Отчеты по учебной практике

Разработка Android-приложений с использованием Jetpack Compose

Table of Contents

[Задание 1: «Электронный конспект» на Jetpack Compose 1](#_Toc182069304)

[Задание 2: Архитектура MVVM на примере приложения «Список покупок» 12](#_Toc182069305)

[Задание 3: Создание приложения “Напоминалка” 23](#_Toc182069306)

[Задание 4: таймер с обратным отсчётом 34](#_Toc182069307)

[Заключение 43](#_Toc182069308)

# Задание 1: «Электронный конспект» на Jetpack Compose

В процессе разработки я изучил основные компоненты Android и Jetpack Compose, а также познакомился с библиотекой Room для хранения данных в локальной базе данных.

## Основные технологии, которые я использовал

1. Jetpack Compose — библиотека для создания пользовательского интерфейса.
2. Room — библиотека для работы с локальной базой данных.

## Шаг 1: Настройка проекта

### 1.1. Создание нового проекта

Для начала работы я: 1. Открыл Android Studio. 2. Создал новый проект с шаблоном Empty Compose Activity. 3. Назвал проект ElectronicNotes. 4. Выбрал минимальную версию SDK — Android 8.0 (Oreo) API 26.

### 1.2. Подключение библиотек

Я открыл файл build.gradle.kts моего модуля (app/build.gradle.kts) и добавил необходимые зависимости для Jetpack Compose и Room:

// Compose dependencies  
 implementation("androidx.activity:activity-compose:1.9.2")  
 implementation(platform("androidx.compose:compose-bom:2023.08.00"))  
 implementation("androidx.compose.ui:ui")  
 implementation("androidx.compose.ui:ui-graphics")  
 implementation("androidx.compose.ui:ui-tooling-preview")  
 implementation("androidx.compose.material3:material3")  
 implementation("androidx.compose.material:material-icons-extended:1.5.4")  
   
 // Room  
 val roomVersion = "2.6.0"  
 implementation("androidx.room:room-runtime:$roomVersion")  
 implementation("androidx.room:room-ktx:$roomVersion")  
 kapt("androidx.room:room-compiler:$roomVersion")  
  
 // Coroutines  
 implementation("org.jetbrains.kotlinx:kotlinx-coroutines-android:1.7.3")  
 implementation("org.jetbrains.kotlinx:kotlinx-coroutines-core:1.7.3")  
  
 // Lifecycle components  
 implementation("androidx.lifecycle:lifecycle-viewmodel-ktx:2.6.2")  
 implementation("androidx.lifecycle:lifecycle-runtime-ktx:2.6.2")  
 implementation("androidx.lifecycle:lifecycle-livedata-ktx:2.6.2")  
  
 // Jetpack Navigation for Compose  
 implementation("androidx.navigation:navigation-compose:2.5.0")

## Шаг 2: Работа с базой данных Room

### 2.1. Что такое Room?

В процессе разработки я узнал, что Room — это библиотека, которая помогает сохранять данные локально на устройстве в базе данных SQLite. Она предоставляет удобный интерфейс для создания таблиц, выполнения запросов и управления данными. Я решил использовать Room для сохранения заметок в моем приложении.

### 2.2. Создание сущности (Entity)

Для представления заметки в базе данных я создал класс Note в пакете data:

import androidx.room.Entity  
import androidx.room.PrimaryKey  
  
@Entity(tableName = "notes")  
data class Note(  
 @PrimaryKey(autoGenerate = true) val id: Int = 0,  
 val title: String,  
 val content: String,  
 val type: NoteType = NoteType.OTHER  
)

### 2.3. Создание DAO (Data Access Object)

Далее я создал интерфейс NoteDao для работы с базой данных:

import androidx.room.\*  
import kotlinx.coroutines.flow.Flow  
  
@Dao  
interface NoteDao {  
 @Query("SELECT \* FROM notes ORDER BY id DESC")  
 fun getAllNotes(): Flow<List<Note>>  
  
 @Insert(onConflict = OnConflictStrategy.REPLACE)  
 suspend fun insert(note: Note): Long  
  
 @Delete  
 suspend fun delete(note: Note): Int  
  
 @Query("SELECT \* FROM notes WHERE id = :id")  
 suspend fun getNoteById(id: Int): Note?  
  
 @Query("DELETE FROM notes")  
 suspend fun deleteAll()  
}

### 2.4. Создание базы данных

Для связи сущностей и DAO я создал класс NoteDatabase:

import android.content.Context  
import androidx.room.Database  
import androidx.room.Room  
import androidx.room.RoomDatabase  
import androidx.room.TypeConverters  
import com.example.electronicnotes.data.Note  
import com.example.electronicnotes.data.NoteDao  
  
// Определение базы данных с помощью аннотации @Database  
@Database(entities = [Note::class], version = 2, exportSchema = false)  
@TypeConverters(Converters::class)  
abstract class NoteDatabase : RoomDatabase() {  
 // Абстрактная функция для получения объекта NoteDao  
 abstract fun noteDao(): NoteDao  
  
 // Компаньон-объект для доступа к базе данных  
 companion object {  
 // Волатильная переменная для хранения единственного экземпляра базы данных  
 @Volatile  
 private var INSTANCE: NoteDatabase? = null  
  
 // Функция для получения экземпляра базы данных  
 fun getDatabase(context: Context): NoteDatabase {  
 // Если экземпляр базы данных уже существует, возвращаем его  
 return INSTANCE ?: synchronized(this) {  
 // Если экземпляр базы данных еще не создан, создаем его  
 val instance = Room.databaseBuilder(  
 context.applicationContext,  
 NoteDatabase::class.java,  
 "note\_database"  
 )  
 .fallbackToDestructiveMigration() // При изменении схемы БД пересоздаст её  
 .build()  
 INSTANCE = instance  
 instance  
 }  
 }  
 }  
}

### 2.5. Мои наблюдения

В процессе работы я понял, что: - Entity — это модель данных, представляющая таблицу в базе данных. - DAO — это интерфейс для взаимодействия с базой данных. - Database — это класс, связывающий сущности и DAO.

## Шаг 3: Создание пользовательского интерфейса (UI) с Jetpack Compose

### 3.1. Экран списка заметок

Я разработал экран для отображения списка всех заметок:

@OptIn(ExperimentalMaterial3Api::class)  
@Composable  
fun NotesListScreen(  
 notes: List<Note>,  
 onAddNote: () -> Unit,  
 onNoteClick: (Note) -> Unit,  
 noteDao: NoteDao  
) {  
 val scope = rememberCoroutineScope()  
 var showDeleteDialog by remember { mutableStateOf<Note?>(null) }  
  
 showDeleteDialog?.let { note ->  
 AlertDialog(  
 onDismissRequest = { showDeleteDialog = null },  
 title = { Text("Удалить заметку?") },  
 text = { Text("Вы уверены, что хотите удалить заметку \"${note.title}\"?") },  
 confirmButton = {  
 TextButton(  
 onClick = {  
 scope.launch {  
 withContext(Dispatchers.IO) {  
 noteDao.delete(note)  
 }  
 showDeleteDialog = null  
 }  
 }  
 ) {  
 Text("Удалить", color = MaterialTheme.colorScheme.error)  
 }  
 },  
 dismissButton = {  
 TextButton(onClick = { showDeleteDialog = null }) {  
 Text("Отмена")  
 }  
 }  
 )  
 }  
  
 Scaffold(  
 topBar = {  
 LargeTopAppBar(  
 title = {   
 Text(  
 "Мои заметки",  
 style = MaterialTheme.typography.headlineMedium  
 )  
 },  
 colors = TopAppBarDefaults.largeTopAppBarColors(  
 containerColor = MaterialTheme.colorScheme.primary,  
 titleContentColor = MaterialTheme.colorScheme.onPrimary,  
 navigationIconContentColor = MaterialTheme.colorScheme.onPrimary,  
 actionIconContentColor = MaterialTheme.colorScheme.onPrimary  
 )  
 )  
 },  
 floatingActionButton = {  
 ExtendedFloatingActionButton(  
 onClick = onAddNote,  
 containerColor = MaterialTheme.colorScheme.secondary,  
 contentColor = MaterialTheme.colorScheme.onSecondary,  
 icon = { Icon(Icons.Default.Edit, "Добавить") },  
 text = { Text("Новая заметка") }  
 )  
 }  
 ) { paddingValues ->  
 if (notes.isEmpty()) {  
 Box(  
 modifier = Modifier  
 .fillMaxSize()  
 .padding(paddingValues),  
 contentAlignment = Alignment.Center  
 ) {  
 Text(  
 text = "Нет заметок\nНажмите '+' чтобы добавить",  
 style = MaterialTheme.typography.bodyLarge,  
 textAlign = TextAlign.Center,  
 color = MaterialTheme.colorScheme.onBackground.copy(alpha = 0.6f)  
 )  
 }  
 } else {  
 LazyColumn(  
 modifier = Modifier  
 .fillMaxSize()  
 .padding(paddingValues),  
 contentPadding = PaddingValues(16.dp),  
 verticalArrangement = Arrangement.spacedBy(8.dp)  
 ) {  
 items(  
 items = notes,  
 key = { note -> note.id }  
 ) { note ->  
 val dismissState = rememberDismissState(  
 initialValue = DismissValue.Default,  
 confirmValueChange = { dismissValue ->  
 if (dismissValue != DismissValue.Default) {  
 showDeleteDialog = note  
 }  
 false  
 },  
 positionalThreshold = { density -> density \* 56f }  
 )  
  
 SwipeToDismiss(  
 state = dismissState,  
 background = {  
 val color = Color.Red.copy(alpha = 0.8f)  
 val direction = dismissState.dismissDirection  
  
 Row(  
 modifier = Modifier  
 .fillMaxSize()  
 .padding(horizontal = 20.dp),  
 horizontalArrangement = if (direction == DismissDirection.StartToEnd)   
 Arrangement.Start else Arrangement.End,  
 verticalAlignment = Alignment.CenterVertically  
 ) {  
 Icon(  
 Icons.Default.Delete,  
 contentDescription = "Удалить",  
 tint = Color.White  
 )  
 }  
 },  
 dismissContent = {  
 AnimatedVisibility(  
 visible = true,  
 enter = expandVertically() + fadeIn(),  
 exit = shrinkVertically() + fadeOut()  
 ) {  
 NoteCard(  
 note = note,  
 onClick = { onNoteClick(note) }  
 )  
 }  
 }  
 )  
 }  
 }  
 }  
 }  
}

### 3.2. Экран создания/редактирования заметки

Затем я создал экран для добавления и редактирования заметок:

@OptIn(ExperimentalMaterial3Api::class)  
@Composable  
fun AddEditNoteScreen(  
 noteDao: NoteDao,  
 onNoteSaved: () -> Unit,  
 existingNote: Note? = null  
) {  
 var title by remember { mutableStateOf(existingNote?.title ?: "") }  
 var content by remember { mutableStateOf(existingNote?.content ?: "") }  
 var selectedType by remember { mutableStateOf(existingNote?.type ?: NoteType.OTHER) }  
 var showTypeDialog by remember { mutableStateOf(false) }  
 val scope = rememberCoroutineScope()  
  
 if (showTypeDialog) {  
 AlertDialog(  
 onDismissRequest = { showTypeDialog = false },  
 title = { Text("Выберите тип заметки") },  
 text = {  
 LazyColumn {  
 items(NoteType.values()) { type ->  
 ListItem(  
 headlineContent = { Text(type.label) },  
 leadingContent = {  
 Icon(  
 imageVector = type.icon,  
 contentDescription = null,  
 tint = MaterialTheme.colorScheme.primary  
 )  
 },  
 modifier = Modifier.clickable {  
 selectedType = type  
 showTypeDialog = false  
 }  
 )  
 }  
 }  
 },  
 confirmButton = {  
 TextButton(onClick = { showTypeDialog = false }) {  
 Text("Отмена")  
 }  
 }  
 )  
 }  
  
 Scaffold(  
 topBar = {  
 TopAppBar(  
 title = { Text(if (existingNote == null) "Новая заметка" else "Редактирование") },  
 navigationIcon = {  
 IconButton(onClick = { onNoteSaved() }) {  
 Icon(Icons.Default.ArrowBack, contentDescription = "Назад")  
 }  
 },  
 colors = TopAppBarDefaults.topAppBarColors(  
 containerColor = MaterialTheme.colorScheme.primary,  
 titleContentColor = MaterialTheme.colorScheme.onPrimary,  
 navigationIconContentColor = MaterialTheme.colorScheme.onPrimary  
 )  
 )  
 },  
 floatingActionButton = {  
 FloatingActionButton(  
 onClick = {  
 scope.launch {  
 withContext(Dispatchers.IO) {  
 val note = existingNote?.copy(  
 title = title,  
 content = content,  
 type = selectedType  
 ) ?: Note(  
 title = title,  
 content = content,  
 type = selectedType  
 )  
 noteDao.insert(note)  
 }  
 onNoteSaved()  
 }  
 },  
 containerColor = MaterialTheme.colorScheme.secondary  
 ) {  
 Text(  
 "Сохранить",  
 color = MaterialTheme.colorScheme.onSecondary,  
 modifier = Modifier.padding(horizontal = 16.dp)  
 )  
 }  
 }  
 ) { paddingValues ->  
 Column(  
 modifier = Modifier  
 .fillMaxSize()  
 .padding(paddingValues)  
 .padding(16.dp),  
 verticalArrangement = Arrangement.spacedBy(8.dp)  
 ) {  
 OutlinedTextField(  
 value = title,  
 onValueChange = { title = it },  
 label = { Text("Заголовок") },  
 modifier = Modifier.fillMaxWidth(),  
 colors = TextFieldDefaults.outlinedTextFieldColors(  
 focusedBorderColor = MaterialTheme.colorScheme.primary,  
 focusedLabelColor = MaterialTheme.colorScheme.primary  
 )  
 )  
   
 OutlinedButton(  
 onClick = { showTypeDialog = true },  
 modifier = Modifier.fillMaxWidth()  
 ) {  
 Icon(  
 imageVector = selectedType.icon,  
 contentDescription = null,  
 modifier = Modifier.size(24.dp)  
 )  
 Spacer(Modifier.width(8.dp))  
 Text(selectedType.label)  
 }  
   
 OutlinedTextField(  
 value = content,  
 onValueChange = { content = it },  
 label = { Text("Содержание") },  
 modifier = Modifier  
 .fillMaxWidth()  
 .weight(1f),  
 colors = TextFieldDefaults.outlinedTextFieldColors(  
 focusedBorderColor = MaterialTheme.colorScheme.primary,  
 focusedLabelColor = MaterialTheme.colorScheme.primary  
 ),  
 minLines = 5  
 )  
 }  
 }  
}

### 3.3. Основной экран приложения

После этого я объединил всё в MainActivity:

class MainActivity : ComponentActivity() {  
 @SuppressLint("StateFlowValueCalledInComposition")  
 override fun onCreate(savedInstanceState: Bundle?) {  
 super.onCreate(savedInstanceState)  
   
 val database = NoteDatabase.getDatabase(this)  
 val noteDao = database.noteDao()  
  
 setContent {  
 val navController = rememberNavController()  
 var notes by remember { mutableStateOf<List<Note>>(emptyList()) }  
   
 // Собираем Flow из базы данных  
 LaunchedEffect(Unit) {  
 noteDao.getAllNotes().collectLatest { notesList ->  
 notes = notesList  
 }  
 }  
   
 ElectronicNotesTheme {  
 Surface(  
 modifier = Modifier.fillMaxSize(),  
 color = MaterialTheme.colorScheme.background  
 ) {  
 NavHost(  
 navController = navController,  
 startDestination = "notes\_list"  
 ) {  
 composable("notes\_list") {  
 NotesListScreen(  
 notes = notes,  
 onAddNote = {  
 navController.navigate("add\_edit\_note")  
 },  
 onNoteClick = { note ->  
 navController.navigate("add\_edit\_note?noteId=${note.id}")  
 },  
 noteDao = noteDao  
 )  
 }  
  
 composable(  
 route = "add\_edit\_note?noteId={noteId}",  
 arguments = listOf(  
 navArgument("noteId") {  
 type = NavType.IntType  
 defaultValue = -1  
 }  
 )  
 ) { backStackEntry ->  
 val noteId = backStackEntry.arguments?.getInt("noteId") ?: -1  
 val existingNote = notes.find { it.id == noteId }  
  
 AddEditNoteScreen(  
 noteDao = noteDao,  
 existingNote = existingNote,  
 onNoteSaved = {  
 navController.popBackStack()  
 }  
 )  
 }  
 }  
 }  
 }  
 }  
 }  
}

## Дополнительные задания, которые я выполнил

### 1. Просмотр добавленных заметок

Реализовал функционал просмотра заметок со следующими возможностями: - Отображение списка всех заметок в виде карточек - Сортировка заметок по дате создания (новые сверху) - Отображение типа заметки с помощью иконок - Предпросмотр содержимого заметки в списке - Пустое состояние с подсказкой при отсутствии заметок

### 2. Удаление заметок

Добавил интуитивный механизм удаления заметок: - Свайп влево/вправо для удаления заметки - Диалог подтверждения удаления - Анимация удаления - Возможность отменить действие - Визуальная индикация при свайпе (красный фон и иконка удаления)

### 3. Редактирование заметок

Реализовал полноценное редактирование существующих заметок: - Редактирование заголовка и содержимого - Изменение типа заметки - Автоматическое сохранение изменений - Возможность отмены редактирования - Сохранение времени последнего редактирования

### 4. Улучшение дизайна интерфейса

Значительно улучшил визуальную составляющую приложения: - Добавил Material Design 3 цветовую схему - Реализовал темную и светлую темы - Добавил анимации переходов между экранами - Использовал иконки Material Icons - Применил современные компоненты Material Design: - Extended FAB для создания заметок - Swipe-to-dismiss для удаления - Красивые карточки для заметок - Адаптивные отступы и типографику

### 5. Дополнительные улучшения

* Добавил категоризацию заметок по типам
* Реализовал сохранение состояния при повороте экрана
* Оптимизировал производительность списка с помощью LazyColumn
* Добавил обработку ошибок при работе с базой данных
* Реализовал плавные анимации при добавлении/удалении заметок

## Заключение

В ходе учебной практики я успешно разработал простой прототип приложения для создания и управления заметками. Я освоил работу с Jetpack Compose и Room, изучил основные компоненты Android, научился работать с локальной базой данных и создавать современный пользовательский интерфейс. Этот опыт дал мне хорошее понимание основ разработки Android-приложений.

# Задание 2: Архитектура MVVM на примере приложения «Список покупок»

В этом отчете по учебной практике я рассмотрел применение архитектуры MVVM на примере простого приложения для управления списком покупок. Я продолжил изучение Jetpack Compose и познакомился с новым компонентом — ViewModel.

## Шаг 1: Что такое MVVM?

Я изучил, что MVVM (Model-View-ViewModel) — это архитектурный паттерн, который разделяет приложение на три основных компонента:

* Model — управляет данными. Это могут быть данные из базы данных, сети или локальные данные.
* View — пользовательский интерфейс (UI), с которым взаимодействует пользователь.
* ViewModel — управляет состоянием, обрабатывает действия пользователя, взаимодействует с Model и предоставляет данные для View.

## Шаг 2: Подключаем зависимости

Я добавил в build.gradle.kts зависимости для работы с ViewModel, Room и LiveData:

// Core Android dependencies  
 implementation("androidx.core:core-ktx:1.12.0")  
 implementation("androidx.lifecycle:lifecycle-runtime-ktx:2.7.0")  
 implementation("androidx.lifecycle:lifecycle-viewmodel-compose:2.7.0")  
   
 // Material Icons Extended  
 implementation("androidx.compose.material:material-icons-extended")  
  
 // Добавляем Compose Navigation  
 implementation("androidx.navigation:navigation-compose:2.7.7")  
   
 // Добавляем Compose Runtime  
 implementation("androidx.compose.runtime:runtime")  
 implementation("androidx.compose.runtime:runtime-livedata")  
  
 // Room  
 val roomVersion = "2.6.1"  
 implementation("androidx.room:room-runtime:$roomVersion")  
 implementation("androidx.room:room-ktx:$roomVersion")   
 kapt("androidx.room:room-compiler:$roomVersion")  
  
 // Coroutines  
 implementation("org.jetbrains.kotlinx:kotlinx-coroutines-android:1.7.3")  
 implementation("org.jetbrains.kotlinx:kotlinx-coroutines-core:1.7.3")

## Шаг 3: Создание простого ViewModel для списка покупок

Я создал приложение, в котором пользователь может: 1. Добавлять товары в список покупок 2. Удалять товары из списка 3. Редактировать существующие товары 4. Отмечать товары как купленные 5. Сортировать список по разным критериям

### 3.1. Создаём ViewModel

import android.app.Application  
import androidx.lifecycle.AndroidViewModel  
import androidx.lifecycle.viewModelScope  
import androidx.compose.runtime.mutableStateOf  
import androidx.compose.runtime.State  
import com.example.foodler.model.ShoppingItem  
import com.example.foodler.model.SortType  
import com.example.foodler.data.ShoppingDatabase  
import com.example.foodler.data.ShoppingRepository  
import kotlinx.coroutines.flow.collect  
import kotlinx.coroutines.launch  
  
class ShoppingListViewModel(application: Application) : AndroidViewModel(application) {  
 private val repository: ShoppingRepository  
   
 // Состояние для списка покупок  
 private val \_items = mutableStateOf<List<ShoppingItem>>(emptyList())  
 val items: State<List<ShoppingItem>> = \_items  
  
 // Состояние для поля ввода нового товара  
 private val \_newItemName = mutableStateOf("")  
 val newItemName: State<String> = \_newItemName  
  
 // Состояние для редактирования  
 private val \_editingItem = mutableStateOf<ShoppingItem?>(null)  
 val editingItem: State<ShoppingItem?> = \_editingItem  
  
 // Состояние для типа сортировки  
 private val \_sortType = mutableStateOf(SortType.CREATED)  
 val sortType: State<SortType> = \_sortType  
  
 init {  
 val database = ShoppingDatabase.getDatabase(application)  
 repository = ShoppingRepository(database.shoppingDao())  
   
 // Наблюдаем за изменениями в базе данных  
 viewModelScope.launch {  
 repository.allItems.collect { newItems ->  
 \_items.value = sortItems(newItems)  
 }  
 }  
 }  
  
 fun updateNewItemName(name: String) {  
 \_newItemName.value = name  
 }  
  
 fun addItem(name: String) {  
 if (name.isBlank()) return  
 viewModelScope.launch {  
 repository.insertItem(  
 ShoppingItem(  
 name = name.trim()  
 )  
 )  
 }  
 \_newItemName.value = ""  
 }  
  
 fun removeItem(item: ShoppingItem) {  
 viewModelScope.launch {  
 repository.deleteItem(item)  
 }  
 }  
  
 fun updateItem() {  
 val editingItem = \_editingItem.value ?: return  
 viewModelScope.launch {  
 repository.updateItem(  
 editingItem.copy(name = \_newItemName.value.trim())  
 )  
 }  
 cancelEditing()  
 }  
  
 fun toggleItemCheck(item: ShoppingItem) {  
 viewModelScope.launch {  
 repository.updateItem(item.copy(isChecked = !item.isChecked))  
 }  
 }  
  
 // Новые функции для редактирования  
 fun startEditing(item: ShoppingItem) {  
 \_editingItem.value = item  
 \_newItemName.value = item.name  
 }  
  
 fun cancelEditing() {  
 \_editingItem.value = null  
 \_newItemName.value = ""  
 }  
  
 // Функции для сортировки  
 fun changeSortType(type: SortType) {  
 \_sortType.value = type  
 \_items.value = sortItems(\_items.value)  
 }  
  
 private fun sortItems(items: List<ShoppingItem>): List<ShoppingItem> {  
 return when (\_sortType.value) {  
 SortType.NAME -> items.sortedBy { it.name }  
 SortType.CREATED -> items.sortedBy { it.createdAt }  
 SortType.CHECKED -> items.sortedBy { it.isChecked }  
 }  
 }  
}

В процессе разработки я использовал: - \_items — это список товаров. Я использовал mutableStateOf для отслеживания состояния списка покупок. - \_newItemName — состояние для поля ввода нового товара. - \_editingItem — состояние для редактируемого товара. - \_sortType — состояние для текущего типа сортировки.

Каждое состояние имеет публичную версию только для чтения через тип State<T>, что обеспечивает инкапсуляцию и безопасность данных: - items: State<List<ShoppingItem>> - newItemName: State<String> - editingItem: State<ShoppingItem?> - sortType: State<SortType>

Основные функции ViewModel: - addItem() — добавление нового товара в список - removeItem() — удаление товара из списка - updateItem() — обновление существующего товара - toggleItemCheck() — отметка товара как купленного/не купленного - changeSortType() — изменение способа сортировки списка

### 3.2. Использование ViewModel в Jetpack Compose

import androidx.compose.foundation.layout.\*  
import androidx.compose.foundation.lazy.LazyColumn  
import androidx.compose.foundation.lazy.items  
import androidx.compose.material.icons.Icons  
import androidx.compose.material.icons.filled.Add  
import androidx.compose.material.icons.filled.Delete  
import androidx.compose.material.icons.filled.Edit  
import androidx.compose.material.icons.filled.Sort  
import androidx.compose.material3.\*  
import androidx.compose.runtime.\*  
import androidx.compose.ui.Alignment  
import androidx.compose.ui.Modifier  
import androidx.compose.ui.unit.dp  
import com.example.foodler.model.ShoppingItem  
import com.example.foodler.model.SortType  
import com.example.foodler.viewmodel.ShoppingListViewModel  
  
@Composable  
fun ShoppingListScreen(  
 viewModel: ShoppingListViewModel,  
 modifier: Modifier = Modifier  
) {  
 Column(  
 modifier = modifier  
 .fillMaxSize()  
 .padding(16.dp)  
 ) {  
 // Добавляем меню сортировки  
 SortMenu(  
 currentSort = viewModel.sortType.value,  
 onSortChange = { viewModel.changeSortType(it) }  
 )  
  
 Spacer(modifier = Modifier.height(8.dp))  
  
 AddItemSection(  
 value = viewModel.newItemName.value,  
 onValueChange = { viewModel.updateNewItemName(it) },  
 onAddClick = {  
 if (viewModel.editingItem.value != null) {  
 viewModel.updateItem()  
 } else {  
 viewModel.addItem(viewModel.newItemName.value)  
 }  
 },  
 isEditing = viewModel.editingItem.value != null,  
 onCancelEdit = { viewModel.cancelEditing() }  
 )  
  
 Spacer(modifier = Modifier.height(16.dp))  
  
 ShoppingList(  
 items = viewModel.items.value,  
 onCheckedChange = { viewModel.toggleItemCheck(it) },  
 onDeleteClick = { viewModel.removeItem(it) },  
 onEditClick = { viewModel.startEditing(it) }  
 )  
 }  
}  
  
@Composable  
fun SortMenu(  
 currentSort: SortType,  
 onSortChange: (SortType) -> Unit,  
 modifier: Modifier = Modifier  
) {  
 var expanded by remember { mutableStateOf(false) }  
  
 Row(  
 modifier = modifier.fillMaxWidth(),  
 horizontalArrangement = Arrangement.End  
 ) {  
 IconButton(onClick = { expanded = true }) {  
 Icon(Icons.Default.Sort, "Сортировка")  
 }  
 DropdownMenu(  
 expanded = expanded,  
 onDismissRequest = { expanded = false }  
 ) {  
 DropdownMenuItem(  
 text = { Text("По алфавиту") },  
 onClick = {   
 onSortChange(SortType.NAME)  
 expanded = false  
 }  
 )  
 DropdownMenuItem(  
 text = { Text("По дате добавления") },  
 onClick = {   
 onSortChange(SortType.CREATED)  
 expanded = false  
 }  
 )  
 DropdownMenuItem(  
 text = { Text("По статусу") },  
 onClick = {   
 onSortChange(SortType.CHECKED)  
 expanded = false  
 }  
 )  
 }  
 }  
}  
  
@Composable  
fun AddItemSection(  
 value: String,  
 onValueChange: (String) -> Unit,  
 onAddClick: () -> Unit,  
 isEditing: Boolean,  
 onCancelEdit: () -> Unit,  
 modifier: Modifier = Modifier  
) {  
 Row(  
 modifier = modifier.fillMaxWidth(),  
 verticalAlignment = Alignment.CenterVertically  
 ) {  
 OutlinedTextField(  
 value = value,  
 onValueChange = onValueChange,  
 modifier = Modifier.weight(1f),  
 label = { Text(if (isEditing) "Редактировать товар" else "Добавить товар") },  
 singleLine = true  
 )  
 Spacer(modifier = Modifier.width(8.dp))  
 if (isEditing) {  
 IconButton(onClick = onCancelEdit) {  
 Icon(Icons.Default.Delete, "Отменить")  
 }  
 }  
 IconButton(onClick = onAddClick) {  
 Icon(  
 if (isEditing) Icons.Default.Edit else Icons.Default.Add,  
 if (isEditing) "Сохранить" else "Добавить"  
 )  
 }  
 }  
}  
  
@Composable  
fun ShoppingList(  
 items: List<ShoppingItem>,  
 onCheckedChange: (ShoppingItem) -> Unit,  
 onDeleteClick: (ShoppingItem) -> Unit,  
 onEditClick: (ShoppingItem) -> Unit,  
 modifier: Modifier = Modifier  
) {  
 LazyColumn(  
 modifier = modifier.fillMaxWidth(),  
 verticalArrangement = Arrangement.spacedBy(8.dp)  
 ) {  
 items(items) { item ->  
 ShoppingItemRow(  
 item = item,  
 onCheckedChange = { onCheckedChange(item) },  
 onDeleteClick = { onDeleteClick(item) },  
 onEditClick = { onEditClick(item) }  
 )  
 }  
 }  
}  
  
@Composable  
fun ShoppingItemRow(  
 item: ShoppingItem,  
 onCheckedChange: () -> Unit,  
 onDeleteClick: () -> Unit,  
 onEditClick: () -> Unit,  
 modifier: Modifier = Modifier  
) {  
 Card(  
 modifier = modifier.fillMaxWidth(),  
 ) {  
 Row(  
 modifier = Modifier  
 .fillMaxWidth()  
 .padding(8.dp),  
 horizontalArrangement = Arrangement.SpaceBetween,  
 verticalAlignment = Alignment.CenterVertically  
 ) {  
 Row(  
 verticalAlignment = Alignment.CenterVertically  
 ) {  
 Checkbox(  
 checked = item.isChecked,  
 onCheckedChange = { onCheckedChange() }  
 )  
 Spacer(modifier = Modifier.width(8.dp))  
 Text(  
 text = item.name,  
 style = MaterialTheme.typography.bodyLarge  
 )  
 }  
 Row {  
 IconButton(onClick = onEditClick) {  
 Icon(Icons.Default.Edit, "Редактировать")  
 }  
 IconButton(onClick = onDeleteClick) {  
 Icon(Icons.Default.Delete, "Удалить")  
 }  
 }  
 }  
 }  
}

Я создал интерфейс для взаимодействия с моим списком покупок, используя Jetpack Compose. Основной экран приложения реализован в компоненте ShoppingListScreen, который содержит:

* Меню сортировки (SortMenu) для выбора способа сортировки списка
* Секцию добавления товара (AddItemSection) с текстовым полем и кнопками
* Список покупок (ShoppingList) с возможностью отмечать, редактировать и удалять товары

Каждый элемент списка отображается в виде карточки (ShoppingItemRow) с чекбоксом и кнопками управления.

Для управления состоянием используется ViewModel, который предоставляет все необходимые методы для работы со списком:

* Добавление новых товаров через addItem()
* Удаление товаров через removeItem()
* Редактирование через updateItem()
* Отметка о покупке через toggleItemCheck()
* Сортировка через changeSortType()

Все изменения состояния автоматически отражаются в UI благодаря использованию State в Compose.

## Шаг 4: Жизненный цикл ViewModel

Я убедился, что ViewModel сохраняет состояние даже при изменении конфигурации (например, повороте экрана). Когда я добавлял товар в список и поворачивал экран — список оставался неизменным благодаря ViewModel.

## Задания для студентов

1. Я изучил пример: Как ViewModel управляет состоянием и как изменения состояния отражаются на UI.
2. Я расширил приложение: Добавил функцию редактирования товаров:

В ShoppingListViewModel.kt я добавил: - Состояние \_editingItem для хранения редактируемого товара - Функцию startEditing() для начала редактирования - Функцию updateItem() для сохранения изменений - Функцию cancelEditing() для отмены редактирования

В ShoppingListScreen.kt я реализовал: - Отображение кнопки редактирования для каждого товара - Изменение поля ввода при редактировании - Кнопки для сохранения/отмены изменений - Обработку нажатий через ViewModel

1. Я реализовал сортировку списка: Добавил возможность сортировать список по алфавиту, дате добавления и статусу выполнения.

В ShoppingListViewModel.kt я добавил: - Состояние \_sortType для хранения текущего типа сортировки - Функцию changeSortType() для изменения сортировки - Функцию sortItems() для применения сортировки к списку

В ShoppingListScreen.kt я реализовал: - Компонент SortMenu с выпадающим списком типов сортировки - Обработку выбора типа сортировки через ViewModel - Автоматическое обновление списка при смене сортировки

1. Я внедрил кэширование данных: Реализовал сохранение списка покупок в базе данных с помощью Room:

В ShoppingDatabase.kt я добавил:

@Database(entities = [ShoppingItemEntity::class], version = 1)  
abstract class ShoppingDatabase : RoomDatabase() {  
 abstract fun shoppingDao(): ShoppingDao  
  
 companion object {  
 @Volatile  
 private var INSTANCE: ShoppingDatabase? = null  
  
 fun getDatabase(context: Context): ShoppingDatabase {  
 return INSTANCE ?: synchronized(this) {  
 val instance = Room.databaseBuilder(  
 context.applicationContext,  
 ShoppingDatabase::class.java,  
 "shopping\_database"  
 ).build()  
 INSTANCE = instance  
 instance  
 }  
 }  
 }  
}

В ShoppingDao.kt я реализовал методы для работы с базой данных:

@Dao  
interface ShoppingDao {  
 @Query("SELECT \* FROM shopping\_items")  
 fun getAllItems(): Flow<List<ShoppingItemEntity>>  
  
 @Insert(onConflict = OnConflictStrategy.REPLACE)  
 suspend fun insertItem(item: ShoppingItemEntity)  
  
 @Delete  
 suspend fun deleteItem(item: ShoppingItemEntity)  
  
 @Update  
 suspend fun updateItem(item: ShoppingItemEntity)  
}

В ShoppingItem.kt я определил модель данных:

data class ShoppingItem(  
 val id: Int = 0,  
 val name: String,  
 val quantity: Int = 1,  
 val isChecked: Boolean = false,  
 val createdAt: Date = Date()  
)

В ShoppingItemEntity.kt я создал сущность для базы данных:

@Entity(tableName = "shopping\_items")  
data class ShoppingItemEntity(  
 @PrimaryKey(autoGenerate = true)  
 val id: Int = 0,  
 val name: String,  
 val quantity: Int,  
 val isChecked: Boolean,  
 val createdAt: Long  
) {  
 fun toShoppingItem(): ShoppingItem {  
 return ShoppingItem(  
 id = id,  
 name = name,  
 quantity = quantity,  
 isChecked = isChecked,  
 createdAt = Date(createdAt)  
 )  
 }  
}  
  
fun ShoppingItem.toEntity(): ShoppingItemEntity {  
 return ShoppingItemEntity(  
 id = id,  
 name = name,  
 quantity = quantity,  
 isChecked = isChecked,  
 createdAt = createdAt.time  
 )  
}

В ShoppingRepository.kt я реализовал репозиторий для работы с данными:

class ShoppingRepository(private val shoppingDao: ShoppingDao) {  
 val allItems: Flow<List<ShoppingItem>> = shoppingDao.getAllItems()  
 .map { items -> items.map { it.toShoppingItem() } }  
  
 suspend fun insertItem(item: ShoppingItem) {  
 shoppingDao.insertItem(item.toEntity())  
 }  
  
 suspend fun deleteItem(item: ShoppingItem) {  
 shoppingDao.deleteItem(item.toEntity())  
 }  
  
 suspend fun updateItem(item: ShoppingItem) {  
 shoppingDao.updateItem(item.toEntity())  
 }  
}

Это позволило реализовать полноценное локальное хранение данных с возможностью их получения, добавления, обновления и удаления, а также правильное преобразование между сущностями базы данных и моделями приложения.

# Задание 3: Создание приложения “Напоминалка”

В данном отчете я рассмотрю процесс создания приложения для управления напоминаниями с использованием уведомлений.

Для добавления возможности показа уведомлений в назначенную дату в моем приложении на Jetpack Compose мне необходимо было использовать AlarmManager, BroadcastReceiver и систему уведомлений Android.

## Шаги реализации:

### Шаг 1: Создание BroadcastReceiver для отображения уведомлений

Создадим NotificationReceiver, который будет получать сигнал от AlarmManager и показывать уведомление:

class AlarmReceiver : BroadcastReceiver() {  
 override fun onReceive(context: Context, intent: Intent) {  
 // Get alarm details from intent  
 val alarmId = intent.getIntExtra("ALARM\_ID", 0)  
 val title = intent.getStringExtra("TITLE") ?: "Напоминание"  
 val description = intent.getStringExtra("DESCRIPTION") ?: "Пора выполнить задачу!"  
  
 // Get notification manager service  
 val notificationManager =   
 context.getSystemService(Context.NOTIFICATION\_SERVICE) as NotificationManager  
  
 // Create notification channel for Android O and above  
 if (Build.VERSION.SDK\_INT >= Build.VERSION\_CODES.O) {  
 val channelDescription = "Канал для уведомлений о задачах"  
 val channel = NotificationChannel(  
 "task\_channel",  
 "Задачи",  
 NotificationManager.IMPORTANCE\_HIGH  
 ).apply {  
 setDescription(channelDescription)  
 enableVibration(true)  
 enableLights(true)  
 }  
 notificationManager.createNotificationChannel(channel)  
 }  
  
 // Build and show notification  
 val notification = NotificationCompat.Builder(context, "task\_channel")  
 .setSmallIcon(R.drawable.ic\_launcher\_foreground)  
 .setContentTitle(title)  
 .setContentText(description)  
 .setPriority(NotificationCompat.PRIORITY\_HIGH)  
 .setCategory(NotificationCompat.CATEGORY\_ALARM)  
 .setAutoCancel(true)  
 .setDefaults(NotificationCompat.DEFAULT\_ALL)  
 .build()  
  
 notificationManager.notify(alarmId, notification)  
 }  
}

Зарегистрировали NotificationReceiver в AndroidManifest.xml:

<?xml version="1.0" encoding="utf-8"?>  
<manifest xmlns:android="http://schemas.android.com/apk/res/android"  
 xmlns:tools="http://schemas.android.com/tools">  
  
 <uses-permission android:name="android.permission.POST\_NOTIFICATIONS" />  
 <uses-permission android:name="android.permission.SCHEDULE\_EXACT\_ALARM" />  
  
 <application  
 android:allowBackup="true"  
 android:dataExtractionRules="@xml/data\_extraction\_rules"  
 android:fullBackupContent="@xml/backup\_rules"  
 android:icon="@drawable/alert"  
 android:label="@string/app\_name"  
 android:roundIcon="@drawable/alert"  
 android:supportsRtl="true"  
 android:theme="@style/Theme.Alertio"  
 tools:targetApi="31">  
   
 <receiver android:name=".AlarmReceiver" />  
   
 ...  
 </application>  
  
</manifest>

### Шаг 2: Настройка AlarmManager для показа уведомлений

Когда пользователь создаёт задачу, установите будильник для уведомления на указанное время.

fun setAlarm(context: Context, alarmId: Int, timeInMillis: Long, title: String, description: String) {  
 val intent = Intent(context, AlarmReceiver::class.java).apply {  
 putExtra("ALARM\_ID", alarmId)  
 putExtra("TITLE", title)  
 putExtra("DESCRIPTION", description)  
 }  
   
 val pendingIntent = PendingIntent.getBroadcast(  
 context,  
 alarmId,  
 intent,  
 PendingIntent.FLAG\_IMMUTABLE or PendingIntent.FLAG\_UPDATE\_CURRENT  
 )  
   
 val alarmManager = context.getSystemService(Context.ALARM\_SERVICE) as AlarmManager  
   
 if (Build.VERSION.SDK\_INT >= Build.VERSION\_CODES.S) {  
 if (alarmManager.canScheduleExactAlarms()) {  
 scheduleExactAlarm(alarmManager, timeInMillis, pendingIntent)  
 } else {  
 // Запрашиваем разрешение на точные уведомления для Android 12+  
 val intent = Intent(Settings.ACTION\_REQUEST\_SCHEDULE\_EXACT\_ALARM)  
 context.startActivity(intent)  
 }  
 } else {  
 scheduleExactAlarm(alarmManager, timeInMillis, pendingIntent)  
 }  
 }

Здесь alarmId используется как уникальный идентификатор для каждого уведомления.

### Шаг 3: Отмена уведомления при удалении задачи

Когда пользователь удаляет задачу, отмените соответствующий будильник.

fun cancelAlarm(context: Context, alarmId: Int) {  
 val intent = Intent(context, AlarmReceiver::class.java).apply {  
 putExtra("ALARM\_ID", alarmId)  
 }  
   
 val pendingIntent = PendingIntent.getBroadcast(  
 context,  
 alarmId,  
 intent,  
 PendingIntent.FLAG\_IMMUTABLE or PendingIntent.FLAG\_UPDATE\_CURRENT  
 )  
   
 val alarmManager = context.getSystemService(Context.ALARM\_SERVICE) as AlarmManager  
 alarmManager.cancel(pendingIntent)  
 pendingIntent.cancel()  
 }

### Шаг 4: Интеграция с базой данных и UI

## Основные компоненты

### 1. Создание сущности (Entity)

Для начала я создал класс TaskEntity, который представляет собой модель данных для хранения задач. В классе определены следующие поля: - id: уникальный идентификатор задачи (автогенерируемый) - title: заголовок задачи - description: описание задачи - dateTime: время напоминания в миллисекундах

Класс помечен аннотацией @Entity для работы с Room:

@Entity(tableName = "tasks")  
data class TaskEntity(  
 @PrimaryKey(autoGenerate = true)  
 val id: Int = 0,  
 val title: String,  
 val description: String,  
 val dateTime: Long  
)

### 2. Создание DAO (Data Access Object)

Далее я реализовал интерфейс TaskDao, который определяет методы для работы с базой данных: - getAllTasks(): получение всех задач, отсортированных по времени - insertTask(): добавление новой задачи - deleteTask(): удаление задачи - updateTask(): обновление существующей задачи

Интерфейс использует корутины и Flow для асинхронной работы:

@Dao  
interface TaskDao {  
 @Query("SELECT \* FROM tasks ORDER BY dateTime ASC")  
 fun getAllTasks(): Flow<List<TaskEntity>>  
  
 @Insert  
 suspend fun insertTask(task: TaskEntity)  
  
 @Delete  
 suspend fun deleteTask(task: TaskEntity)  
  
 @Update  
 suspend fun updateTask(task: TaskEntity)  
}

### 3. Реализация репозитория

Для соблюдения принципов Clean Architecture я создал класс TaskRepository, который инкапсулирует логику работы с DAO:

class TaskRepository(private val taskDao: TaskDao) {  
 val allTasks: Flow<List<TaskEntity>> = taskDao.getAllTasks()  
  
 suspend fun insertTask(task: TaskEntity) {  
 taskDao.insertTask(task)  
 }  
  
 suspend fun deleteTask(task: TaskEntity) {  
 taskDao.deleteTask(task)  
 }  
  
 suspend fun updateTask(task: TaskEntity) {  
 taskDao.updateTask(task)  
 }  
}

Репозиторий предоставляет более высокоуровневый интерфейс для работы с данными и может быть расширен дополнительной логикой в будущем.

### 4. Создание класса базы данных

Финальным этапом работы с БД стало создание класса TaskDatabase, который: - Наследуется от RoomDatabase - Использует паттерн Singleton для создания единственного экземпляра - Предоставляет доступ к DAO

@Database(entities = [TaskEntity::class], version = 1)  
abstract class TaskDatabase : RoomDatabase() {  
 abstract fun taskDao(): TaskDao  
  
 companion object {  
 @Volatile  
 private var INSTANCE: TaskDatabase? = null  
  
 fun getDatabase(context: Context): TaskDatabase {  
 return INSTANCE ?: synchronized(this) {  
 val instance = Room.databaseBuilder(  
 context.applicationContext,  
 TaskDatabase::class.java,  
 "task\_database"  
 ).build()  
 INSTANCE = instance  
 instance  
 }  
 }  
 }  
}

### 5. ViewModel

Я реализовал ViewModel, который является ключевым компонентом архитектурного паттерна MVVM. В моей реализации:

1. Создал класс TaskViewModel, наследующийся от AndroidViewModel
2. Добавил приватное поле repository для работы с данными
3. Реализовал публичное свойство allTasks типа Flow для получения списка задач
4. В блоке инициализации настроил подключение к базе данных через TaskDatabase
5. Реализовал методы для работы с задачами (вставка, удаление, обновление)

Все операции с базой данных выполняются в фоновом потоке с помощью корутин через viewModelScope.

class TaskViewModel(application: Application) : AndroidViewModel(application) {  
 private val repository: TaskRepository  
 val allTasks: Flow<List<TaskEntity>>  
  
 init {  
 val taskDao = TaskDatabase.getDatabase(application).taskDao()  
 repository = TaskRepository(taskDao)  
 allTasks = repository.allTasks  
 }  
  
 fun insertTask(task: TaskEntity) = viewModelScope.launch {  
 repository.insertTask(task)  
 }  
  
 fun deleteTask(task: TaskEntity) = viewModelScope.launch {  
 repository.deleteTask(task)  
 }  
  
 fun updateTask(task: TaskEntity) = viewModelScope.launch {  
 repository.updateTask(task)  
 }  
}

### 6. UI

В этом разделе я реализовал главную активность приложения (MainActivity). Основные компоненты:

1. Инициализация ViewModel с помощью viewModel() делегата
2. Создание экземпляра AlarmScheduler для работы с уведомлениями
3. Использование Jetpack Compose для построения UI:
   * MaterialTheme для стилизации
   * Scaffold с FloatingActionButton для добавления задач
   * Column для отображения списка задач
   * Диалог добавления новой задачи
4. Обработка событий:
   * Добавление новых задач
   * Удаление существующих задач
   * Планирование уведомлений

Код реализации:

class MainActivity : ComponentActivity() {  
 private val alarmScheduler = AlarmScheduler()  
   
 override fun onCreate(savedInstanceState: Bundle?) {  
 super.onCreate(savedInstanceState)  
   
 setContent {  
 val viewModel: TaskViewModel = viewModel()  
 var showDialog by remember { mutableStateOf(false) }  
 val tasks by viewModel.allTasks.collectAsState(initial = emptyList())  
   
 MaterialTheme {  
 Scaffold(  
 floatingActionButton = {  
 FloatingActionButton(onClick = { showDialog = true }) {  
 Text("+")  
 }  
 }  
 ) { padding ->  
 Column(  
 modifier = Modifier  
 .fillMaxSize()  
 .padding(padding)  
 .padding(16.dp)  
 ) {  
 tasks.forEach { task ->  
 TaskItem(  
 task = task,  
 onDelete = {   
 alarmScheduler.cancelAlarm(this@MainActivity, task.id)  
 viewModel.deleteTask(task)  
 }  
 )  
 }  
 }  
   
 if (showDialog) {  
 AddTaskDialog(  
 onDismiss = { showDialog = false },  
 onTaskAdded = { title, description, dateTime ->  
 val newTask = TaskEntity(  
 title = title,  
 description = description,  
 dateTime = dateTime  
 )  
 viewModel.insertTask(newTask)  
 alarmScheduler.setAlarm(  
 context = this@MainActivity,  
 alarmId = newTask.id,  
 timeInMillis = newTask.dateTime,  
 title = newTask.title,  
 description = newTask.description  
 )  
 showDialog = false  
 }  
 )  
 }  
 }  
 }  
 }  
 }  
}

@Composable  
fun TaskItem(  
 task: TaskEntity,  
 onDelete: () -> Unit  
) {  
 Card(  
 modifier = Modifier  
 .fillMaxWidth()  
 .padding(vertical = 8.dp)  
 ) {  
 Column(  
 modifier = Modifier.padding(16.dp)  
 ) {  
 Text(text = task.title, style = MaterialTheme.typography.titleMedium)  
 Text(text = task.description, style = MaterialTheme.typography.bodyMedium)  
 Text(  
 text = "Дата: ${SimpleDateFormat("dd/MM/yyyy HH:mm", Locale.getDefault()).format(Date(task.dateTime))}",  
 style = MaterialTheme.typography.bodySmall  
 )  
 Button(onClick = onDelete) {  
 Text("Удалить")  
 }  
 }  
 }  
}

@Composable  
fun AddTaskDialog(  
 onDismiss: () -> Unit,  
 onTaskAdded: (String, String, Long) -> Unit  
) {  
 var title by remember { mutableStateOf("") }  
 var description by remember { mutableStateOf("") }  
 var dateTimeInMillis by remember { mutableStateOf(0L) }  
   
 AlertDialog(  
 onDismissRequest = onDismiss,  
 title = { Text("Добавить задачу") },  
 text = {  
 Column {  
 TextField(  
 value = title,  
 onValueChange = { title = it },  
 label = { Text("Название") }  
 )  
 Spacer(modifier = Modifier.height(8.dp))  
 TextField(  
 value = description,  
 onValueChange = { description = it },  
 label = { Text("Описание") }  
 )  
 Spacer(modifier = Modifier.height(16.dp))  
 DateTimePicker { millis ->  
 dateTimeInMillis = millis  
 }  
 }  
 },  
 confirmButton = {  
 Button(  
 onClick = {  
 if (dateTimeInMillis > 0) {  
 onTaskAdded(title, description, dateTimeInMillis)  
 }  
 },  
 enabled = title.isNotBlank() && dateTimeInMillis > 0  
 ) {  
 Text("Добавить")  
 }  
 },  
 dismissButton = {  
 Button(onClick = onDismiss) {  
 Text("Отмена")  
 }  
 }  
 )  
}

@OptIn(ExperimentalMaterial3Api::class)  
@Composable  
fun DateTimePicker(  
 onDateTimeSelected: (Long) -> Unit  
) {  
 var showDatePicker by remember { mutableStateOf(false) }  
 var showTimePicker by remember { mutableStateOf(false) }  
   
 val calendar = Calendar.getInstance()  
   
 var selectedYear by remember { mutableStateOf(calendar.get(Calendar.YEAR)) }  
 var selectedMonth by remember { mutableStateOf(calendar.get(Calendar.MONTH)) }  
 var selectedDay by remember { mutableStateOf(calendar.get(Calendar.DAY\_OF\_MONTH)) }  
 var selectedHour by remember { mutableStateOf(calendar.get(Calendar.HOUR\_OF\_DAY)) }  
 var selectedMinute by remember { mutableStateOf(calendar.get(Calendar.MINUTE)) }  
   
 // Функция для проверки валидности выбранной даты/времени  
 fun isValidDateTime(): Boolean {  
 val selectedCalendar = Calendar.getInstance().apply {  
 set(selectedYear, selectedMonth, selectedDay, selectedHour, selectedMinute, 0)  
 set(Calendar.MILLISECOND, 0)  
 }  
 return selectedCalendar.timeInMillis > System.currentTimeMillis()  
 }  
  
 Column {  
 Button(onClick = { showDatePicker = true }) {  
 Text("Выбрать дату: $selectedDay/${selectedMonth + 1}/$selectedYear")  
 }  
   
 Button(onClick = { showTimePicker = true }) {  
 Text("Выбрать время: ${String.format("%02d:%02d", selectedHour, selectedMinute)}")  
 }  
  
 val selectedCalendar = Calendar.getInstance().apply {  
 set(selectedYear, selectedMonth, selectedDay, selectedHour, selectedMinute, 0)  
 set(Calendar.MILLISECOND, 0)  
 }  
   
 if (isValidDateTime()) {  
 Text(  
 "Напоминание будет показано: " +  
 SimpleDateFormat("dd/MM/yyyy HH:mm", Locale.getDefault())  
 .format(selectedCalendar.time),  
 color = MaterialTheme.colorScheme.primary  
 )  
 // Вызываем callback только если дата валидна  
 onDateTimeSelected(selectedCalendar.timeInMillis)  
 } else {  
 Text("Выберите будущую дату", color = MaterialTheme.colorScheme.error)  
 }  
 }  
  
 if (showDatePicker) {  
 val datePickerState = rememberDatePickerState(  
 initialSelectedDateMillis = Calendar.getInstance().timeInMillis  
 )  
   
 DatePickerDialog(  
 onDismissRequest = { showDatePicker = false },  
 confirmButton = {  
 TextButton(  
 onClick = {  
 datePickerState.selectedDateMillis?.let { selectedDateMillis ->  
 val newDate = Calendar.getInstance().apply {  
 timeInMillis = selectedDateMillis  
 }  
 selectedYear = newDate.get(Calendar.YEAR)  
 selectedMonth = newDate.get(Calendar.MONTH)  
 selectedDay = newDate.get(Calendar.DAY\_OF\_MONTH)  
 }  
 showDatePicker = false  
 }  
 ) {  
 Text("OK")  
 }  
 },  
 dismissButton = {  
 TextButton(onClick = { showDatePicker = false }) {  
 Text("Отмена")  
 }  
 }  
 ) {  
 DatePicker(state = datePickerState)  
 }  
 }  
  
 if (showTimePicker) {  
 val timePickerState = rememberTimePickerState(  
 initialHour = selectedHour,  
 initialMinute = selectedMinute  
 )  
   
 AlertDialog(  
 onDismissRequest = { showTimePicker = false },  
 confirmButton = {  
 TextButton(  
 onClick = {  
 selectedHour = timePickerState.hour  
 selectedMinute = timePickerState.minute  
 showTimePicker = false  
 }  
 ) {  
 Text("OK")  
 }  
 },  
 dismissButton = {  
 TextButton(onClick = { showTimePicker = false }) {  
 Text("Отмена")  
 }  
 },  
 text = {  
 TimePicker(state = timePickerState)  
 }  
 )  
 }  
}

В данном коде все Composable-функции принимают параметры в виде callback-функций для обработки событий:

* MainActivity передает callback-функции в AddTaskDialog для обработки добавления задачи (onTaskAdded) и закрытия диалога (onDismiss)
* TaskItem принимает callback onDelete для удаления задачи
* DateTimePicker получает callback onDateTimeSelected для передачи выбранной даты и времени
* Диалоги DatePickerDialog и AlertDialog также используют callback-функции для обработки подтверждения и отмены

Такой подход с передачей callback-функций через параметры обеспечивает слабую связанность компонентов и позволяет переиспользовать их в разных местах приложения.

# Задание 4: таймер с обратным отсчётом

Приложение позволяет пользователю задать время в минутах, после чего начинается обратный отсчет. Состояние таймера сохраняется между запусками приложения с помощью SharedPreferences.

Основные компоненты приложения:

1. TimerViewModel - отвечает за логику работы таймера:

* Управление CountDownTimer для отсчета времени с интервалом в 1 секунду
* Хранение состояния таймера в mutableStateOf (timeLeft, isRunning, isPaused)
* Запуск таймера с заданным временем и callback при завершении
* Отмена таймера и остановка звукового сигнала
* Работа с уведомлениями через NotifService
* Управление звуковыми сигналами через MediaPlayerService
* Сохранение состояния в TimerPreferences

import android.os.CountDownTimer  
import androidx.compose.runtime.mutableStateOf  
import androidx.lifecycle.ViewModel  
import androidx.lifecycle.viewModelScope  
import com.example.timerio.data.TimerPreferences  
import kotlinx.coroutines.launch  
import com.example.timerio.service.NotifService  
import com.example.timerio.service.MediaPlayerService  
  
class TimerViewModel(  
 private val timerPreferences: TimerPreferences,  
 private val notifService: NotifService,  
 private val mediaPlayerService: MediaPlayerService  
) : ViewModel() {  
   
 private var timer: CountDownTimer? = null  
 var timeLeft = mutableStateOf(0L)  
 var isRunning = mutableStateOf(false)  
 var isPaused = mutableStateOf(false)  
  
 init {  
 mediaPlayerService.initMediaPlayer()  
 }  
  
 // Запускаем таймер  
 fun startTimer(timeInMillis: Long, onFinish: () -> Unit) {  
 timer?.cancel()  
 mediaPlayerService.stopAlarmSound()  
   
 timeLeft.value = timeInMillis  
 isRunning.value = true  
   
 timer = object : CountDownTimer(timeInMillis, 1000) {  
 override fun onTick(millisUntilFinished: Long) {  
 timeLeft.value = millisUntilFinished  
 }  
  
 override fun onFinish() {  
 isRunning.value = false  
 timeLeft.value = 0  
 mediaPlayerService.playAlarmSound()  
 notifService.showNotification("Таймер завершен!")  
 onFinish()  
 }  
 }.start()  
 }  
  
 // Останавливаем таймер  
 fun cancelTimer() {  
 timer?.cancel()  
 mediaPlayerService.stopAlarmSound()  
 isRunning.value = false  
 timeLeft.value = 0  
 }  
  
 fun showNotification(message: String) {  
 notifService.showNotification(message)  
 }  
  
 fun cancelNotification() {  
 notifService.cancelNotification()  
 }  
  
 // метод для сохранения состояния  
 fun saveState() {  
 timerPreferences.saveState(timeLeft.value, isRunning.value)  
 }  
}

1. Функции для работы с SharedPreferences:

* saveTimerState() - сохраняет оставшееся время в SharedPreferences через putLong()
* loadTimerState() - загружает сохраненное время из SharedPreferences через getLong() с дефолтным значением 0L
* saveState() - сохраняет оставшееся время и состояние таймера (isRunning)
* getContext() - возвращает контекст приложения

import android.content.Context  
import android.content.SharedPreferences  
  
class TimerPreferences(appContext: Context) {  
 private val context = appContext  
 private val sharedPreferences: SharedPreferences =   
 context.getSharedPreferences(TIMER\_PREFERENCES, Context.MODE\_PRIVATE)  
  
 fun saveTimerState(timeLeft: Long) {  
 sharedPreferences.edit().apply {  
 putLong(KEY\_TIME\_LEFT, timeLeft)  
 apply()  
 }  
 }  
  
 fun loadTimerState(): Long {  
 return sharedPreferences.getLong(KEY\_TIME\_LEFT, 0L)  
 }  
  
 fun saveState(timeLeft: Long, isRunning: Boolean) {  
 sharedPreferences.edit().apply {  
 putLong(KEY\_TIME\_LEFT, timeLeft)  
 putBoolean(KEY\_IS\_RUNNING, isRunning)  
 apply()  
 }  
 }  
  
 fun getContext(): Context {  
 return context  
 }  
  
 companion object {  
 private const val TIMER\_PREFERENCES = "timer\_prefs"  
 private const val KEY\_TIME\_LEFT = "time\_left"  
 private const val KEY\_IS\_RUNNING = "is\_running"  
 }  
}

1. Пользовательский интерфейс (TimerScreen):

* Поле ввода времени с возможностью выбора между минутами и секундами через FilterChip
* TextField для ввода числового значения с ограничением в 3 цифры
* Кнопка запуска таймера, активная только при непустом вводе
* При запущенном таймере:
  + Отображение оставшегося времени в формате MM:SS
  + Кнопка отмены таймера красного цвета
* Центрированное расположение всех элементов
* Уведомление при завершении отсчета

import androidx.compose.foundation.layout.\*  
import androidx.compose.foundation.text.KeyboardOptions  
import androidx.compose.material3.\*  
import androidx.compose.runtime.\*  
import androidx.compose.ui.Alignment  
import androidx.compose.ui.Modifier  
import androidx.compose.ui.text.input.KeyboardType  
import androidx.compose.ui.unit.dp  
import androidx.compose.ui.text.style.TextAlign  
import java.util.concurrent.TimeUnit  
  
@OptIn(ExperimentalMaterial3Api::class)  
@Composable  
fun TimerScreen(viewModel: TimerViewModel) {  
 var inputTime by remember { mutableStateOf("") }  
 var isMinutesSelected by remember { mutableStateOf(true) }  
 val timeLeft by viewModel.timeLeft  
 val isRunning by viewModel.isRunning  
  
 Box(modifier = Modifier.fillMaxSize()) {  
 Column(  
 modifier = Modifier  
 .fillMaxSize()  
 .padding(16.dp),  
 horizontalAlignment = Alignment.CenterHorizontally,  
 verticalArrangement = Arrangement.Center  
 ) {  
 if (!isRunning) {  
 // Переключатель единиц времени  
 Row(  
 modifier = Modifier.padding(horizontal = 16.dp),  
 horizontalArrangement = Arrangement.Center  
 ) {  
 FilterChip(  
 selected = isMinutesSelected,  
 onClick = { isMinutesSelected = true },  
 label = { Text("Минуты") },  
 modifier = Modifier.padding(end = 8.dp)  
 )  
 FilterChip(  
 selected = !isMinutesSelected,  
 onClick = { isMinutesSelected = false },  
 label = { Text("Секунды") }  
 )  
 }  
  
 Spacer(modifier = Modifier.height(16.dp))  
  
 // Поле для ввода времени  
 TextField(  
 value = inputTime,  
 onValueChange = { if (it.length <= 3) inputTime = it.filter { char -> char.isDigit() } },  
 label = { Text(if (isMinutesSelected) "Введите время в минутах" else "Введите время в секундах") },  
 keyboardOptions = KeyboardOptions(keyboardType = KeyboardType.Number),  
 modifier = Modifier.fillMaxWidth()  
 )  
   
 Spacer(modifier = Modifier.height(16.dp))  
   
 // Кнопка запуска таймера  
 Button(  
 onClick = {  
 val time = inputTime.toLongOrNull()  
 if (time != null && time > 0) {  
 val timeInMillis = if (isMinutesSelected) {  
 time \* 60 \* 1000 // конвертация минут в миллисекунды  
 } else {  
 time \* 1000 // конвертация секунд в миллисекунды  
 }  
 viewModel.startTimer(  
 timeInMillis = timeInMillis,  
 onFinish = {  
 viewModel.showNotification("Таймер завершен!")  
 }  
 )  
 inputTime = ""  
 }  
 },  
 enabled = inputTime.isNotEmpty()  
 ) {  
 Text("Запустить таймер")  
 }  
 } else {  
 // Отображение оставшегося времени  
 Text(  
 text = formatTime(timeLeft),  
 style = MaterialTheme.typography.headlineLarge,  
 textAlign = TextAlign.Center,  
 modifier = Modifier.padding(16.dp)  
 )  
   
 Spacer(modifier = Modifier.height(16.dp))  
   
 // Кнопка отмены таймера  
 Button(  
 onClick = { viewModel.cancelNotification() },  
 colors = ButtonDefaults.buttonColors(  
 containerColor = MaterialTheme.colorScheme.error  
 )  
 ) {  
 Text("Отменить")  
 }  
 }  
 }  
 }  
}  
  
private fun formatTime(timeInMillis: Long): String {  
 val minutes = TimeUnit.MILLISECONDS.toMinutes(timeInMillis)  
 val seconds = TimeUnit.MILLISECONDS.toSeconds(timeInMillis) % 60  
 return String.format("%02d:%02d", minutes, seconds)  
}

1. MainActivity:

* Инициализация ViewModel через Koin (private val viewModel: TimerViewModel by viewModel())
* Отображение TimerScreen в MaterialTheme с помощью Jetpack Compose
* Сохранение состояния таймера в onStop() через lifecycleScope.launch, если таймер активен

import android.os.Bundle  
import androidx.activity.ComponentActivity  
import androidx.activity.compose.setContent  
import androidx.compose.foundation.layout.fillMaxSize  
import androidx.compose.material3.MaterialTheme  
import androidx.compose.material3.Surface  
import androidx.compose.ui.Modifier  
import androidx.lifecycle.lifecycleScope  
import com.example.timerio.ui.TimerScreen  
import com.example.timerio.ui.TimerViewModel  
import com.example.timerio.ui.theme.TimerioTheme  
import kotlinx.coroutines.launch  
import org.koin.androidx.viewmodel.ext.android.viewModel  
  
class MainActivity : ComponentActivity() {  
 private val viewModel: TimerViewModel by viewModel()  
  
 override fun onCreate(savedInstanceState: Bundle?) {  
 super.onCreate(savedInstanceState)  
  
 setContent {  
 TimerioTheme {  
 Surface(  
 modifier = Modifier.fillMaxSize(),  
 color = MaterialTheme.colorScheme.background  
 ) {  
 TimerScreen(viewModel)  
 }  
 }  
 }  
 }  
  
 override fun onStop() {  
 super.onStop()  
 // Сохраняем состояние при сворачивании приложения  
 lifecycleScope.launch {  
 if (viewModel.isRunning.value) {  
 // Сохраняем только если таймер активен  
 viewModel.saveState()  
 }  
 }  
 }  
}

1. Сервис уведомлений (NotifService):

* Создание канала уведомлений с высоким приоритетом
* Отображение уведомлений с кнопкой остановки таймера
* Отмена уведомлений при необходимости

import android.Manifest  
import android.app.NotificationChannel  
import android.app.NotificationManager  
import android.app.PendingIntent  
import android.content.Context  
import android.content.Intent  
import android.content.pm.PackageManager  
import android.os.Build  
import androidx.core.app.ActivityCompat  
import androidx.core.app.NotificationCompat  
import androidx.core.app.NotificationManagerCompat  
import com.example.timerio.R  
import com.example.timerio.receiver.NotificationActionReceiver  
  
class NotifService(private val context: Context) {  
  
 private val channelId = "timer\_channel"  
 private val notificationId = 1001  
 private val notificationManager: NotificationManager =   
 context.getSystemService(Context.NOTIFICATION\_SERVICE) as NotificationManager  
  
 companion object {  
 const val EXTRA\_NOTIFICATION\_ID = "notification\_id"  
 const val EXTRA\_TIME\_LEFT = "time\_left"  
 }  
  
 init {  
 createNotificationChannel()  
 }  
  
 private fun createNotificationChannel() {  
 if (Build.VERSION.SDK\_INT >= Build.VERSION\_CODES.O) {  
 val channel = NotificationChannel(  
 channelId,  
 "Timer Channel",  
 NotificationManager.IMPORTANCE\_HIGH  
 )  
 notificationManager.createNotificationChannel(channel)  
 }  
 }  
  
 fun showNotification(message: String, timeLeft: Long = 0) {  
 val stopIntent = Intent(context, NotificationActionReceiver::class.java).apply {  
 action = "ACTION\_STOP\_ALARM"  
 }  
 val stopPendingIntent = PendingIntent.getBroadcast(  
 context, 0, stopIntent,   
 PendingIntent.FLAG\_UPDATE\_CURRENT or PendingIntent.FLAG\_IMMUTABLE  
 )  
  
 val notification = NotificationCompat.Builder(context, channelId)  
 .setSmallIcon(R.drawable.ic\_timer)  
 .setContentTitle("Timer")  
 .setContentText(message)  
 .addAction(R.drawable.ic\_stop, "Stop", stopPendingIntent)  
 .setPriority(NotificationCompat.PRIORITY\_HIGH)  
 .build()  
  
 notificationManager.notify(notificationId, notification)  
 }  
  
 fun cancelNotification() {  
 notificationManager.cancel(notificationId)  
 }  
}

1. MediaPlayerService - сервис для работы со звуком:

* Инициализация MediaPlayer и загрузка звукового файла из ресурсов
* Методы для управления воспроизведением:
  + initMediaPlayer() - создание и подготовка плеера
  + playAlarmSound() - запуск воспроизведения
  + stopAlarmSound() - остановка воспроизведения
  + release() - освобождение ресурсов

import android.content.Context  
import android.media.MediaPlayer  
import android.util.Log  
import com.example.timerio.R  
  
class MediaPlayerService(private val context: Context) {  
 private var mediaPlayer: MediaPlayer? = null  
  
 // Создаем плеер  
 fun initMediaPlayer() {  
 mediaPlayer = MediaPlayer()  
 mediaPlayer?.setDataSource(context.resources.openRawResourceFd(R.raw.clock))  
 mediaPlayer?.prepare()  
 }  
  
 // Включаем звук  
 fun playAlarmSound() {  
 mediaPlayer?.start()  
 }  
  
 // Выключаем звук  
 fun stopAlarmSound() {  
 mediaPlayer?.let { player ->  
 if (player.isPlaying) {  
 player.stop()  
 player.prepare()  
 }  
 }  
 }  
  
 // Освобождаем ресурсы  
 fun release() {  
 mediaPlayer?.release()  
 mediaPlayer = null  
 }  
}

1. NotificationActionReceiver - обработчик действий с уведомлениями:

* Наследуется от BroadcastReceiver для получения широковещательных сообщений
* Использует Koin для внедрения зависимостей (TimerViewModel, MediaPlayerService, NotifService)
* В методе onReceive обрабатывает действие ACTION\_STOP\_ALARM:
  + Останавливает проигрывание звука через MediaPlayerService
  + Отменяет таймер через TimerViewModel
  + Убирает уведомление через NotifService

Мы используем вндрение зависимостей через Koin для удобства вызова методов из ресивера.

Приложение демонстрирует работу с: - Jetpack Compose для создания UI - ViewModel для управления состоянием - SharedPreferences для сохранения данных - CountDownTimer для отсчета времени - Уведомлениями и звуковыми сигналами

# Заключение

В ходе учебной практики были разработаны два приложения: «Электронный конспект» и «Список задач» и «Напоминалка» и «Таймер». Оба приложения были созданы с использованием современных инструментов и библиотек, таких как Jetpack Compose и Room. Студенты получили практический опыт разработки Android-приложений, изучили основные компоненты и подходы к созданию пользовательского интерфейса и работе с локальной базой данных.

Дополнительные задания позволили студентам углубить свои знания и навыки, реализовав более сложные функции и улучшив дизайн приложений. Такой практический подход к обучению способствует лучшему пониманию и закреплению изученного материала.

В целом, учебная практика была успешно завершена, и студенты продемонстрировали высокий уровень владения современными технологиями разработки Android-приложений.