para el compilador de bases de datos, mostrando un tipo `InventoryDatabase`. En este punto, el método aún no está implementado.

Autor/a: Sabela Sobrino Última actualización: 08.02.2024

CODELABS

- Aspectos Básicos Jetpack Compose
- Room y Compose
- Conservar datos con Room