TECNOLOGICO NACIONAL DE MEXICO

INSTITUTO TECNOLOGICO DE CIUDAD MADERO

Carrera: Ingeniería en Sistemas Computacionales.

Materia: Programación Nativa Para Móviles

Alumnos:

Yañez Herrera Karina Aileen

Zavala Osorio Camilo Alexander

Números de control:

21070415

21070336

Profesor: Jorge Peralta Escobar

Hora: 2:00 pm – 3:00 pm

Semestre: Enero - Junio 2025.

Lección 1

Video 1:

En este video comenzamos la Unidad 5 del curso, y como estudiantes nos introdujeron a los temas clave que vamos a explorar en esta etapa, especialmente el manejo de estados y ViewModel en Jetpack Compose. Aprendimos que, hasta ahora, muchas de nuestras apps eran sencillas y con lógica directa, pero que a medida que las aplicaciones se vuelven más interactivas y con más funcionalidades, es esencial entender cómo mantener el estado de la UI correctamente.

Nos explicaron que Jetpack Compose facilita esto, pero también requiere que seamos conscientes de cómo fluye la información entre los componentes. Aquí es donde entra en juego ViewModel, una clase que nos ayuda a almacenar y manejar datos relacionados con la UI de forma segura y persistente durante cambios de configuración como rotación de pantalla.

En resumen, esta unidad nos prepara para crear apps más robustas y con una experiencia de usuario más consistente, enseñándonos a separar claramente la lógica del estado de la interfaz visual, lo cual es un paso importante para seguir creciendo como desarrolladores de Android.

Video 2:

En este video aprendimos sobre el estado y el state hoisting en Jetpack Compose, dos conceptos fundamentales para construir interfaces reactivas y bien estructuradas. Como estudiantes, ya habíamos trabajado con Compose, pero ahora entendimos mejor cómo manejar el estado de forma adecuada.

Vimos que el estado representa los datos que cambian con el tiempo y que afectan lo que se muestra en la UI, por ejemplo, el texto que el usuario escribe o el contenido seleccionado. Compose actualiza automáticamente la interfaz cuando el estado cambia, lo cual es muy poderoso. También conocimos el concepto de state hoisting, que consiste en elevar el estado a un nivel superior para que pueda ser controlado desde afuera del composable, lo que permite que nuestros componentes sean más reutilizables, predecibles y fáciles de probar.

Comprendimos que un buen manejo del estado es clave para que nuestras apps funcionen bien y sean fáciles de mantener, y que aplicar estos principios correctamente es un paso importante para ser desarrolladores más avanzados.

Actividades:

Training Juice:

1.- MainActivity.kt

```
package com.example.juicetracker
import android.os.Bundle
import android.view.LayoutInflater
import androidx.appcompat.app.AppCompatActivity
import androidx.core.view.WindowCompat
import com.example.juicetracker.databinding.ActivityMainBinding

class MainActivity : AppCompatActivity() {

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)

        // Configura para que el contenido pueda extenderse detrás de las barras
del sistema (edge to edge)
        WindowCompat.setDecorFitsSystemWindows(window, false)

        // Infla la vista principal usando View Binding para un acceso más seguro a
las vistas
        val binding = ActivityMainBinding.inflate(LayoutInflater.from(this))

        // Establece la vista raíz inflada como contenido de la actividad
        setContentView(binding.root)

        // Establece la Toolbar definida en el layout como ActionBar de la actividad
        setSupportActionBar(binding.toolbar)
    }
}
```

Explicación breve: Esta clase es la actividad principal que se inicia cuando se abre la app. Configura la ventana para usar el modo edge-to-edge, infla el layout principal usando View Binding y asigna una Toolbar como barra de acciones.

2.- TrackerFragment.kt

```
package com.example.juicetracker
import android.os.Bundle
import android.view.LayoutInflater
```

```
import android.view.View
import android.view.ViewGroup
import androidx.fragment.app.Fragment
import androidx.fragment.app.viewModels
import androidx.lifecycle.Lifecycle
import androidx.lifecycle.lifecycleScope
import androidx.lifecycle.repeatOnLifecycle
import androidx.navigation.findNavController
import androidx.navigation.fragment.findNavController
import com.example.juicetracker.databinding.FragmentTrackerBinding
import com.example.juicetracker.ui.AppViewModelProvider
import com.example.juicetracker.ui.JuiceListAdapter
import com.example.juicetracker.ui.TrackerViewModel
import kotlinx.coroutines.launch

class TrackerFragment : Fragment() {

    // Obtiene el ViewModel con un proveedor personalizado
    private val viewModel by viewModels<TrackerViewModel> {
AppViewModelProvider.Factory }

    override fun onCreateView(
        inflater: LayoutInflater,
        container: ViewGroup?,
        savedInstanceState: Bundle?
    ): View {
        // Infla la vista usando View Binding para este fragmento
        return FragmentTrackerBinding.inflate(inflater, container, false).root
    }

    // Adaptador para la lista de jugos, con acciones para editar y borrar
    private val adapter = JuiceListAdapter(
        onEdit = { drink ->
            // Navega a la pantalla de edición pasando el id del jugo
            findNavController().navigate(

TrackerFragmentDirections.actionTrackerFragmentToEntryDialogFragment(drin
k.id)
            )
        },
        onDelete = { drink ->
            // Llama a la función del ViewModel para borrar el jugo
            viewModel.deleteJuice(drink)
        }
```

```
    )

    override fun onViewCreated(view: View, savedInstanceState: Bundle?) {
        val binding = FragmentTrackerBinding.bind(view)
        // Asigna el adaptador al RecyclerView
        binding.recyclerView.adapter = adapter

        // Configura el botón flotante para navegar al diálogo de entrada de jugo
nuevo
        binding.fab.setOnClickListener { fabView ->
            fabView.findNavController().navigate(

TrackerFragmentDirections.actionTrackerFragmentToEntryDialogFragment()
            )
        }

        // Observa el flujo de datos de jugos y actualiza la lista cuando cambia
        viewLifecycleOwner.lifecycleScope.launch {
            viewLifecycleOwner.repeatOnLifecycle(Lifecycle.State.STARTED) {
                viewModel.juicesStream.collect {
                    adapter.submitList(it)
                }
            }
        }
    }
}
```

Explicación breve:
Fragmento que muestra una lista de jugos consumidos. Usa un ViewModel para obtener los datos y actualiza el RecyclerView cuando los datos cambian. Permite editar o eliminar elementos y añadir nuevos jugos con un diálogo.

3.- JuiceTrackerApplication.kt

```
package com.example.juicetracker
import android.app.Application
import com.example.juicetracker.data.AppContainer
import com.example.juicetracker.data.AppDataContainer

class JuiceTrackerApplication : Application() {

    // Contenedor para manejar dependencias de la app (repositorios, bases de
datos, etc.)
```

```
    lateinit var container: AppContainer

    override fun onCreate() {
        super.onCreate()
        // Inicializa el contenedor con la implementación que usa datos reales
        container = AppDataContainer(this)
    }
}
```

Explicación breve: Clase Application personalizada para inicializar y proveer un contenedor de dependencias que luego será usado por otros componentes para acceder a repositorios o servicios.

4.- EntryDialogFragment.kt

```
package com.example.juicetracker
import android.annotation.SuppressLint
import android.os.Bundle
import android.view.LayoutInflater
import android.view.View
import android.view.ViewGroup
import android.widget.AdapterView
import android.widget.ArrayAdapter
import androidx.appcompat.R.layout
import androidx.core.widget.doOnTextChanged
import androidx.fragment.app.viewModels
import androidx.lifecycle.Lifecycle
import androidx.lifecycle.lifecycleScope
import androidx.lifecycle.repeatOnLifecycle
import androidx.navigation.fragment.navArgs
import com.example.juicetracker.data.JuiceColor
import com.example.juicetracker.databinding.FragmentEntryDialogBinding
import com.example.juicetracker.ui.AppViewModelProvider
import com.example.juicetracker.ui.EntryViewModel
import com.google.android.material.bottomsheet.BottomSheetDialogFragment
import kotlinx.coroutines.flow.filterNotNull
import kotlinx.coroutines.launch

class EntryDialogFragment : BottomSheetDialogFragment() {

    // ViewModel para manejar la lógica del diálogo de entrada
    private val entryViewModel by viewModels<EntryViewModel> {
AppViewModelProvider.Factory }
```

```
    var selectedColor: JuiceColor = JuiceColor.Red

    override fun onCreateView(
        inflater: LayoutInflater,
        container: ViewGroup?,
        savedInstanceState: Bundle?
    ): View {
        // Infla el layout para el diálogo con View Binding
        return FragmentEntryDialogBinding.inflate(inflater, container, false).root
    }

    @SuppressLint("SetTextI18n")
    override fun onViewCreated(view: View, savedInstanceState: Bundle?) {

        // Mapea etiquetas de color a sus valores correspondientes
        val colorLabelMap = JuiceColor.values().associateBy { getString(it.label) }
        val binding = FragmentEntryDialogBinding.bind(view)

        // Obtiene argumentos pasados a este diálogo, por ejemplo el id del jugo a
editar
        val args: EntryDialogFragmentArgs by navArgs()
        val juiceId = args.itemId

        if (args.itemId > 0) {
            // Si editamos un jugo existente, cargamos sus datos y los mostramos
            viewLifecycleOwner.lifecycleScope.launch {
                repeatOnLifecycle(Lifecycle.State.STARTED) {
                    entryViewModel.getJuiceStream(args.itemId).filterNotNull().collect {
item ->
                        with(binding){
                            name.setText(item.name)
                            description.setText(item.description)
                            ratingBar.rating = item.rating.toFloat()
                            colorSpinner.setSelection(findColorIndex(item.color))
                        }
                    }
                }
            }
        }

        // Habilita el botón guardar solo si el campo nombre tiene texto
        binding.name.doOnTextChanged { _, start, _, count ->
            binding.saveButton.isEnabled = (start + count) > 0
        }
```

```
        // Configura el spinner con los nombres de colores para selección
        binding.colorSpinner.adapter = ArrayAdapter(
            requireContext(),
            layout.support_simple_spinner_dropdown_item,
            colorLabelMap.map { it.key }
        )

        // Listener para cuando se selecciona un color del spinner
        binding.colorSpinner.onItemSelectedListener = object :
AdapterView.OnItemSelectedListener {
            override fun onItemSelected(parent: AdapterView<*>, view: View?, pos:
Int, id: Long) {
                val selected = parent.getItemAtPosition(pos).toString()
                selectedColor = colorLabelMap[selected] ?: selectedColor
            }
            override fun onNothingSelected(parent: AdapterView<*>) {
                selectedColor = JuiceColor.Red
            }
        }

        // Acción para guardar el jugo cuando se presiona el botón
        binding.saveButton.setOnClickListener {
            entryViewModel.saveJuice(
                juiceId,
                binding.name.text.toString(),
                binding.description.text.toString(),
                selectedColor.name,
                binding.ratingBar.rating.toInt()
            )
            dismiss() // Cierra el diálogo
        }

        // Cierra el diálogo si se presiona cancelar
        binding.cancelButton.setOnClickListener {
            dismiss()
        }
    }

    // Encuentra el índice del color en el arreglo de colores para seleccionarlo en
el spinner
    private fun findColorIndex(color: String): Int {
        val juiceColor = JuiceColor.valueOf(color)
        return JuiceColor.values().indexOf(juiceColor)
```

```
    }
}
```

Explicación breve: Diálogo que permite crear o editar un jugo. Permite ingresar nombre, descripción, color y calificación. Carga datos si es edición y valida entrada para habilitar el botón guardar.

5.- TrackerViewModel.kt

```
package com.example.juicetracker.ui
import androidx.lifecycle.ViewModel
import androidx.lifecycle.viewModelScope
import com.example.juicetracker.data.Juice
import com.example.juicetracker.data.JuiceRepository
import kotlinx.coroutines.flow.Flow
import kotlinx.coroutines.launch

class TrackerViewModel(private val juiceRepository: JuiceRepository):
ViewModel()  {

    // Flujo que expone la lista actualizada de jugos
    val juicesStream: Flow<List<Juice>> = juiceRepository.juicesStream

    // Función para eliminar un jugo de la base de datos o fuente de datos
    fun deleteJuice(juice: Juice) = viewModelScope.launch {
        juiceRepository.deleteJuice(juice)
    }
}
```

Explicación breve: ViewModel que expone un flujo de jugos para la UI y permite eliminar jugos usando el repositorio. Usa corutinas para operaciones asíncronas.

6.- JuiceListAdapter.kt

```
package com.example.juicetracker.ui
import android.view.LayoutInflater
import android.view.ViewGroup
import androidx.recyclerview.widget.DiffUtil
import androidx.recyclerview.widget.ListAdapter
import androidx.recyclerview.widget.RecyclerView
import com.example.juicetracker.data.Juice
```

```
import com.example.juicetracker.data.JuiceColor
import com.example.juicetracker.databinding.ListItemBinding

class JuiceListAdapter(
    private var onEdit: (Juice) -> Unit,
    private var onDelete: (Juice) -> Unit
) : ListAdapter<Juice,
JuiceListAdapter.JuiceListViewHolder>(JuiceDiffCallback()) {

    // ViewHolder para manejar la vista de cada ítem de jugo
    class JuiceListViewHolder(
        private val binding: ListItemBinding,
        private val onEdit: (Juice) -> Unit,
        private val onDelete: (Juice) -> Unit
    ) : RecyclerView.ViewHolder(binding.root) {
        private val nameView = binding.name
        private val description = binding.description
        private val drinkImage = binding.drinkColorOverlay
        private val ratingBar = binding.ratingBar

        fun bind(juice: Juice) {
            // Muestra los datos del jugo en la vista
            nameView.text = juice.name
            description.text = juice.description
            // Aplica filtro de color basado en el color del jugo
            drinkImage.setColorFilter(
                JuiceColor.valueOf(juice.color).color,
                android.graphics.PorterDuff.Mode.SRC_IN
            )
            ratingBar.rating = juice.rating.toFloat()

            // Configura botón para borrar el jugo
            binding.deleteButton.setOnClickListener {
                onDelete(juice)
            }

            // Click en el ítem para editar
            binding.root.setOnClickListener {
                onEdit(juice)
            }
        }
    }

    override fun onCreateViewHolder(parent: ViewGroup, viewType: Int) =
```

```
JuiceListViewHolder(
        ListItemBinding.inflate(LayoutInflater.from(parent.context), parent, false),
        onEdit,
        onDelete
    )

    override fun onBindViewHolder(holder: JuiceListViewHolder, position: Int) {
        holder.bind(getItem(position))
    }
}

//


Callback para optimizar el RecyclerView detectando cambios en la lista
 class JuiceDiffCallback : DiffUtil.ItemCallback() {
 override fun areItemsTheSame(oldItem: Juice, newItem: Juice): Boolean {
 return oldItem.id == newItem.id
 }
override fun areContentsTheSame(oldItem: Juice, newItem: Juice): Boolean {
    return oldItem == newItem
}


}
```

Explicación breve: Adaptador para RecyclerView que muestra una lista de jugos con nombre, descripción, color y calificación. Permite borrar y editar ítems mediante callbacks.

1.- RoomJuiceRepository.kt

```
package com.example.juicetracker.data
import kotlinx.coroutines.flow.Flow

class RoomJuiceRepository(private val juiceDao: JuiceDao) : JuiceRepository {
    override val juicesStream: Flow<List<Juice>> = juiceDao.getAll()
    override fun getJuiceStream(id: Long): Flow<Juice?> = juiceDao.get(id)
    override suspend fun addJuice(juice: Juice) = juiceDao.insert(juice)
    override suspend fun deleteJuice(juice: Juice) = juiceDao.delete(juice)
    override suspend fun updateJuice(juice: Juice) = juiceDao.update(juice)
}
```

Explicación:

- Esta clase implementa la interfaz JuiceRepository usando Room (base de datos SQLite en Android). Usa un JuiceDao para hacer consultas y operaciones de base de datos.
- juicesStream es un flujo (Flow) que emite la lista completa de jugos.
- Cada función delega la acción al juiceDao para obtener, agregar, borrar o actualizar jugos en la base de datos.

2.- JuiceRepository.kt

```
package com.example.juicetracker.data
import kotlinx.coroutines.flow.Flow

interface JuiceRepository {
    val juicesStream: Flow<List<Juice>>

    fun getJuiceStream(id: Long): Flow<Juice?>
    suspend fun addJuice(juice: Juice)
    suspend fun deleteJuice(juice: Juice)
    suspend fun updateJuice(juice: Juice)
}
```

Explicación:

- Esta es la interfaz que define el contrato para manejar datos de jugos.
- Define operaciones básicas: obtener todos los jugos, obtener uno por id, agregar, borrar y actualizar.
- Usa Flow para manejar datos reactivos (se puede observar cambios).
- Es independiente de la implementación específica (por ejemplo, puede usarse con Room o con otra fuente).

3.- JuiceDao.kt

```
package com.example.juicetracker.data
import androidx.room.Dao
import androidx.room.Delete
import androidx.room.Insert
import androidx.room.Query
import androidx.room.Update
import kotlinx.coroutines.flow.Flow

@Dao
interface JuiceDao {
    @Query("SELECT * FROM juice")
```

```
    fun getAll(): Flow<List<Juice>>

    @Query("SELECT * FROM juice WHERE id = :id")
    fun get(id: Long): Flow<Juice>

    @Insert
    suspend fun insert(juice: Juice)

    @Delete
    suspend fun delete(juice: Juice)

    @Update
    suspend fun update(juice: Juice)
}
```

Explicación:

- Esta interfaz es un DAO (Data Access Object) para Room, donde defines las consultas SQL y operaciones básicas para la entidad Juice.
- getAll() devuelve un flujo de lista con todos los jugos.
- get(id) devuelve un flujo con un jugo específico.
- Los métodos insert, delete y update son operaciones suspendidas para modificar la base de datos.

4.- Juice.kt

```
package com.example.juicetracker.data
import android.graphics.Color
import androidx.annotation.StringRes
import androidx.room.Entity
import androidx.room.PrimaryKey
import com.example.juicetracker.R

@Entity
data class Juice(
    @PrimaryKey(autoGenerate = true)
    val id: Long,
    val name: String,
    val description: String = "",
    val color: String,
    val rating: Int
)
```

```
enum class JuiceColor(val color: Int, @StringRes val label: Int) {
    Red(Color.RED, R.string.red),
    Blue(Color.BLUE, R.string.blue),
    Green(Color.GREEN, R.string.green),
    Cyan(Color.CYAN, R.string.cyan),
    Yellow(Color.YELLOW, R.string.yellow),
    Magenta(Color.MAGENTA, R.string.magenta)
}
```

Explicación:

- La clase es una entidad de Room con @Entity.
- Tiene un id autogenerado, nombre, descripción (opcional), color (como String, por ejemplo un código hexadecimal), y una calificación (rating).
- Además, defines un enum JuiceColor con colores predefinidos para usar en la UI, cada uno con un color Android y una referencia a un string de recurso.

5.- AppDataContainer.kt

```
package com.example.juicetracker.data
import android.content.Context

class AppDataContainer(private val context: Context) : AppContainer {

    override val trackerRepository: JuiceRepository by lazy {
        RoomJuiceRepository(AppDatabase.getDatabase(context).juiceDao())
    }
}
```

Explicación:

- Este contenedor de datos es una implementación de AppContainer que proporciona la instancia de JuiceRepository.
- Usa el contexto para obtener la base de datos y el DAO, luego construye el repositorio Room.
- El repositorio se inicializa perezosamente (by lazy), se crea solo cuando se usa por primera vez.

6.- AppDatabase.kt

```
package com.example.juicetracker.data
import android.content.Context
```

```
import androidx.room.Database
import androidx.room.Room
import androidx.room.RoomDatabase

@Database(entities = [Juice::class], version = 1)
abstract class AppDatabase: RoomDatabase() {
    abstract fun juiceDao(): JuiceDao
    companion object {
        @Volatile
        private var INSTANCE: AppDatabase? = null
        fun getDatabase(context: Context): AppDatabase {
            return INSTANCE ?: synchronized(this) {
                Room.databaseBuilder(context, AppDatabase::class.java,
"app_database")
                    .fallbackToDestructiveMigration()
                    .build()
                    .also { INSTANCE = it }
            }
        }
    }
}
```

Explicación:

- Clase que configura la base de datos Room para la app.
- Declara que Juice es la entidad para esta base.
- Define el DAO juiceDao().
- Usa un patrón Singleton para asegurar que solo haya una instancia de la base de datos durante toda la app.
- El método getDatabase crea o devuelve la instancia existente.
- fallbackToDestructiveMigration() indica que si la versión cambia, se destruye la base y se crea una nueva (pérdida de datos en ese caso).

7.- AppContainer.kt

```
package com.example.juicetracker.data

interface AppContainer {
    val trackerRepository: JuiceRepository
}
```

Explicación:

- Es una interfaz simple que define un contenedor para las dependencias de datos de la app.
- Solo expone la propiedad trackerRepository de tipo JuiceRepository.
- Permite hacer inyección de dependencias o separar la lógica de datos para pruebas o para cambiar la implementación.

Video 3:

En este video aprendimos sobre el uso del ViewModel en Jetpack Compose, una herramienta clave para manejar el estado de manera segura y eficiente en nuestras aplicaciones. Como estudiantes, vimos que el ViewModel nos ayuda a guardar y gestionar datos relacionados con la interfaz de usuario de forma que esos datos persistan incluso cuando ocurren cambios de configuración, como rotar la pantalla.

Entendimos que, al usar ViewModel, podemos separar la lógica de la UI del resto de la aplicación, lo que mejora la organización y facilita las pruebas. También aprendimos a crear un ViewModel, a acceder a él desde los composables y a observar sus datos para que la UI se actualice automáticamente cuando el estado cambia. En resumen, este video nos mostró cómo el ViewModel es una pieza fundamental para construir apps robustas y con buen manejo del ciclo de vida en Jetpack Compose, y cómo usarlo correctamente mejora la calidad y estabilidad de nuestras aplicaciones.

Lección 2

Video 1:

En este video aprendimos sobre SavedStateHandle, una herramienta dentro del ViewModel que nos ayuda a guardar y restaurar el estado de la UI incluso si la app es cerrada o el proceso es terminado por el sistema. Como estudiantes, vimos que aunque el ViewModel mantiene datos durante cambios de configuración como la rotación de pantalla, no siempre protege contra que la app sea eliminada de la memoria.

Ahí es donde entra SavedStateHandle, que guarda datos en un lugar seguro para que puedan recuperarse después. Nos mostraron cómo usar SavedStateHandle para almacenar datos clave y cómo acceder a ellos desde el ViewModel para mantener la experiencia del usuario intacta, incluso en escenarios complejos. En resumen, aprendimos que SavedStateHandle es esencial para crear apps más resistentes y que mantienen el estado del usuario, mejorando la estabilidad y usabilidad de nuestras aplicaciones.

Actividades:

1.- MainActivity.kt

```
package com.example.juicetracker
import android.os.Bundle
import androidx.activity.ComponentActivity
import androidx.activity.compose.setContent
import androidx.activity.enableEdgeToEdge
import com.example.juicetracker.ui.JuiceTrackerApp
import com.example.juicetracker.ui.theme.JuiceTrackerTheme

class MainActivity : ComponentActivity() {
    override fun onCreate(savedInstanceState: Bundle?) {
        enableEdgeToEdge()
        super.onCreate(savedInstanceState)
        setContent {
            JuiceTrackerTheme {
                JuiceTrackerApp()
            }
        }
    }
}
```

Explicación:

- Esta es la actividad principal de la app.
- Usa ComponentActivity compatible con Jetpack Compose.
- En onCreate, primero habilita el modo Edge-to-Edge para que el contenido ocupe toda la pantalla (sin barras transparentes).
- Usa setContent para configurar la UI declarativa con Compose.
- Aplica un tema personalizado JuiceTrackerTheme.
- Llama a la función composable JuiceTrackerApp() que contiene la interfaz principal.

2.- JuiceTrackerApplication.kt

```
package com.example.juicetracker
import android.app.Application
import com.example.juicetracker.data.AppContainer
import com.example.juicetracker.data.AppDataContainer

class JuiceTrackerApplication : Application() {
```

```
    lateinit var container: AppContainer
    override fun onCreate() {
        super.onCreate()
        container = AppDataContainer(this)
    }
}
```

Explicación:

- Esta clase extiende Application y se usa para inicializar componentes globales.
- Declara una variable container de tipo AppContainer que gestionará dependencias de datos.
- En onCreate inicializa container con AppDataContainer, que usa contexto para proveer el repositorio de datos.

3.- AppContainer.kt

```
package com.example.juicetracker.data

interface AppContainer {
    val juiceRepository: JuiceRepository
}
```

Explicación:

- Interfaz que define un contenedor para proveer el repositorio de jugos (JuiceRepository).
- Facilita la inyección de dependencias y la abstracción para pruebas o cambios futuros.

4.- AppDataContainer.kt

```
package com.example.juicetracker.data
import android.content.Context

class AppDataContainer(private val context: Context) : AppContainer {

    override val juiceRepository: JuiceRepository by lazy {
        RoomJuiceRepository(AppDatabase.getDatabase(context).juiceDao())
    }
}
```

Explicación:

- Implementación concreta de AppContainer.
- Usa contexto para obtener la instancia de la base de datos y el DAO.
- Inicializa perezosamente el repositorio con RoomJuiceRepository, que maneja la persistencia usando Room.

5.- Juice.kt

```
package com.example.juicetracker.data

import androidx.annotation.StringRes
import androidx.compose.ui.graphics.Color
import androidx.room.Entity
import androidx.room.PrimaryKey
import com.example.juicetracker.R
import com.example.juicetracker.ui.theme.Orange as OrangeColor

@Entity
data class Juice(
    @PrimaryKey(autoGenerate = true)
    val id: Long,
    val name: String,
    val description: String = "",
    val color: String,
    val rating: Int
)

enum class JuiceColor(val color: Color, @StringRes val label: Int) {
    Red(Color.Red, R.string.red),
    Blue(Color.Blue, R.string.blue),
    Green(Color.Green, R.string.green),
    Cyan(Color.Cyan, R.string.cyan),
    Yellow(Color.Yellow, R.string.yellow),
    Magenta(Color.Magenta, R.string.magenta),
    Orange(OrangeColor, R.string.orange)
}
```

Explicación:

- Modelo de datos Juice para Room y UI.
- @Entity marca que es una tabla en la base de datos.
- Los colores ahora usan Color de Compose para facilitar el manejo en la UI.

- Se agrega un nuevo color Orange con su color personalizado desde el tema.
- Enum JuiceColor asocia colores y recursos de strings para mostrar etiquetas.

6.- JuiceDao.kt

```
package com.example.juicetracker.data
import androidx.room.Dao
import androidx.room.Delete
import androidx.room.Insert
import androidx.room.Query
import androidx.room.Update
import kotlinx.coroutines.flow.Flow

@Dao
interface JuiceDao {
    @Query("SELECT * FROM juice")
    fun getAll(): Flow<List<Juice>>

    @Insert
    suspend fun insert(juice: Juice)

    @Delete
    suspend fun delete(juice: Juice)

    @Update
    suspend fun update(juice: Juice)
}
```

Explicación:

- DAO de Room para acceso a la tabla juice.
- Permite obtener todos los jugos con un flujo.
- Operaciones básicas de insertar, borrar y actualizar.

7.- JuiceRepository.kt

```
package com.example.juicetracker.data
import kotlinx.coroutines.flow.Flow

interface JuiceRepository {
    val juiceStream: Flow<List<Juice>>
```

```
    suspend fun addJuice(juice: Juice)
    suspend fun deleteJuice(juice: Juice)
    suspend fun updateJuice(juice: Juice)
}
```

Explicación:

- Interfaz que define las operaciones para manejar los datos de jugos.
- Flujo para observar la lista de jugos.
- Funciones para agregar, borrar y actualizar.

8.- RoomJuiceRepository.kt

```
package com.example.juicetracker.data
import kotlinx.coroutines.flow.Flow

class RoomJuiceRepository(private val juiceDao: JuiceDao) : JuiceRepository {
    override val juiceStream: Flow<List<Juice>> = juiceDao.getAll()

    override suspend fun addJuice(juice: Juice) = juiceDao.insert(juice)
    override suspend fun deleteJuice(juice: Juice) = juiceDao.delete(juice)
    override suspend fun updateJuice(juice: Juice) = juiceDao.update(juice)
}
```

Explicación:

- Implementación concreta de JuiceRepository usando JuiceDao de Room.
- Provee el flujo de lista completa.
- Delegación directa de operaciones de datos a las funciones del DAO.

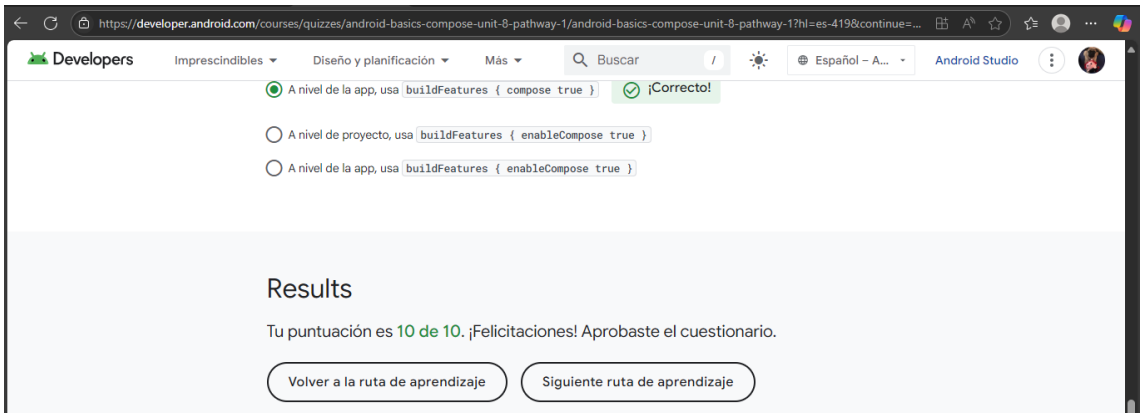
Video 2:

En este video exploramos cómo usar StateFlow y SharedFlow en Jetpack Compose para manejar flujos de datos de manera reactiva y eficiente dentro de nuestras aplicaciones. Como estudiantes, aprendimos que StateFlow es ideal para representar estados que cambian y necesitan ser observados continuamente por la UI, mientras que SharedFlow es mejor para eventos únicos o acciones que no representan un estado persistente, como mostrar un mensaje o navegar a otra pantalla. Vimos cómo integrar estas herramientas dentro de un ViewModel para emitir cambios y cómo los composables pueden recolectar esos flujos para actualizar la interfaz automáticamente.

Este enfoque nos permite tener un manejo más limpio y estructurado del estado y los eventos en nuestras apps, facilitando la separación de responsabilidades y mejorando la reactividad y la experiencia del usuario. En resumen, este video nos dio conocimientos importantes para avanzar en la gestión avanzada del estado usando flujos en Kotlin dentro de Jetpack Compose.

Cuestionario:

Lección 1:



Lección 2: