**Documentación para Implementar AutocompleteManager en Proyectos Android**

**1. Introducción**

AutocompleteManager es una clase auxiliar diseñada para simplificar la carga y gestión de datos en AutoCompleteTextView en una aplicación Android. Esta clase permite la carga opcional de diferentes tipos de datos, como empresas, equipos u obras, y actualiza los autocompletes en tiempo real a medida que los datos se obtienen de la API.

**2. Estructura de Archivos**

* **Directorio utils**: AutocompleteManager se coloca en este directorio junto con otras utilidades del proyecto.

css

Copiar código

+---utils

| | AutocompleteManager.kt

| | Constant.kt

| | Event.kt

| | SharedPreferencesHelper.kt

| | ToastCustom.kt

**3. Implementación de AutocompleteManager**

**3.1. Clase AutocompleteManager**

**Ubicación:** utils/AutocompleteManager.kt

kotlin

Copiar código

package com.example.gestionequipos.utils

import android.content.Context

import android.widget.ArrayAdapter

import android.widget.AutoCompleteTextView

import androidx.lifecycle.LifecycleOwner

import androidx.lifecycle.Observer

import com.example.gestionequipos.viewmodel.AppDataViewModel

class AutocompleteManager(private val context: Context, private val viewModel: AppDataViewModel) {

fun loadEmpresas(autoCompleteTextView: AutoCompleteTextView? = null, lifecycleOwner: LifecycleOwner? = null) {

viewModel.cargarEmpresas()

if (autoCompleteTextView != null && lifecycleOwner != null) {

viewModel.empresas.observe(lifecycleOwner, Observer { empresas ->

val adapter = ArrayAdapter(context, android.R.layout.simple\_dropdown\_item\_1line, empresas.map { it.nombre })

autoCompleteTextView.setAdapter(adapter)

})

}

}

fun loadEquipos(autoCompleteTextView: AutoCompleteTextView? = null, lifecycleOwner: LifecycleOwner? = null) {

viewModel.cargarEquipos()

if (autoCompleteTextView != null && lifecycleOwner != null) {

viewModel.equipos.observe(lifecycleOwner, Observer { equipos ->

val adapter = ArrayAdapter(context, android.R.layout.simple\_dropdown\_item\_1line, equipos.map { "${it.interno} - ${it.descripcion}" })

autoCompleteTextView.setAdapter(adapter)

})

}

}

fun loadObras(autoCompleteTextView: AutoCompleteTextView? = null, lifecycleOwner: LifecycleOwner? = null) {

viewModel.cargarObras()

if (autoCompleteTextView != null && lifecycleOwner != null) {

viewModel.obras.observe(lifecycleOwner, Observer { obras ->

val adapter = ArrayAdapter(context, android.R.layout.simple\_dropdown\_item\_1line, obras.map { "${it.centro\_costo} - ${it.nombre}" })

autoCompleteTextView.setAdapter(adapter)

})

}

}

}

**3.2. Métodos Principales**

* **loadEmpresas**: Carga los datos de empresas y los asigna a un AutoCompleteTextView.
* **loadEquipos**: Carga los datos de equipos y los asigna a un AutoCompleteTextView.
* **loadObras**: Carga los datos de obras y los asigna a un AutoCompleteTextView.

Cada método acepta:

* **AutoCompleteTextView** (opcional): El componente UI donde se mostrarán los datos.
* **LifecycleOwner** (opcional): Generalmente la actividad o fragmento actual, que controla el ciclo de vida del LiveData.

**4. Uso en una Actividad o Fragmento**

**4.1. Ejemplo en LoginActivity**

**Código:**

kotlin

Copiar código

class LoginActivity : AppCompatActivity() {

private lateinit var autocompleteManager: AutocompleteManager

private lateinit var appDataViewModel: AppDataViewModel

private lateinit var binding: ActivityLoginBinding

override fun onCreate(savedInstanceState: Bundle?) {

super.onCreate(savedInstanceState)

binding = ActivityLoginBinding.inflate(layoutInflater)

setContentView(binding.root)

// Inicializar ViewModel

appDataViewModel = ViewModelProvider(this).get(AppDataViewModel::class.java)

// Inicializar AutocompleteManager

autocompleteManager = AutocompleteManager(this, appDataViewModel)

// Cargar datos y configurar el AutoCompleteTextView de empresas

autocompleteManager.loadEmpresas(binding.empresaAutocomplete, this)

}

}

**4.2. Ejemplo en ParteDiarioFragment**

**Código:**

kotlin

Copiar código

class ParteDiarioFragment : Fragment() {

private var \_binding: FragmentParteDiarioBinding? = null

private val binding get() = \_binding!!

private lateinit var autocompleteManager: AutocompleteManager

private val appDataViewModel: AppDataViewModel by activityViewModels()

override fun onCreateView(

inflater: LayoutInflater, container: ViewGroup?,

savedInstanceState: Bundle?

): View {

\_binding = FragmentParteDiarioBinding.inflate(inflater, container, false)

return binding.root

}

override fun onViewCreated(view: View, savedInstanceState: Bundle?) {

super.onViewCreated(view, savedInstanceState)

// Inicializar AutocompleteManager

autocompleteManager = AutocompleteManager(requireContext(), appDataViewModel)

// Configurar los AutoCompleteTextView con AutocompleteManager

autocompleteManager.loadEquipos(binding.equipoAutocomplete, viewLifecycleOwner)

autocompleteManager.loadObras(binding.obraAutocomplete, viewLifecycleOwner)

// Otras configuraciones del fragmento

setupRecyclerView()

setupFab()

setupTextWatchers()

setupListeners()

actualizarHistorialPartes()

}

override fun onDestroyView() {

super.onDestroyView()

\_binding = null

}

}

**4.3. Práctica Recomendada para el Uso de ViewBinding en Fragmentos**

**4.3. Práctica Recomendada para el Uso de ViewBinding en Fragmentos**

* **En Fragmentos:** Es buena práctica utilizar private var \_binding: FragmentParteDiarioBinding? = null y private val binding get() = \_binding!! para manejar el ViewBinding. Esto asegura que la referencia a la vista se libere cuando la vista del fragmento se destruye, ayudando a evitar fugas de memoria.
  + **\_binding**: Se usa para almacenar una referencia nula cuando la vista del fragmento se destruye (onDestroyView()), lo que previene el acceso a una vista que ya no existe.
  + **binding**: Es una propiedad que ofrece un acceso seguro a la vista del fragmento mientras está activa. El operador !! asegura que se lance una excepción si se intenta acceder a la vista cuando ya se ha destruido, ayudando a detectar errores en tiempo de desarrollo.
* **En Actividades:** Es común utilizar lateinit var binding: ActivityLoginBinding, ya que el ciclo de vida de una actividad es más simple. La vista se crea en onCreate y se destruye cuando la actividad se destruye, lo que hace que lateinit sea una opción segura y directa.

**5. Ventajas del Enfoque**

* **Modularidad:** Permite mantener el código bien organizado y fácilmente extensible para nuevos tipos de datos.
* **Flexibilidad:** Los métodos loadEmpresas, loadEquipos, y loadObras pueden invocarse según sea necesario, evitando cargar datos innecesarios.
* **Reutilización:** El mismo AutocompleteManager puede ser usado en diferentes actividades y fragmentos sin duplicar código.
* **Seguridad en la Memoria:** El uso de ViewBinding con private var \_binding en fragmentos asegura que la memoria se gestione correctamente, previniendo posibles fugas de memoria.

**6. Conclusión**

Con esta configuración, puedes implementar fácilmente la carga y gestión de autocompletes en cualquier actividad o fragmento de tu aplicación Android, manteniendo el código limpio, eficiente y fácil de mantener. Además, el manejo recomendado de ViewBinding en fragmentos contribuye a una mejor gestión de recursos y seguridad en el uso de la memoria.