



**TECNOLÓGICO
NACIONAL DE MÉXICO**



**INSTITUTO TECNOLÓGICO
DE CIUDAD MADERO**
www.itcm.edu.mx

Instituto Tecnológico de Ciudad Madero

INGENIERÍA EN SISTEMAS COMPUTACIONALES

EQUIPO

Flores Ramírez Manuel Abraham

Ian samuel García Pérez

No. De control

20070511

20070611

MATERIA

Programación nativa para móviles

TAREA

Tarea No 6 Unidad 6 Persistencia de datos

Semestre

Decimo

Maestro

Jorge Peralta Escobar

Introducción a SQL

Como usar Room para la persistencia de datos

<https://github.com/Abraham20070511/PruebasdeFlujo>

```
package com.example.inventory
import android.content.Context
import androidx.room.Room
import androidx.test.core.app.ApplicationProvider
import androidx.test.ext.junit.runners.AndroidJUnit4
import com.example.inventory.data.InventoryDatabase
import com.example.inventory.data.Item
import com.example.inventory.data.ItemDao
import kotlinx.coroutines.flow.first
import kotlinx.coroutines.runBlocking
import org.junit.After
import org.junit.Assert.assertEquals
import org.junit.Assert.assertTrue
import org.junit.Before
import org.junit.Test
import org.junit.runner.RunWith
import java.io.IOException

// Indica que esta clase usará el runner de pruebas de Android para ejecutar
pruebas JUnit
@RunWith(AndroidJUnit4::class)
class ItemDaoTest {

    // Declaración de las variables necesarias
    private lateinit var itemDao: ItemDao // DAO que se va a probar
    private lateinit var inventoryDatabase: InventoryDatabase // Base de datos en
memoria

    private val item1 = Item(1, "Apples", 10.0, 20) // Item de prueba 1
    private val item2 = Item(2, "Bananas", 15.0, 97) // Item de prueba 2
```

```

// Esta función se ejecuta antes de cada prueba
@Before
fun createDb() {
    val context: Context = ApplicationProvider.getApplicationContext() // Obtiene
    el contexto de prueba

    // Crea una base de datos en memoria (no persistente) para pruebas
    inventoryDatabase = Room.inMemoryDatabaseBuilder(context,
InventoryDatabase::class.java)
        .allowMainThreadQueries() // Permite consultas en el hilo principal solo
    para pruebas
        .build()

    // Obtiene el DAO de la base de datos
    itemDao = inventoryDatabase.itemDao()
}

// Esta función se ejecuta después de cada prueba
@After
@Throws(IOException::class)
fun closeDb() {
    inventoryDatabase.close() // Cierra la base de datos
}

// Prueba que al insertar un ítem, este se guarda correctamente
@Test
@Throws(Exception::class)
fun daoInsert_insertsItemIntoDB() = runBlocking {
    addOneItemToDb() // Inserta item1
    val allItems = itemDao.getAllItems().first() // Obtiene todos los ítems
    assertEquals(allItems[0], item1) // Verifica que el primer ítem sea igual a item1
}

// Prueba que se puedan recuperar múltiples ítems
@Test

```

```

@Throws(Exception::class)
fun daoGetAllItems_returnsAllItemsFromDB() = runBlocking {
    addTwoItemsToDb() // Inserta item1 y item2
    val allItems = itemDao.getAllItems().first()
    assertEquals(allItems[0], item1) // Verifica el primer ítem
    assertEquals(allItems[1], item2) // Verifica el segundo ítem
}

// Prueba que se pueda obtener un ítem específico por ID
@Test
@Throws(Exception::class)
fun daoGetItem_returnsItemFromDB() = runBlocking {
    addOneItemToDb() // Inserta item1
    val item = itemDao.getItem(1) // Recupera el ítem con ID 1
    assertEquals(item.first(), item1) // Verifica que sea igual a item1
}

// Prueba que se puedan eliminar ítems de la base de datos
@Test
@Throws(Exception::class)
fun daoDeleteItems_deletesAllItemsFromDB() = runBlocking {
    addTwoItemsToDb() // Inserta item1 y item2
    itemDao.delete(item1) // Elimina item1
    itemDao.delete(item2) // Elimina item2
    val allItems = itemDao.getAllItems().first()
    assertTrue(allItems.isEmpty()) // Verifica que la base de datos esté vacía
}

// Prueba que se puedan actualizar ítems en la base de datos
@Test
@Throws(Exception::class)
fun daoUpdateItems_updatesItemsInDB() = runBlocking {
    addTwoItemsToDb() // Inserta ítems originales

    // Actualiza ambos ítems con nuevos valores

```

```

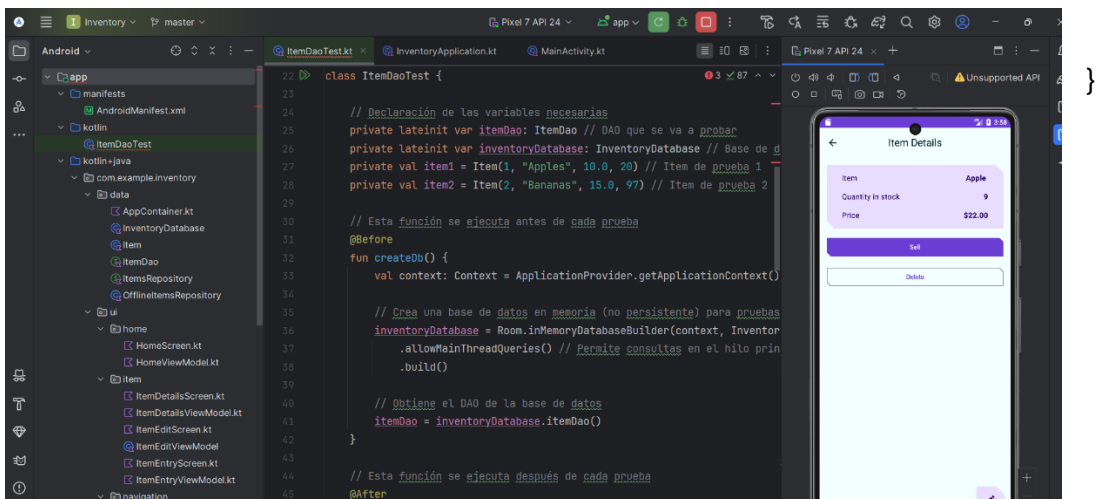
itemDao.update(Item(1, "Apples", 15.0, 25))
itemDao.update(Item(2, "Bananas", 5.0, 50))

val allItems = itemDao.getAllItems().first()
// Verifica que los ítems se hayan actualizado correctamente
assertEquals(allItems[0], Item(1, "Apples", 15.0, 25))
assertEquals(allItems[1], Item(2, "Bananas", 5.0, 50))
}

// Función auxiliar para insertar solo item1
private suspend fun addOneItemToDb() {
    itemDao.insert(item1)
}

// Función auxiliar para insertar item1 y item2
private suspend fun addTwoItemsToDb() {
    itemDao.insert(item1)
    itemDao.insert(item2)
}

```



Realiza pruebas automatizadas para verificar el correcto funcionamiento del sistema de inventario de una aplicación Android. Específicamente, comprueba que se puedan insertar, obtener, actualizar y eliminar productos (ítems) en la base de datos utilizando Room. Estas pruebas aseguran que el manejo de los datos del inventario sea confiable, permitiendo operaciones como agregar productos, modificar su información (precio, cantidad), consultarlos por ID o lista completa, y eliminarlos correctamente. Todo esto se prueba en una base de datos en memoria para no afectar los datos reales de la aplicación.

Bus Schedule

<https://github.com/Abraham20070511/PracticaBusSchedule>

/*

* Copyright (C) 2023 The Android Open Source Project

*

* Licensed under the Apache License, Version 2.0 (the "License");

* you may not use this file except in compliance with the License.

* You may obtain a copy of the License at

*

* <https://www.apache.org/licenses/LICENSE-2.0>

*

* Unless required by applicable law or agreed to in writing, software

* distributed under the License is distributed on an "AS IS" BASIS,

* WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.

* See the License for the specific language governing permissions and

* limitations under the License.

*/

package com.example.busschedule.ui

import androidx.activity.compose.BackHandler

import androidx.compose.foundation.clickable

import androidx.compose.foundation.layout.Arrangement

import androidx.compose.foundation.layout.Column

import androidx.compose.foundation.layout.PaddingValues

import androidx.compose.foundation.layout.Row

import androidx.compose.foundation.layout.calculateEndPadding

import androidx.compose.foundation.layout.calculateStartPadding

import androidx.compose.foundation.layout.fillMaxWidth

import androidx.compose.foundation.layout.padding

import androidx.compose.foundation.lazy.LazyColumn

import androidx.compose.foundation.lazy.items

import androidx.compose.material.icons.Icons

import androidx.compose.material.icons.filled.ArrowBack

```
import androidx.compose.material3.Divider
import androidx.compose.material3.ExperimentalMaterial3Api
import androidx.compose.material3.Icon
import androidx.compose.material3.IconButton
import androidx.compose.material3.MaterialTheme
import androidx.compose.material3.Scaffold
import androidx.compose.material3.Text
import androidx.compose.material3.TopAppBar
import androidx.compose.runtime.Composable
import androidx.compose.runtime.collectAsState
import androidx.compose.runtime.getValue
import androidx.compose.runtime.mutableStateOf
import androidx.compose.runtime.remember
import androidx.compose.runtime.setValue
import androidx.compose.ui.Modifier
import androidx.compose.ui.platform.LocalLayoutDirection
import androidx.compose.ui.res.dimensionResource
import androidx.compose.ui.res.stringResource
import androidx.compose.ui.text.font.FontWeight
import androidx.compose.ui.text.style.TextAlign
import androidx.compose.ui.tooling.preview.Preview
import androidx.compose.ui.unit.dp
import androidx.compose.ui.unit.sp
import androidx.lifecycle.viewmodel.compose.viewModel
import androidx.navigation.NavType
import androidx.navigation.compose.NavHost
import androidx.navigation.compose.composable
import androidx.navigation.compose.rememberNavController
import androidx.navigation.navArgument
import com.example.busschedule.R
import com.example.busschedule.data.BusSchedule
import com.example.busschedule.ui.theme.BusScheduleTheme
import java.text.SimpleDateFormat
import java.util.Date
import java.util.Locale
```

```

// Enumeración que define las pantallas disponibles en la app
enum class BusScheduleScreens {
    FullSchedule, // Pantalla con todos los horarios
    RouteSchedule // Pantalla con horarios de una ruta específica
}

/**
 * Composable principal de la aplicación.
 * @param viewModel ViewModel que maneja la lógica de negocio
 */
@Composable
fun BusScheduleApp(
    viewModel: BusScheduleViewModel = viewModel(factory =
BusScheduleViewModel.factory)
) {
    // 1. Controlador de navegación que gestiona el stack de pantallas
    val navController = rememberNavController()

    // 2. Título inicial de la app (usando recursos de strings)
    val fullScheduleTitle = stringResource(R.string.full_schedule)

    // 3. Estado para el título de la TopAppBar (puede cambiar al navegar)
    var topAppBarTitle by remember { mutableStateOf(fullScheduleTitle) }

    // 4. Obtiene el horario completo como un estado observable
    val fullSchedule by viewModel.getFullSchedule().collectAsState(emptyList())

    // 5. Función que maneja el evento de retroceso
    val onBackHandler = {
        topAppBarTitle = fullScheduleTitle // Restaura el título
        navController.navigateUp() // Navega a la pantalla anterior
    }

    // 6. Scaffold es la estructura base de la pantalla (con TopAppBar y contenido)

```



```

Scaffold(
  topBar = {
    // 7. Barra superior personalizada
    BusScheduleTopAppBar(
      title = topAppBarTitle,
      canNavigateBack = navController.previousBackStackEntry != null,
      onBackClick = { onBackHandler() }
    )
  }
) { innerPadding -> // Padding interno calculado por Scaffold
  // 8. Sistema de navegación
  NavHost(
    navController = navController,
    startDestination = BusScheduleScreens.FullSchedule.name
  ) {
    // 9. Definición de la pantalla de horario completo
    composable(BusScheduleScreens.FullSchedule.name) {
      FullScheduleScreen(
        busSchedules = fullSchedule,
        contentPadding = innerPadding,
        onScheduleClick = { busStopName ->
          // 10. Navega a la pantalla de detalle al hacer clic en una ruta
          navController.navigate(
            "${BusScheduleScreens.RouteSchedule.name}/$busStopName"
          )
          topAppBarTitle = busStopName // Actualiza el título
        }
      )
    }
  }

  // 11. Argumento para pasar el nombre de la parada entre pantallas
  val busRouteArgument = "busRoute"

  // 12. Definición de la pantalla de horario por ruta
  composable(

```

```

        route = BusScheduleScreens.RouteSchedule.name +
"/{$busRouteArgument}",
        arguments = listOf(navArgument(busRouteArgument) { type =
NavType.StringType })
    ) { backStackEntry ->
        // 13. Obtiene el nombre de la parada desde los argumentos
        val stopName =
backStackEntry.arguments?.getString(busRouteArgument)
            ?: error("busRouteArgument cannot be null")

        // 14. Obtiene el horario específico para esa parada
        val routeSchedule by
viewModel.getScheduleFor(stopName).collectAsState(emptyList())

        // 15. Muestra la pantalla de detalle
        RouteScheduleScreen(
            stopName = stopName,
            busSchedules = routeSchedule,
            contentPadding = innerPadding,
            onBack = { onBackHandler() }
        )
    }
}
}
}

/**
 * Pantalla que muestra todos los horarios disponibles.
 * @param busSchedules Lista de horarios
 * @param onScheduleClick Callback al seleccionar una ruta
 * @param modifier Modificador para personalización
 * @param contentPadding Padding interno
 */
@Composable
fun FullScheduleScreen(

```

```

        busSchedules: List<BusSchedule>,
        onScheduleClick: (String) -> Unit,
        modifier: Modifier = Modifier,
        contentPadding: PaddingValues = PaddingValues(0.dp),
    ) {
        // 16. Reutiliza el componente base pasando el callback
        BusScheduleScreen(
            busSchedules = busSchedules,
            onScheduleClick = onScheduleClick,
            contentPadding = contentPadding,
            modifier = modifier
        )
    }

```

```

/**
 * Pantalla que muestra los horarios de una ruta específica.
 * @param stopName Nombre de la parada
 * @param busSchedules Lista de horarios para esta parada
 * @param modifier Modificador para personalización
 * @param contentPadding Padding interno
 * @param onBack Callback para el botón de retroceso
 */

```

```

@Composable

```

```

fun RouteScheduleScreen(
    stopName: String,
    busSchedules: List<BusSchedule>,
    modifier: Modifier = Modifier,
    contentPadding: PaddingValues = PaddingValues(0.dp),
    onBack: () -> Unit = {}
) {
    // 17. Maneja el botón de retroceso físico
    BackHandler { onBack() }

```

```

    // 18. Reutiliza el componente base especificando el nombre de la parada
    BusScheduleScreen(

```

```

        busSchedules = busSchedules,
        modifier = modifier,
        contentPadding = contentPadding,
        stopName = stopName
    )
}

/**
 * Componente base para mostrar horarios de autobuses.
 * @param busSchedules Lista de horarios
 * @param modifier Modificador para personalización
 * @param contentPadding Padding interno
 * @param stopName Nombre de la parada (opcional)
 * @param onScheduleClick Callback al seleccionar (opcional)
 */
@Composable
fun BusScheduleScreen(
    busSchedules: List<BusSchedule>,
    modifier: Modifier = Modifier,
    contentPadding: PaddingValues = PaddingValues(0.dp),
    stopName: String? = null,
    onScheduleClick: ((String) -> Unit)? = null,
) {
    // 19. Determina el texto del encabezado según si hay nombre de parada
    específico
    val stopNameText = if (stopName == null) {
        stringResource(R.string.stop_name) // Texto genérico
    } else {
        "$stopName ${stringResource(R.string.route_stop_name)}" // Texto específico
    }

    // 20. Obtiene la dirección del layout (LTR o RTL)
    val layoutDirection = LocalLayoutDirection.current

    // 21. Columna principal que organiza los elementos

```

```

Column(
    modifier = modifier.padding(
        start = contentPadding.calculateStartPadding(layoutDirection),
        end = contentPadding.calculateEndPadding(layoutDirection),
    )
) {
    // 22. Fila para los encabezados de las columnas
    Row(
        modifier = Modifier
            .fillMaxWidth()
            .padding(
                top = contentPadding.calculateTopPadding(),
                bottom = dimensionResource(R.dimen.padding_medium),
                start = dimensionResource(R.dimen.padding_medium),
                end = dimensionResource(R.dimen.padding_medium),
            ),
        horizontalArrangement = Arrangement.SpaceBetween // Espacio entre
elementos
    ) {
        Text(stopNameText) // Encabezado de nombre de parada
        Text(stringResource(R.string.arrival_time)) // Encabezado de hora
    }

    Divider() // 23. Línea divisoria

    // 24. Componente con la lista de horarios
    BusScheduleDetails(
        contentPadding = PaddingValues(
            bottom = contentPadding.calculateBottomPadding()
        ),
        busSchedules = busSchedules,
        onScheduleClick = onScheduleClick
    )
}
}

```

```

/**
 * Componente que muestra la lista de horarios.
 * @param busSchedules Lista de horarios
 * @param modifier Modificador para personalización
 * @param contentPadding Padding interno
 * @param onScheduleClick Callback al seleccionar (opcional)
 */
@Composable
fun BusScheduleDetails(
    busSchedules: List<BusSchedule>,
    modifier: Modifier = Modifier,
    contentPadding: PaddingValues = PaddingValues(0.dp),
    onScheduleClick: ((String) -> Unit)? = null
) {
    // 25. LazyColumn es una lista eficiente que solo renderiza lo visible
    LazyColumn(
        modifier = modifier,
        contentPadding = contentPadding,
    ) {
        // 26. Itera sobre los horarios
        items(
            items = busSchedules,
            key = { busSchedule -> busSchedule.id } // Clave única para cada item
        ) { schedule ->
            // 27. Fila para cada horario
            Row(
                modifier = Modifier
                    .fillMaxWidth()
                    .clickable(enabled = onScheduleClick != null) {
                        // 28. Si hay callback, lo ejecuta al hacer clic
                        onScheduleClick?.invoke(schedule.stopName)
                    }
                .padding(dimensionResource(R.dimen.padding_medium)),
                horizontalArrangement = Arrangement.SpaceBetween
            )
        }
    }
}

```

```

) {
    // 29. Renderizado condicional del nombre de parada
    if (onScheduleClick == null) {
        // 30. Si no hay callback, muestra un placeholder
        Text(
            text = "--",
            style = MaterialTheme.typography.bodyLarge.copy(
                fontSize = dimensionResource(R.dimen.font_large).value.sp,
                fontWeight = FontWeight(300)
            ),
            textAlign = TextAlign.Center,
            modifier = Modifier.weight(1f)
        )
    } else {
        // 31. Muestra el nombre real de la parada
        Text(
            text = schedule.stopName,
            style = MaterialTheme.typography.bodyLarge.copy(
                fontSize = dimensionResource(R.dimen.font_large).value.sp,
                fontWeight = FontWeight(300)
            )
        )
    }
}

// 32. Muestra la hora formateada
Text(
    text = SimpleDateFormat("h:mm a", Locale.getDefault())
        .format(Date(schedule.arrivalTimeInMillis.toLong() * 1000)),
    style = MaterialTheme.typography.bodyLarge.copy(
        fontSize = dimensionResource(R.dimen.font_large).value.sp,
        fontWeight = FontWeight(600) // Texto más grueso
    ),
    textAlign = TextAlign.End,
    modifier = Modifier.weight(2f) // Ocupa más espacio
)

```

```

    }
    }
}

/**
 * Barra superior personalizada.
 * @param title Título a mostrar
 * @param canNavigateBack Si muestra el botón de retroceso
 * @param onBackClick Callback para el botón de retroceso
 * @param modifier Modificador para personalización
 */
@OptIn(ExperimentalMaterial3Api::class)
@Composable
fun BusScheduleTopAppBar(
    title: String,
    canNavigateBack: Boolean,
    onBackClick: () -> Unit,
    modifier: Modifier = Modifier
) {
    // 33. Renderizado condicional según si debe mostrar botón de retroceso
    if (canNavigateBack) {
        TopAppBar(
            title = { Text(title) }, // Título
            navigationIcon = {
                // 34. Botón de retroceso con icono
                IconButton(onClick = onBackClick) {
                    Icon(
                        imageVector = Icons.Filled.ArrowBack,
                        contentDescription = stringResource(R.string.back)
                    )
                }
            },
            modifier = modifier
        )
    }
}

```



```

    } else {
        // 35. Barra sin botón de retroceso
        TopAppBar(
            title = { Text(title) },
            modifier = modifier
        )
    }
}

```

// Previews para Android Studio

```

@Preview(showBackground = true)
@Composable
fun FullScheduleScreenPreview() {
    BusScheduleTheme {
        FullScheduleScreen(
            busSchedules = List(3) { index ->
                BusSchedule(
                    index,
                    "Main Street",
                    111111
                )
            },
            onScheduleClick = {}
        )
    }
}

```

```

@Preview(showBackground = true)
@Composable
fun RouteScheduleScreenPreview() {
    BusScheduleTheme {
        RouteScheduleScreen(
            stopName = "Main Street",
            busSchedules = List(3) { index ->

```

The screenshot displays the Android Studio IDE with the following components:

- Left Panel (Project Explorer):** Shows the project structure for "Bus Schedule". The "ui" folder contains "BusScheduleScreens.kt", which is the current file being edited.
- Top Panel (Tabs):** Shows the "BusScheduleViewModel.kt" and "BusScheduleScreens.kt" tabs.
- Center Panel (Code Editor):** Displays the Kotlin code for "BusScheduleApp". The code defines a route argument and a back stack entry, and it shows the logic for displaying the bus schedule.
- Right Panel (Preview):** Shows a preview of the app's UI. It displays a list of bus stops and their arrival times, matching the data in the code.

The code in the center panel is as follows:

```

79 fun BusScheduleApp(
110     ) { innerPadding -> // Padding interno calculado por Sc
115     } {
132         val busRouteArgument = "busRoute"
133
134         // 12. Definición de la pantalla de horario por
135         // composables()
136         route = BusScheduleScreens.RouteSchedule.nav
137         arguments = listOf(navArgument<busRouteArgum
138     ) { backStackEntry ->
139         // 13. Obtiene el nombre de la parada desde
140         val stopName = backStackEntry.arguments?.get
141             ?: error("busRouteArgument cannot be nul
142
143         // 14. Obtiene el horario específico para es
144         val routeSchedule by viewModel.getScheduleFc
145
146         // 15. Muestra la pantalla de detalle
147         RouteScheduleScreen(
148             stopName = stopName,
149             busSchedules = routeSchedule,
150             contentPadding = innerPadding,
151             onBackPressed = { onBackPressed() }
152         )
153     }
154 }
155

```

The UI preview on the right shows a list of bus stops and their arrival times:

Stop Name	Arrival Time
Main Street	3:00 PM
Park Street	3:12 PM
Maple Avenue	3:25 PM
Broadway Avenue	3:41 PM
Post Street	3:58 PM
Elm Street	4:09 PM
Oak Drive	4:20 PM
Middle Street	4:34 PM
Palm Avenue	4:51 PM
Winding Way	4:55 PM
Main Street	5:00 PM
Park Street	5:12 PM
Maple Avenue	5:25 PM
Broadway Avenue	5:41 PM

La aplicación utiliza un sistema de navegación basado en NavHost y NavController, lo cual permite cambiar de pantalla de manera fluida sin perder el estado. También maneja el botón físico de retroceso del dispositivo Android usando BackHandler, lo que garantiza una experiencia coherente para el usuario. Además, la barra superior

(TopAppBar) se actualiza dinámicamente para mostrar el título adecuado según la pantalla actual, ya sea el nombre de la parada o el título general de los horarios.

La estructura del diseño se basa en componentes reutilizables. Por ejemplo, BusScheduleScreen se utiliza tanto para la pantalla principal como para la pantalla por parada, permitiendo mantener la consistencia visual y reducir duplicación de código. Los datos que se muestran provienen de un ViewModel, el cual probablemente se conecta a una base de datos local como Room, y se exponen como flujos observables (collectAsState). El componente LazyColumn permite mostrar listas de forma eficiente, cargando solo los elementos visibles.

En cuanto a los datos, cada entrada de horario de autobús está representada por una clase BusSchedule que contiene un identificador (id), el nombre de la parada (stopName) y la hora de llegada en formato Unix (arrivalTimeInMillis). Esta hora se convierte a un formato legible para el usuario, como "4:30 PM", usando SimpleDateFormat.

Como almacenar datos y acceder a ellos mediante claves con DataStore

Como guardar tus preferencias de forma local con DataStore

<https://github.com/Abraham20070511/AlmacenarDataStore>

/*

* Copyright (C) 2023 The Android Open Source Project

*

* Licensed under the Apache License, Version 2.0 (the "License");

* you may not use this file except in compliance with the License.

* You may obtain a copy of the License at

*

* <https://www.apache.org/licenses/LICENSE-2.0>

*

* Unless required by applicable law or agreed to in writing, software

* distributed under the License is distributed on an "AS IS" BASIS,

* WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.

* See the License for the specific language governing permissions and

* limitations under the License.

*/

package com.example.dessertrelease.ui

```
import androidx.compose.foundation.layout.Arrangement
import androidx.compose.foundation.layout.PaddingValues
import androidx.compose.foundation.layout.fillMaxHeight
import androidx.compose.foundation.layout.fillMaxWidth
import androidx.compose.foundation.layout.height
import androidx.compose.foundation.layout.padding
import androidx.compose.foundation.layout.wrapContentHeight
import androidx.compose.foundation.lazy.LazyColumn
import androidx.compose.foundation.lazy.grid.GridCells
import androidx.compose.foundation.lazy.grid.LazyVerticalGrid
import androidx.compose.foundation.lazy.grid.items
import androidx.compose.foundation.lazy.items
import androidx.compose.material3.Card
import androidx.compose.material3.CardDefaults
import androidx.compose.material3.ExperimentalMaterial3Api
import androidx.compose.material3.Icon
import androidx.compose.material3.IconButton
import androidx.compose.material3.MaterialTheme
import androidx.compose.material3.Scaffold
import androidx.compose.material3.Text
import androidx.compose.material3.TopAppBar
import androidx.compose.material3.TopAppBarDefaults
import androidx.compose.runtime.Composable
import androidx.compose.runtime.collectAsState
import androidx.compose.ui.Alignment
import androidx.compose.ui.Modifier
import androidx.compose.ui.res.dimensionResource
import androidx.compose.ui.res.painterResource
import androidx.compose.ui.res.stringResource
import androidx.compose.ui.text.style.TextAlign
import androidx.compose.ui.text.style.TextOverflow
```

```
import androidx.compose.ui.tooling.preview.Preview
import androidx.compose.ui.unit.dp
import androidx.lifecycle.viewmodel.compose.viewModel
import com.example.dessertrelease.R
import com.example.dessertrelease.data.local.LocalDessertReleaseData
import com.example.dessertrelease.ui.theme.DessertReleaseTheme
```

```
/*
 * Screen level composable
 */
@Composable
fun DessertReleaseApp(
    dessertReleaseViewModel: DessertReleaseViewModel = viewModel(
        factory = DessertReleaseViewModel.Factory
    )
) {
    DessertReleaseScreen(
        uiState = dessertReleaseViewModel.uiState.collectAsState().value,
        selectLayout = dessertReleaseViewModel::selectLayout
    )
}
```

```
@OptIn(ExperimentalMaterial3Api::class)
@Composable
private fun DessertReleaseScreen(
    uiState: DessertReleaseUiState,
    selectLayout: (Boolean) -> Unit
) {
    val isLinearLayout = uiState.isLinearLayout
    Scaffold(
        topBar = {
            TopAppBar(
                title = { Text(stringResource(R.string.top_bar_name)) },
                actions = {
                    IconButton(
```

```

        onClick = {
            selectLayout(!isLinearLayout)
        }
    ) {
        Icon(
            painter = painterResource(uiState.toggleIcon),
            contentDescription =
stringResource(uiState.toggleContentDescription),
            tint = MaterialTheme.colorScheme.onBackground
        )
    }
},
colors = TopAppBarDefaults.largeTopAppBarColors(
    containerColor = MaterialTheme.colorScheme.inversePrimary
)
)
}
) { innerPadding ->
    val modifier = Modifier
        .padding(
            top = dimensionResource(R.dimen.padding_medium),
            start = dimensionResource(R.dimen.padding_medium),
            end = dimensionResource(R.dimen.padding_medium),
        )
    if (isLinearLayout) {
        DessertReleaseLinearLayout(
            modifier = modifier.fillMaxWidth(),
            contentPadding = innerPadding
        )
    } else {
        DessertReleaseGridLayout(
            modifier = modifier,
            contentPadding = innerPadding,
        )
    }
}

```

```

    }
}

@Composable
fun DessertReleaseLinearLayout(
    modifier: Modifier = Modifier,
    contentPadding: PaddingValues = PaddingValues(0.dp)
) {
    LazyColumn(
        modifier = modifier,
        contentPadding = contentPadding,
        verticalArrangement =
Arrangement.spacedBy(dimensionResource(R.dimen.padding_small)),
    ) {
        items(
            items = LocalDessertReleaseData.dessertReleases,
            key = { dessert -> dessert }
        ) { dessert ->
            Card(
                colors = CardDefaults.cardColors(
                    containerColor = MaterialTheme.colorScheme.primary
                ),
                shape = MaterialTheme.shapes.medium
            ) {
                Text(
                    text = dessert,
                    modifier = Modifier
                        .fillMaxWidth()
                        .padding(dimensionResource(R.dimen.padding_medium)),
                    textAlign = TextAlign.Center
                )
            }
        }
    }
}

```

```

@Composable
fun DessertReleaseGridLayout(
    modifier: Modifier = Modifier,
    contentPadding: PaddingValues = PaddingValues(0.dp)
) {
    LazyVerticalGrid(
        modifier = modifier,
        columns = GridCells.Fixed(3),
        contentPadding = contentPadding,
        verticalArrangement =
Arrangement.spacedBy(dimensionResource(R.dimen.padding_medium)),
        horizontalArrangement =
Arrangement.spacedBy(dimensionResource(R.dimen.padding_medium))
    ) {
        items(
            items = LocalDessertReleaseData.dessertReleases,
            key = { dessert -> dessert }
        ) { dessert ->
            Card(
                colors = CardDefaults.cardColors(
                    containerColor = MaterialTheme.colorScheme.primary
                ),
                modifier = Modifier.height(110.dp),
                shape = MaterialTheme.shapes.medium
            ) {
                Text(
                    text = dessert,
                    maxLines = 2,
                    overflow = TextOverflow.Ellipsis,
                    modifier = Modifier
                        .fillMaxHeight()
                        .wrapContentHeight(Alignment.CenterVertically)
                        .padding(dimensionResource(R.dimen.padding_small))
                        .align(Alignment.CenterHorizontally),
                )
            }
        }
    }
}

```



```

        textAlign = TextAlign.Center
    )
}
}
}
}

```

```

@Preview(showBackground = true)
@Composable
fun DessertReleaseLinearLayoutPreview() {
    DessertReleaseTheme {
        DessertReleaseLinearLayout()
    }
}

```

```

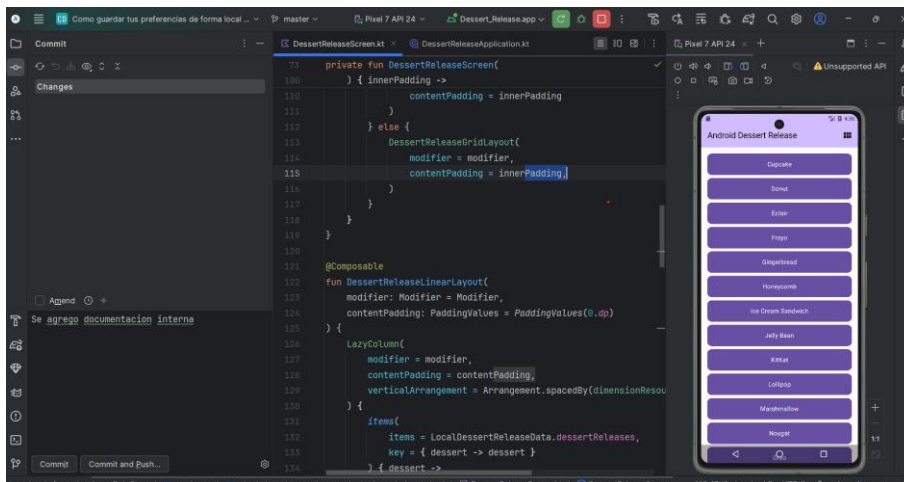
@Preview(showBackground = true)
@Composable
fun DessertReleaseGridLayoutPreview() {
    DessertReleaseTheme {
        DessertReleaseGridLayout()
    }
}

```

```

@Preview
@Composable
fun DessertReleaseAppPreview() {
    DessertReleaseTheme {
        DessertReleaseScreen(
            uiState = DessertReleaseUiState(),
            selectLayout = {}
        )
    }
}

```



La estructura principal de la app se basa en un componente `BusScheduleApp`, que configura la navegación entre pantallas utilizando `NavHost` y `NavController`. Se definen dos pantallas principales: `FullScheduleScreen` (que muestra todos los horarios disponibles) y `RouteScheduleScreen` (que muestra los horarios filtrados por una parada específica). Al seleccionar una parada desde la lista general, la aplicación navega a la pantalla correspondiente a esa parada y actualiza dinámicamente el título de la barra superior para reflejar el cambio.

El componente visual `BusScheduleScreen` se utiliza para renderizar tanto la vista completa como la vista filtrada, lo que permite reutilizar lógica y diseño. Dentro de él, se muestra una cabecera con los nombres de las columnas, una línea divisoria, y una lista eficiente (`LazyColumn`) con los horarios. Cada entrada de la lista muestra el nombre de la parada (si aplica) y la hora de llegada, que se convierte a un formato legible utilizando `SimpleDateFormat`.

La navegación está cuidadosamente controlada, incluso el botón de retroceso físico del dispositivo está manejado con `BackHandler`, lo que permite regresar a la vista anterior de forma intuitiva. Además, el componente `TopAppBar` se personaliza según el contexto: puede mostrar un botón de "volver" cuando el usuario está viendo los detalles de una parada, o simplemente el título cuando está en la vista principal.

El diseño tiene en cuenta detalles como márgenes (`PaddingValues`), tamaños de fuente, pesos para distribución del espacio, e incluso recursos localizados (`stringResource`), lo que lo hace adaptable a diferentes idiomas y configuraciones. También se incluye una vista previa para Android Studio que permite visualizar cómo se ve la aplicación sin necesidad de compilarla completamente.