UAX Universidad Alfonso X el Sabio

Unidad Didáctica 3:
CÓMO GUARDAR DATOS DEL USUARIO

Programación Dirigida por Eventos

UAX Universidad Alfonso X el Sabio

ÍNDICE

Contenido

1. Introducción

¡Bienvenido a la asignatura Programación Dirigida por Eventos!

En esta unidad didáctica, nos centraremos en el "Almacenamiento de datos del usuario". Este tema es fundamental para cualquier aplicación que necesite gestionar información de forma persistente. A lo largo de esta unidad, aprenderás a utilizar diferentes métodos de almacenamiento en Android, tales como SharedPreferences, almacenamiento interno y externo, y bases de datos SQLite. También exploraremos el uso de proveedores de contenido para compartir datos entre aplicaciones.

Ten en cuenta:

Es crucial comprender cómo y cuándo utilizar cada método de almacenamiento para garantizar que los datos se manejen de manera segura y eficiente. La elección del método de almacenamiento adecuado puede afectar significativamente el rendimiento y la seguridad de tu aplicación.

Temas que se tratarán en esta unidad:

1. **Opciones de Almacenamiento en Android:** Introducción a las diferentes opciones de almacenamiento disponibles en Android y sus usos adecuados.
2. **SharedPreferences:** Uso de SharedPreferences para almacenar datos clave-valor.
3. **Almacenamiento Interno y Externo:** Comparación entre almacenamiento interno y externo, incluyendo sus ventajas y desventajas.
4. **Bases de Datos SQLite:** Configuración y uso de SQLite para gestionar datos estructurados.
5. **Proveedores de Contenido:** Cómo utilizar proveedores de contenido para compartir datos entre aplicaciones.
6. **Almacenamiento más allá de Android:** Opciones de almacenamiento en la nube y sincronización de datos.
7. **Firebase Realtime Database:** Uso de Firebase para almacenamiento y sincronización de datos en tiempo real.
8. **Conexión de Red y Copia de Seguridad en la Nube:** Cómo almacenar y recuperar datos a través de la red y realizar copias de seguridad en la nube.

Al finalizar esta unidad, deberías ser capaz de implementar diversas estrategias de almacenamiento en tus aplicaciones Android, asegurando que los datos del usuario se gestionen de manera adecuada y eficiente.

© Universidad Alfonso X el Sabio

2. Objetivos

Objetivo general:

- Proporcionar a los estudiantes las habilidades necesarias para gestionar y almacenar datos del usuario en aplicaciones Android utilizando diversas técnicas y herramientas disponibles.

Objetivos específicos:

En esta unidad se establecen tres objetivos específicos:

1. **Comprender las opciones de almacenamiento en Android:** Aprender a utilizar SharedPreferences, almacenamiento interno y externo, y bases de datos SQLite para diferentes tipos de datos.
2. **Implementar técnicas de almacenamiento seguro y eficiente:** Desarrollar la capacidad de elegir y aplicar el método de almacenamiento adecuado según las necesidades de la aplicación y los datos a gestionar.
3. **Utilizar servicios de almacenamiento en la nube:** Aplicar técnicas para almacenar y sincronizar datos utilizando Firebase y otros servicios en la nube, garantizando la disponibilidad y seguridad de los datos.

© Universidad Alfonso X el Sabio

3. Preferencias y configuración

3.1. Almacenamiento de Datos en Android

Opciones de Almacenamiento en Android

Tipos de Almacenamiento

- **Preferencias Compartidas**: Almacena datos primitivos privados en pares clave-valor.
- **Almacenamiento Interno**: Almacena datos privados en la memoria del dispositivo.
- **Almacenamiento Externo**: Almacena datos públicos en el dispositivo o en almacenamiento externo.
- **Bases de Datos SQLite**: Almacena datos estructurados en una base de datos privada.
- **Proveedores de Contenido**: Almacena datos privados y los pone a disposición pública.

Almacenamiento más allá de Android

- **Conexión de Red**: Almacenar datos en la web con tu propio servidor.
- **Copia de Seguridad en la Nube**: Realiza copias de seguridad de los datos de la aplicación y del usuario en la nube.
- **Firebase Realtime Database**: Almacena y sincroniza datos con una base de datos en la nube NoSQL en tiempo real.

Sistema de Archivos de Android

Tipos de Directorios

- **Almacenamiento Externo**: Directorios públicos.
- **Almacenamiento Interno**: Directorios privados solo para tu aplicación.

Navegación por la Estructura de Directorios

Las aplicaciones pueden navegar por la estructura de directorios similar a Linux y java.io.

Almacenamiento Interno

Características

- Siempre disponible.
- Utiliza el sistema de archivos del dispositivo.

© Universidad Alfonso X el Sabio

- Solo tu aplicación puede acceder a los archivos a menos que se configure explícitamente para ser legible o escribible.
- Al desinstalar la aplicación, el sistema elimina todos los archivos de la aplicación del almacenamiento interno.

Directorios Privados

- **Directorio de almacenamiento permanente**: getFilesDir().
- **Directorio de almacenamiento temporal**: getCacheDir().

Creación de un Archivo

java
Copiar código
File file = new File(context.getFilesDir(), filename);

Usa los operadores estándar de java.io para interactuar con los archivos.

Almacenamiento Externo

Características

- Puede ser removido.
- Utiliza el sistema de archivos del dispositivo o almacenamiento externo físico como una tarjeta SD.
- Legible para todo el mundo, cualquier aplicación puede leer.
- Al desinstalar, el sistema no elimina los archivos privados de la aplicación.

Configuración de Permisos

xml
Copiar código
<uses-permission android:name="android.permission.WRITE_EXTERNAL_STORAGE" />
<uses-permission android:name="android.permission.READ_EXTERNAL_STORAGE" />

Verificación de Disponibilidad de Almacenamiento

java
Copiar código
```
public boolean isExternalStorageWritable() {
   String state = Environment.getExternalStorageState();
   if (Environment.MEDIA_MOUNTED.equals(state)) {
      return true;
   }
   return false;
}
```


© Universidad Alfonso X el Sabio

Directorios Públicos Externos

Ejemplos de directorios públicos externos incluyen:

- DIRECTORY_ALARMS y DIRECTORY_RINGTONES: Para archivos de audio.
- DIRECTORY_DOCUMENTS: Para documentos creados por el usuario.
- DIRECTORY_DOWNLOADS: Para archivos descargados por el usuario.

Uso de Almacenamiento Interno/Externo

Cuándo Usar Almacenamiento Interno

Usa almacenamiento interno cuando desees asegurarte de que ni el usuario ni otras aplicaciones puedan acceder a tus archivos.

Cuándo Usar Almacenamiento Externo

Usa almacenamiento externo para archivos que no requieren restricciones de acceso, que desees compartir con otras aplicaciones o que permitas al usuario acceder con una computadora.

Guardar Archivos en Almacenamiento Compartido

Guarda los nuevos archivos que el usuario adquiera a través de tu aplicación en un directorio público donde otras aplicaciones puedan acceder a ellos y el usuario pueda copiarlos fácilmente desde el dispositivo.

SQLite

Características

SQLite es ideal para datos repetitivos o estructurados, como contactos. Android proporciona una base de datos similar a SQL que se cubre en capítulos y prácticas posteriores.

Ejemplos de SQLite

- **Introducción a Bases de Datos SQLite**
- **Almacenamiento de Datos con SQLite**
- **Búsqueda en una Base de Datos SQLite**

© Universidad Alfonso X el Sabio

Shared Preferences

Características

Permite leer y escribir pequeñas cantidades de datos primitivos como pares clave-valor en un archivo en el almacenamiento del dispositivo. Se cubre en capítulos y prácticas posteriores.

Uso de Shared Preferences

Almacena y recupera datos pequeños y simples, como configuraciones de usuario o estados de la aplicación.

Otras Opciones de Almacenamiento

Firebase

Utiliza Firebase para almacenar y compartir datos. Sincroniza datos con la base de datos en la nube Firebase y mantiene los datos disponibles cuando tu aplicación está sin conexión.

Firebase Realtime Database

- **Aplicaciones Conectadas**: Comparten datos.
- **Alojado en la Nube**: Los datos se almacenan como JSON y se sincronizan en tiempo real con todos los clientes conectados.

Conexión de Red

Puedes usar la red (cuando esté disponible) para almacenar y recuperar datos en tus propios servicios web. Utiliza clases en los paquetes java.net.* y android.net.*.

Copia de Seguridad de Datos

- **Copia de Seguridad Automática**: Para Android 6.0 (API nivel 23) y superior. Realiza automáticamente copias de seguridad de los datos de la aplicación en la nube sin necesidad de código y es gratuito.
- **API de Copia de Seguridad**: Para Android 5.1 (API nivel 22). Regístrate en el Servicio de Copia de Seguridad de Android para obtener una clave de servicio, configura el manifiesto para usar el servicio de copia de seguridad y crea un agente de copia de seguridad extendiendo la clase BackupAgentHelper.

3.2. Shared Preferences en Android

© Universidad Alfonso X el Sabio

¿Qué son las Shared Preferences?

Definición

Shared Preferences te permite leer y escribir pequeñas cantidades de datos primitivos como pares clave-valor en un archivo en el almacenamiento del dispositivo. La clase SharedPreferences proporciona APIs para leer, escribir y gestionar estos datos.

Uso Común

Guardar datos en onPause() y restaurarlos en onCreate(), permitiendo que los datos persistan entre sesiones de usuario, incluso si la aplicación se cierra o el dispositivo se reinicia.

Shared Preferences vs. Saved Instance State

Persistencia de Datos

- **Shared Preferences**: Los datos persisten a través de sesiones de usuario, incluso si la aplicación se cierra o el dispositivo se reinicia. Se usa comúnmente para almacenar preferencias de usuario, como configuraciones o puntuaciones de juegos.
- **Saved Instance State**: Los datos persisten a través de cambios de configuración en la misma sesión de usuario, como la rotación del dispositivo. Se usa comúnmente para recrear el estado de la actividad después de un cambio de configuración.

Ejemplos

- **Shared Preferences**: Recordar la configuración preferida del usuario.
- **Saved Instance State**: Mantener la pestaña actualmente seleccionada en una interfaz de usuario.

Creación de Shared Preferences

Archivo de Shared Preferences

Solo necesitas un archivo de Shared Preferences por aplicación. Nómbralo con el nombre del paquete de tu aplicación para que sea único y fácil de asociar.

Ejemplo de Creación

```
private String sharedPrefFile = "com.example.android.hellosharedprefs";
mPreferences = getSharedPreferences(sharedPrefFile, MODE_PRIVATE);
```


© Universidad Alfonso X el Sabio

Modo de Acceso

El argumento MODE para getSharedPreferences() es para compatibilidad con versiones anteriores. Usa solo MODE_PRIVATE para mayor seguridad.

Guardar y Restaurar Datos con Shared Preferences

Guardar Datos

Usa la interfaz SharedPreferences.Editor para guardar datos. Los métodos put sobrescriben si la clave existe y apply() guarda de manera asíncrona y segura.

Ejemplo de Guardar Datos

```
@Override
protected void onPause() {
   super.onPause();
   SharedPreferences.Editor preferencesEditor = mPreferences.edit();
   preferencesEditor.putInt("count", mCount);
   preferencesEditor.putInt("color", mCurrentColor);
   preferencesEditor.apply();
}
```

Restaurar Datos

Restaura los datos en onCreate() de la actividad. Los métodos get toman dos argumentos: la clave y el valor predeterminado si no se encuentra la clave.

Ejemplo de Restaurar Datos

```
@Override
protected void onCreate(Bundle savedInstanceState) {
   super.onCreate(savedInstanceState);
   setContentView(R.layout.activity_main);

   mPreferences = getSharedPreferences(sharedPrefFile, MODE_PRIVATE);

   if (savedInstanceState != null) {
      mCount = mPreferences.getInt("count", 0);
      mShowCount.setText(String.format("%s", mCount));

      mCurrentColor = mPreferences.getInt("color", mCurrentColor);
      mShowCount.setBackgroundColor(mCurrentColor);
```


© Universidad Alfonso X el Sabio

```
        mNewText = mPreferences.getString("text", "");
    } else {
        // Inicialización si no hay estado guardado
    }
}
```

Borrar Datos de Shared Preferences

Borrar Datos

Llama a clear() en SharedPreferences.Editor y aplica los cambios. Puedes combinar llamadas a put y clear, pero clear() siempre se realiza primero, independientemente del orden.

Ejemplo de Borrar Datos

```
SharedPreferences.Editor preferencesEditor = mPreferences.edit();
preferencesEditor.clear();
preferencesEditor.apply();
```

Escuchar Cambios en Shared Preferences

Escuchar Cambios

Implementa la interfaz SharedPreference.OnSharedPreferenceChangeListener y registra el listener con registerOnSharedPreferenceChangeListener() en onResume() y onPause().

Ejemplo de Implementación

```
public class SettingsActivity extends AppCompatActivity implements SharedPreferences.OnSharedPreferenceChangeListener {

    @Override
    protected void onResume() {
        super.onResume();
        mPreferences.registerOnSharedPreferenceChangeListener(this);
    }

    @Override
    protected void onPause() {
        super.onPause();
        mPreferences.unregisterOnSharedPreferenceChangeListener(this);
    }

    @Override
```


© Universidad Alfonso X el Sabio

```
    public void onSharedPreferenceChanged(SharedPreferences sharedPreferences, String key) {
        if (key.equals(MY_KEY)) {
            // Realiza la acción necesaria cuando cambia la preferencia
        }
    }
}
```

3.3. Configuración de la Aplicación en Android

¿Qué son las Configuraciones de la Aplicación?

Definición

Las configuraciones de la aplicación permiten a los usuarios ajustar las características y comportamientos de la aplicación. Ejemplos incluyen:

- Ubicación predeterminada del hogar
- Unidades de medida
- Comportamiento de notificaciones específicas

Uso Común

Las configuraciones son ideales para valores que cambian con poca frecuencia y son relevantes para la mayoría de los usuarios. Si los valores cambian con frecuencia, es mejor usar el menú de opciones o el cajón de navegación.

Pantallas de Configuración

Acceso a Configuraciones

Los usuarios acceden a las configuraciones a través de:

- Cajón de navegación
- Menú de opciones

Organización de Configuraciones

- **Número manejable de opciones**: 7 o menos, organizadas por prioridad.
- **Agrupar configuraciones relacionadas**: 7-15 configuraciones agrupadas bajo divisores de sección.

© Universidad Alfonso X el Sabio

- **Más de 16 configuraciones**: Agrupar en pantallas abiertas desde la pantalla principal de configuraciones.

Preferencias vs. Vistas

Usa objetos Preference en lugar de objetos View en tus pantallas de configuración. Diseña y edita objetos Preference en el editor de diseño al igual que los objetos View.

Definir Configuraciones en una Pantalla de Preferencias

Define las configuraciones en una pantalla de preferencias similar a un diseño en el archivo res/xml/preferences.xml.

Ejemplo de Pantalla de Preferencias

```
<PreferenceScreen>
   <PreferenceCategory android:title="Flight Preferences">
      <CheckBoxPreference android:title="Wake for meals" ... />
      <EditTextPreference android:title="Favorite city" ... />
   </PreferenceCategory>
</PreferenceScreen>
```

Implementación de Configuraciones

Clases de Preferencia

La clase Preference proporciona la vista para cada tipo de configuración y asocia la vista con la interfaz SharedPreferences para almacenar y recuperar los datos de preferencia. Usa la clave en la Preference para almacenar el valor de la configuración.

Subclases de Preferencia

- CheckBoxPreference
- ListPreference
- SwitchPreference
- EditTextPreference
- RingtonePreference

Clases para Agrupar

- PreferenceScreen: Raíz de una jerarquía de diseño de preferencias.
- PreferenceGroup: Agrupa configuraciones (Preference objects).
- PreferenceCategory: Título encima de un grupo como divisor de sección.

© Universidad Alfonso X el Sabio

Fragmentos de Configuración

Usa un Activity con un Fragment para mostrar la pantalla de configuraciones. Usa subclases especializadas de Activity y Fragment que manejan el trabajo de guardar configuraciones.

Actividades y Fragmentos para Configuraciones

- **Android 3.0 y superior**: AppCompatActivity con PreferenceFragmentCompat o Activity con PreferenceFragment.
- **Android anterior a 3.0**: Extender la clase PreferenceActivity.

Pasos para Implementar Configuraciones

1. Crear la pantalla de preferencias.
2. Crear una Activity para las configuraciones.
3. Crear un Fragment para las configuraciones.
4. Añadir el tema de preferencias a AppTheme.
5. Añadir código para invocar la interfaz de usuario de configuraciones.

Ejemplo de Actividad de Configuraciones

```
public class MySettingsActivity extends AppCompatActivity {
   @Override
   protected void onCreate(Bundle savedInstanceState) {
      super.onCreate(savedInstanceState);
      getSupportFragmentManager().beginTransaction()
         .replace(android.R.id.content, new MySettingsFragment())
         .commit();
   }
}
```

Ejemplo de Fragmento de Configuraciones

```
public class MySettingsFragment extends PreferenceFragmentCompat {
   @Override
   public void onCreatePreferences(Bundle savedInstanceState, String rootKey) {
      setPreferencesFromResource(R.xml.preferences, rootKey);
   }
}
```

Añadir PreferenceTheme al Tema de la Aplicación

```
<style name="AppTheme" parent="...">
   ...
   <item name="preferenceTheme">@style/PreferenceThemeOverlay</item>
```


© Universidad Alfonso X el Sabio

</style>

Configuraciones Predeterminadas

Establecer Valores Predeterminados

Establece el valor predeterminado que la mayoría de los usuarios elegiría.

- Contactos
- Usar menos batería
- Bluetooth apagado hasta que el usuario lo encienda
- Menor riesgo para la seguridad y pérdida de datos

Guardar Valores Predeterminados en Shared Preferences

En onCreate() de MainActivity:

PreferenceManager.setDefaultValues(this, R.xml.preferences, false);

Guardar y Recuperar Configuraciones

Guardar Valores de Configuración

No necesitas escribir código para guardar configuraciones. Si usas PreferenceActivity y Fragment, Android guarda automáticamente los valores de configuración en SharedPreferences.

Recuperar Configuraciones de Shared Preferences

En tu código, obtén configuraciones desde las preferencias compartidas predeterminadas. Usa la clave especificada en la vista de preferencia en XML.

Ejemplo de Recuperación

```
SharedPreferences sharedPref = PreferenceManager.getDefaultSharedPreferences(this);
String destinationPref = sharedPref.getString("fav_city", "Jamaica");
```

Responder a Cambios en Configuraciones

Escuchar Cambios

Define setOnPreferenceChangeListener() en onCreatePreferences() en el fragmento de configuraciones.

Ejemplo de Listener

© Universidad Alfonso X el Sabio

```
@Override
public void onCreatePreferences(Bundle savedInstanceState, String rootKey) {
   setPreferencesFromResource(R.xml.preferences, rootKey);
   ListPreference colorPref = (ListPreference) findPreference("color_pref");
   colorPref.setOnPreferenceChangeListener(new Preference.OnPreferenceChangeListener() {
      @Override
      public boolean onPreferenceChange(Preference preference, Object newValue) {
         setMyBackgroundColor(newValue);
         return true;
      }
   });
}
```

Resúmenes para Configuraciones

Resúmenes Condicionales

Para preferencias que tienen valores true/false, establece atributos para definir resúmenes condicionales.

Resúmenes para Otros Valores

Para configuraciones con valores distintos de true/false, actualiza el resumen cuando cambie el valor de la configuración. Establece el resumen en onPreferenceChangeListener().

Ejemplo de Resumen

```
EditTextPreference cityPref = (EditTextPreference) findPreference("fav_city");
cityPref.setOnPreferenceChangeListener(new Preference.OnPreferenceChangeListener() {
   @Override
   public boolean onPreferenceChange(Preference pref, Object value) {
      String city = value.toString();
      pref.setSummary("Your favorite city is " + city);
      return true;
   }
});
```

Plantilla de la Actividad de Configuración

Configuraciones Complejas

Para configuraciones más complejas, usa la plantilla de Settings Activity.

© Universidad Alfonso X el Sabio

4. Cómo almacenar datos con Room

4.1. SQLite en Android

Base de Datos SQLite

Introducción a SQLite

SQLite es una biblioteca de software que implementa un motor de base de datos SQL que es:

- **Autónoma**: No requiere otros componentes.
- **Sin servidor**: No necesita un backend de servidor.
- **Sin configuración**: No necesita ser configurada para tu aplicación.
- **Transaccional**: Los cambios dentro de una transacción en SQLite ocurren completamente o no ocurren en absoluto.

Características de SQLite

- **Tablas**: Almacenan datos en filas y columnas, similar a una hoja de cálculo.
- **Campos**: Intersección de una fila y una columna que contiene datos o referencias.
- **Filas**: Identificadas por IDs únicos.
- **Nombres de columnas**: Únicos por tabla.

Ejemplo de Tabla

_id	word	definition
1	alpha	first letter
2	beta	second letter
3	alpha	particle

Consultas SQL

Operaciones Básicas en SQL

- **Insertar filas**: Añadir nuevos registros a una tabla.
- **Eliminar filas**: Quitar registros existentes de una tabla.
- **Actualizar valores en filas**: Modificar datos de registros existentes.
- **Recuperar filas que cumplen ciertos criterios**: Seleccionar registros basados en condiciones.

© Universidad Alfonso X el Sabio

Ejemplo de Consulta SQL

SELECT word, description FROM WORD_LIST_TABLE WHERE word="alpha"

Transacciones en SQLite

¿Qué es una Transacción?

Una transacción es una secuencia de operaciones realizadas como una única unidad lógica de trabajo. Debe tener cuatro propiedades conocidas como ACID:

- **Atomicidad**: Todas las modificaciones se realizan o no se realizan.
- **Consistencia**: Al completar la transacción, todos los datos están en un estado consistente.
- **Aislamiento**: Las modificaciones hechas por transacciones concurrentes deben estar aisladas entre sí.
- **Durabilidad**: Después de completar una transacción, sus efectos son permanentes en el sistema.

Todo o Nada

Todos los cambios dentro de una transacción en SQLite ocurren completamente o no ocurren en absoluto, incluso si la escritura en el disco es interrumpida por un fallo del programa, un fallo del sistema operativo o una falla de energía.

Operaciones Básicas en SQL

Consultas Básicas

- **SELECT**: Seleccionar columnas para devolver.
- **FROM**: Especificar la tabla de la cual obtener resultados.
- **WHERE**: Condiciones que deben cumplirse.

Ejemplo de Uso de Operadores

SELECT _id FROM WORD_LIST_TABLE WHERE word="alpha" AND definition LIKE "%art%" ORDER BY word DESC LIMIT 1

Ejemplos de Consultas

1. Obtener toda la tabla:

SELECT * FROM WORD_LIST_TABLE

2. Seleccionar palabra y definición donde _id > 2:

© Universidad Alfonso X el Sabio

SELECT word, definition FROM WORD_LIST_TABLE WHERE _id > 2

3. Obtener _id de palabra "alpha" con "art" en la definición:

SELECT _id FROM WORD_LIST_TABLE WHERE word="alpha" AND definition LIKE "%art%"

Consultas SQL con rawQuery() y query()

Método rawQuery()

Permite ejecutar consultas SQL crudas con parámetros opcionales.

Ejemplo de rawQuery()

```
String query = "SELECT * FROM WORD_LIST_TABLE";
Cursor cursor = db.rawQuery(query, null);

String query = "SELECT word, definition FROM WORD_LIST_TABLE WHERE _id > ?";
String[] selectionArgs = new String[]{"2"};
Cursor cursor = db.rawQuery(query, selectionArgs);
```

Método query()

Proporciona una forma estructurada de construir consultas SQL.

Ejemplo de query()

```
String table = "WORD_LIST_TABLE";
String[] columns = new String[]{"*"};
String selection = "word = ?";
String[] selectionArgs = new String[]{"alpha"};
String groupBy = null;
String having = null;
String orderBy = "word ASC";
String limit = "21";

Cursor cursor = db.query(table, columns, selection, selectionArgs, groupBy, having, orderBy, limit);
```

© Universidad Alfonso X el Sabio

Cursors

¿Qué es un Cursor?

Las consultas siempre devuelven un objeto Cursor. Un Cursor es una interfaz que proporciona acceso de lectura y escritura aleatoria al conjunto de resultados devuelto por una consulta de base de datos. Piensa en él como un puntero a filas de tabla.

Uso de Cursors

- Navegar a través de filas.
- Leer datos de columnas.

4.2. Room, LiveData y ViewModel en Android

Room

Descripción General

Room es una biblioteca robusta de mapeo de objetos SQL que genera código SQLite Android y proporciona una API simple para tu base de datos.

Componentes de Room

1. **Entity**: Define el esquema de la tabla de la base de datos.
2. **DAO (Data Access Object)**: Define las operaciones de lectura/escritura para la base de datos.
3. **RoomDatabase**: Contenedor de la base de datos. Utilizado para crear o conectar a la base de datos.

Ejemplo de Entity

```
@Entity(tableName = "word_table")
public class Word {
   @PrimaryKey(autoGenerate = true)
   private int id;

   @NonNull
   @ColumnInfo(name = "word")
   private String word;

   // Getters y setters...
}
```


© Universidad Alfonso X el Sabio

Ejemplo de DAO

```
@Dao
public interface WordDao {
   @Insert
   void insert(Word word);

   @Query("DELETE FROM word_table")
   void deleteAll();

   @Query("SELECT * from word_table ORDER BY word ASC")
   LiveData<List<Word>> getAllWords();
}
```

Ejemplo de RoomDatabase

```
@Database(entities = {Word.class}, version = 1, exportSchema = false)
public abstract class WordRoomDatabase extends RoomDatabase {
   public abstract WordDao wordDao();

   private static volatile WordRoomDatabase INSTANCE;

   static WordRoomDatabase getDatabase(final Context context) {
      if (INSTANCE == null) {
         synchronized (WordRoomDatabase.class) {
            if (INSTANCE == null) {
               INSTANCE = Room.databaseBuilder(context.getApplicationContext(),
                     WordRoomDatabase.class, "word_database")
                     .fallbackToDestructiveMigration()
                     .build();
            }
         }
      }
      return INSTANCE;
   }
}
```

LiveData

Descripción General

LiveData es una clase de contenedor de datos que es consciente de los eventos del ciclo de vida. Mantiene un valor y permite que este valor sea observado.

© Universidad Alfonso X el Sabio

Beneficios

- Notifica a los observadores cuando los datos cambian.
- Es consciente del ciclo de vida: sabe cuándo el dispositivo rota o la aplicación se detiene.

Ejemplo de Uso

```
@Query("SELECT * from word_table ORDER BY word ASC")
LiveData<List<Word>> getAllWords();
```

Observando LiveData

```
java
Copiar código
wordViewModel.getAllWords().observe(this, new Observer<List<Word>>() {
   @Override
   public void onChanged(@Nullable final List<Word> words) {
      // Actualiza la UI aquí
   }
});
```

ViewModel

Descripción General

Los ViewModels son objetos que proporcionan datos para los componentes de UI y sobreviven a los cambios de configuración.

Beneficios

- Proporciona datos a la UI
- Sobrevive a los cambios de configuración
- Puede compartir datos entre fragmentos

Ejemplo de ViewModel

```
public class WordViewModel extends AndroidViewModel {
   private WordRepository mRepository;
   private LiveData<List<Word>> mAllWords;

   public WordViewModel(Application application) {
      super(application);
      mRepository = new WordRepository(application);
      mAllWords = mRepository.getAllWords();
   }
```


© Universidad Alfonso X el Sabio

```
    LiveData<List<Word>> getAllWords() {
        return mAllWords;
    }

    public void insert(Word word) {
        mRepository.insert(word);
    }
}
```

Repositorio

Descripción General

Un repositorio maneja las operaciones de datos. Proporciona una API limpia para que la UI interactúe con los datos.

Beneficios

- Maneja operaciones de datos en segundo plano.
- Puede administrar múltiples backends.

Ejemplo de Repositorio

```
public class WordRepository {
    private WordDao mWordDao;
    private LiveData<List<Word>> mAllWords;

    WordRepository(Application application) {
        WordRoomDatabase db = WordRoomDatabase.getDatabase(application);
        mWordDao = db.wordDao();
        mAllWords = mWordDao.getAllWords();
    }

    LiveData<List<Word>> getAllWords() {
        return mAllWords;
    }

    public void insert(Word word) {
        new insertAsyncTask(mWordDao).execute(word);
    }

    private static class insertAsyncTask extends AsyncTask<Word, Void, Void> {
        private WordDao mAsyncTaskDao;
```


© Universidad Alfonso X el Sabio

```
        insertAsyncTask(WordDao dao) {
            mAsyncTaskDao = dao;
        }

        @Override
        protected Void doInBackground(final Word... params) {
            mAsyncTaskDao.insert(params[0]);
            return null;
        }
    }
}
```

Lifecycle-aware Components

Descripción General

Los componentes conscientes del ciclo de vida realizan acciones en respuesta a un cambio en el estado del ciclo de vida de otro componente.

Beneficios

- Permiten una gestión más eficiente y clara de los recursos.
- Evitan fugas de memoria.

Ejemplo de Implementación

```
public class MyObserver implements LifecycleObserver {
    @OnLifecycleEvent(Lifecycle.Event.ON_START)
    public void start() {
        // Acción a realizar cuando el evento ON_START ocurre
    }

    @OnLifecycleEvent(Lifecycle.Event.ON_STOP)
    public void stop() {
        // Acción a realizar cuando el evento ON_STOP ocurre
    }
}
```

Ejemplo de Implementación

Aplicación RoomWordsSample

La aplicación RoomWordsSample implementa esta arquitectura.

Estructura de la Aplicación

© Universidad Alfonso X el Sabio

1. **MainActivity**: Controlador de la UI.
2. **WordRepository**: Fuente de datos.
3. **WordRoomDatabase**: Base de datos.
4. **WordDao**: Objeto de acceso a datos.
5. **WordViewModel**: Proporciona datos a la UI.
6. **LiveData<List<Word>>**: Lista observable de palabras.
7. **WordListAdapter**: Adaptador de RecyclerView.
8. **Observer**: Observador de cambios en los datos.

© Universidad Alfonso X el Sabio

5. Conclusiones

En esta tercera unidad didáctica, hemos aprendido sobre las distintas opciones de almacenamiento de datos en Android, lo cual es fundamental para cualquier aplicación que necesite gestionar información de manera persistente. El manejo eficiente de datos del usuario no solo mejora la experiencia del usuario, sino que también asegura la integridad y seguridad de la información almacenada.

Las principales técnicas abordadas incluyen **SharedPreferences**, ideal para almacenar pequeños conjuntos de datos clave-valor, y los métodos de **almacenamiento interno** y **externo**, cada uno con sus ventajas y limitaciones según las necesidades de acceso y seguridad. También se discutió la implementación de **bases de datos SQLite**, que permiten la gestión de datos estructurados de forma más robusta y escalable. Finalmente, se exploraron los **proveedores de contenido**, que facilitan el intercambio de datos entre aplicaciones, y el uso de herramientas de almacenamiento en la nube como **Firebase**, que provee sincronización de datos en tiempo real.

En conclusión, esta unidad ha proporcionado una visión completa de las opciones disponibles para almacenar datos en aplicaciones Android, cada una adecuada para diferentes situaciones. La correcta selección y uso de estas técnicas contribuye significativamente a la estabilidad, seguridad y funcionalidad de una aplicación, siendo un aspecto crucial en el desarrollo de aplicaciones móviles eficientes y seguras.

© Universidad Alfonso X el Sabio

6. Bibliografía

Android Developers. (2023). Data Storage in Android. Google. Disponible en: https://developer.android.com/training/data-storage

Phillips, B., Stewart, C., Hardy, K., & Marsicano, B. (2019). Android Programming: The Big Nerd Ranch Guide. Big Nerd Ranch.

Nagpal, S. (2018). Android Data Persistence: Saving Data Locally with SQLite, SharedPreferences, and More. Apress.

Steele, J. (2014). The Android Developer's Cookbook: Building Applications with the Android SDK. Addison-Wesley.

Firebase. (2023). Firebase Realtime Database Documentation. Google. Disponible en: https://firebase.google.com/docs/database

Estas fuentes ofrecen un marco teórico sólido y ejemplos prácticos sobre cómo implementar estrategias de almacenamiento de datos en Android, facilitando el desarrollo de aplicaciones que gestionen de manera eficaz la información de los usuarios.

© Universidad Alfonso X el Sabio

UAX Universidad Alfonso X el Sabio

G R A C I A S

UAX.COM