1. Лабораторна робота №1. Життєвий цикл Activity

Мета: набути теоретичних знань та практичних навичок з побудови архітектури Android-додатку з урахуванням життєвого циклу Activity.

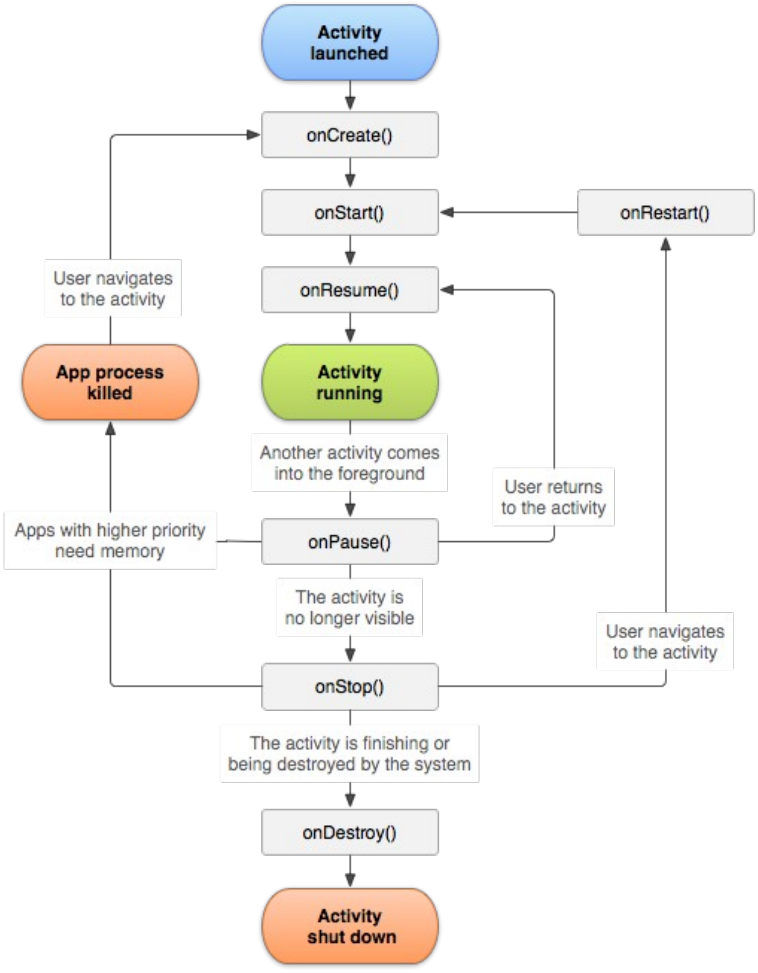
Завдання: розробити додаток відповідно до отриманої специфікації, передбачити реагування додатку на зміну стану Activity.

Життєвий цикл Activity в Android описує набір методів (колбеків), які викликаються системою, щоб повідомити ваш додаток про зміни в стані Activity. Ці методи допомагають розуміти, коли Activity створюється, запускається, призупиняється, відновлюється або знищується. У контексті Android Compose (де інтерфейс користувача будується за допомогою декларативних компонентів), ці колбеки життєвого циклу Activity все одно мають важливе значення, оскільки вони дозволяють вам керувати ресурсами, станом UI і здійснювати відповідну реакцію на зміну стану Activity.

Основні колбеки життєвого циклу Activity:

1. onCreate()
2. onStart()
3. onResume()
4. onPause()
5. onStop()
6. onRestart()
7. onDestroy()

https://developer.android.com/guide/components/activities/activity-lifecycle



Розгорнуте пояснення колбеків:

1. onCreate()

Метод onCreate() викликається один раз, коли Activity створюється, тобто коли воно ініціалізується. Це місце для виконання одноразових операцій налаштування Activity, таких як ініціалізація UI, налаштування зв'язків з базою даних, ініціалізація змінних або інших компонентів додатка.

Приклад використання:

```
class MainActivity : ComponentActivity() {

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContent {
            // Тут встановлюємо інтерфейс користувача за допомогою Jetpack Compose
            Greeting("Hello, World!")
        }
    }
}
```

У цьому прикладі ми використовуємо setContent{} для визначення UI через Compose в методі onCreate().

2. onStart()

Метод onStart() викликається, коли Activity стає видимим для користувача, але ще не взаємодіє з ним. Це означає, що Activity готове до взаємодії, але воно ще не активне.

Приклад:

```
override fun onStart() {
    super.onStart()
    // Тут можна ініціалізувати або оновити UI, якщо це необхідно
}
```

3. onResume()

Метод onResume() викликається, коли Activity починає активно взаємодіяти з користувачем. Це означає, що Activity відновлено на передній план і користувач може взаємодіяти з ним.

Приклад:

```
override fun onResume() {
    super.onResume()
    // Тут можна відновити чи оновити ресурси, наприклад, відновити відтворення відео
або анімації
}
```

4. onPause()

Метод onPause() викликається, коли Activity більше не взаємодіє з користувачем, але все ще видимий. Цей метод може бути викликаний, коли Activity переходить в фоновий режим, наприклад, при відкритті іншої Activity або під час обробки сповіщень.

Приклад:

```
override fun onPause() {
    super.onPause()
    // Тут можна зберігати дані чи припиняти неактивні процеси, наприклад, зупиняти
анімації або відео
}
```

5. onStop()

Метод onStop() викликається, коли Activity більше не видно. Це може статися, коли Activity переходить в фоновий режим або замінюється іншою Activity. Якщо ви хочете зберігати ресурси, чи виконувати інші операції, це місце для їх виконання.

```
Приклад:
override fun onStop() {
    super.onStop()
    // Тут можна звільняти ресурси, наприклад, скасувати запити до сервера або
звільнити пам'ять
}
```

6. onRestart()

Метод onRestart() викликається перед тим, як Activity знову стане видимим для користувача, після того як вона була зупинена. Це хороший момент, щоб оновити або відновити ресурси, які могли бути втрачені під час зупинки Activity.

Приклад:

```
override fun onRestart() {
    super.onRestart()
    // Тут можна відновити процеси або оновити дані після зупинки Activity
}
```

7. onDestroy()

Метод onDestroy() викликається перед тим, як Activity буде зруйновано. Це місце для очищення ресурсів, таких як скасування асинхронних запитів, збереження остаточних даних або очищення пам'яті.

Приклад:

```
override fun onDestroy() {
    super.onDestroy()
    // Тут можна звільняти ресурси або очищати дані
}
```

Взаємодія з Jetpack Compose

У Jetpack Compose використовується інший підхід для управління станом UI, однак колбеки життєвого циклу Activity залишаються важливими для контролю над життєвим циклом додатка та управління ресурсами.

Приклад інтеграції життєвого циклу Activity з Compose:

```
class MainActivity : ComponentActivity() {
    private val lifecycleOwner = this

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

        // Використання життєвого циклу для моніторингу
        lifecycleOwner.lifecycle.addObserver(object : LifecycleObserver {
            override fun onResume() {
                super.onResume()
                // Дії при відновленні Activity
            }

            override fun onPause() {
                super.onPause()
                // Дії при паузі Activity
            }
        })
    }
}
```

У цьому прикладі показано, як можна додати спостерігача життєвого циклу для виконання додаткових дій, що не є частиною стандартного життєвого циклу Activity.

Як це стосується Android Compose?

1. setContent: У Compose метод setContent {} заміняє стандартний XML-UI і дозволяє визначити UI прямо в коді. Однак Activity все одно повинна керувати своїм життєвим циклом, оскільки вона є основним контейнером для Compose UI.
2. Поширення стану: У Compose часто використовують ViewModel для зберігання та управління станом UI. Колбеки життєвого циклу Activity, такі як onStart() або onStop(), можна використовувати для скидання або збереження стану, якщо це потрібно.
3. Асинхронні операції: Оскільки Android Compose часто взаємодіє з асинхронними процесами (наприклад, запити до сервера або бази даних), колбеки життєвого циклу Activity можуть використовуватися для зупинки або відновлення цих процесів залежно від того, чи є Activity активним чи на фоні.

Життєвий цикл Activity є важливою частиною Android-додатків, оскільки він контролює, коли відбуваються важливі операції з ресурсами (створення, зупинка, знищення), і це актуально навіть у контексті Jetpack Compose. Взаємодія з цими колбеками допомагає

ефективно управляти ресурсами, оптимізувати продуктивність додатка та забезпечити належну поведінку додатка під час змін у життєвому циклі.

Приклад із офіційної документації.
https://developer.android.com/guide/components/activities/activity-lifecycle

```
lateinit var textView: TextView

// Some transient state for the activity instance.
var gameState: String? = null

override fun onCreate(savedInstanceState: Bundle?) {
    // Call the superclass onCreate to complete the creation of
    // the activity, like the view hierarchy.
    super.onCreate(savedInstanceState)

    // Recover the instance state.
    gameState = savedInstanceState?.getString(GAME_STATE_KEY)

    // Set the user interface layout for this activity.
    // The layout is defined in the project res/layout/main_activity.xml file.
    setContentView(R.layout.main_activity)

    // Initialize member TextView so it is available later.
    textView = findViewById(R.id.text_view)
}

// This callback is called only when there is a saved instance previously saved using
// onSaveInstanceState(). Some state is restored in onCreate(). Other state can optionally
// be restored here, possibly usable after onStart() has completed.
// The savedInstanceState Bundle is same as the one used in onCreate().
override fun onRestoreInstanceState(savedInstanceState: Bundle?) {
    textView.text = savedInstanceState?.getString(TEXT_VIEW_KEY)
}

// Invoked when the activity might be temporarily destroyed; save the instance state here.
override fun onSaveInstanceState(outState: Bundle?) {
    outState?.run {
        putString(GAME_STATE_KEY, gameState)
        putString(TEXT_VIEW_KEY, textView.text.toString())
    }
    // Call superclass to save any view hierarchy.
    super.onSaveInstanceState(outState)
}
```

Хід роботи

1. Ознайомитися із теоретичними відомостями (виконується до початку лабораторного заняття).
2. Дослідити функціонал, що розглядається у теоретичних відомостях.

3. Виконати практичне завдання, поставлене викладачем (зазвичай суть практичного завдання полягає у розробці невеликого сценарію, функціональність якого визначається темою, що розглядається, ти стислими термінами розробки під час лабораторного заняття).
4. Зробити висновки.
5. Захистити роботу та відповісти на контрольні запитання.

Контрольні запитання

1. Чим життєвий цикл Activity у Jetpack Compose відрізняється від класичного Android UI?
2. Який метод Activity використовується для встановлення Composable-інтерфейсу?
3. Що таке setContent {} у Jetpack Compose і коли він викликається?
4. Як remember і mutableStateOf взаємодіють із життєвим циклом Activity?
5. Чи зберігається стан Composable після знищення Activity? Як цього досягти?
6. Як реагують Composable-функції на події життєвого циклу, такі як onPause() чи onStop()?
7. Яким чином можна відслідковувати життєвий цикл Activity всередині Composable?
8. Яке призначення LaunchedEffect та як він пов’язаний з життєвим циклом?
9. Як використовуються DisposableEffect і SideEffect у зв’язку з життєвим циклом Activity або екрана?
10. Яким чином Jetpack Compose інтегрується з ViewModel для збереження даних при зміні конфігурації Activity?

Довідкова інформація

1. https://developer.android.com/develop/ui/compose/lifecycle
2. https://developer.android.com/guide/components/activities/activity-lifecycle

Лабораторна робота №2. Поняття композиції та рекомпозиції

Мета: набути теоретичних знань та практичних навичок з побудови архітектури Android-додатку з урахуванням понять композиції та рекомпозиції у контексті використанні бібліотеки Compose.

Завдання: розробити додаток відповідно до отриманої специфікації, продемонструвати оновлення інтерфейса користувача з використанням State, State Hoisting.

У контексті Android-розробки, поняття композиції та рекомпозиції найчастіше пов’язані з Jetpack Compose — сучасним підходом до побудови інтерфейсу користувача у Android. Ось пояснення цих понять:

Композиція (Composition)

Композиція — це процес створення UI, коли функції-компоненти (Composable-функції) викликаються для відображення елементів інтерфейсу.

- У Compose UI все будується з функцій із позначкою @Composable.
- Ці функції визначають, *що* має бути показано на екрані.
- Під час композиції фреймворк "записує", які саме компоненти потрібно відобразити та як вони взаємозв’язані.

Приклад:

```
@Composable
fun Greeting(name: String) {
    Text(text = "Hello, $name!")
}
```

Коли ця функція викликається в UI, відбувається композиція — побудова елементів на екрані на основі даних.

Рекомпозиція (Recomposition)

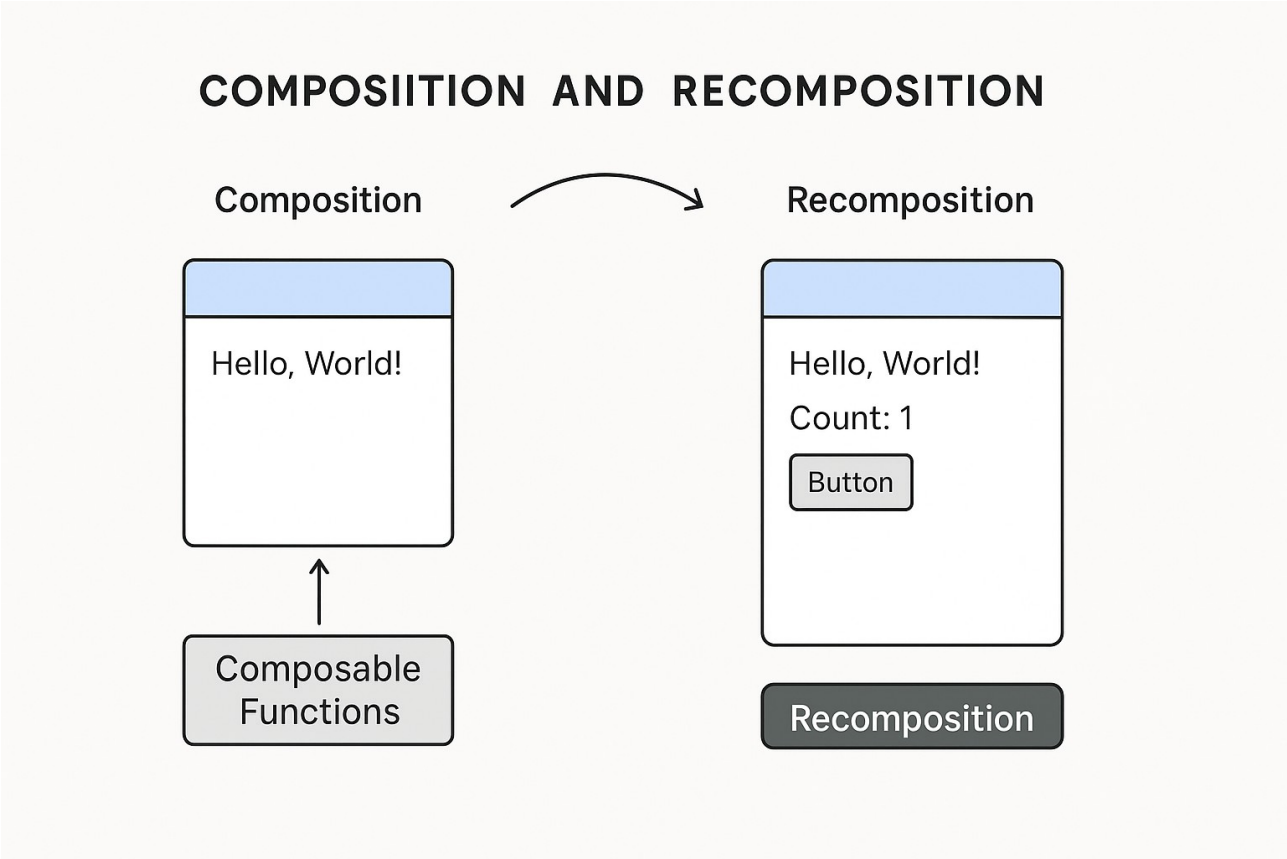
Рекомпозиція — це процес повторного виконання Composable-функцій у відповідь на зміну стану (State), що використовується в UI.

- Compose є реактивним: коли змінюється значення, що використовується у відображенні, тільки ті частини UI, які залежать від нього, перерисовуються.
- Це дозволяє оновлювати інтерфейс ефективно, без повного перезавантаження всього екрану.

Наприклад:

```
@Composable
fun Counter() {
    var count by remember { mutableStateOf(0) }
    Button(onClick = { count++ }) {
        Text("Clicked $count times")
    }
}
```

Коли користувач натискає кнопку, значення count змінюється, і відбувається рекомпозиція — лише Text оновлюється з новим числом, без повного відтворення всього інтерфейсу.



У контексті Composable-функцій в Android (Jetpack Compose), **State** — це ключовий механізм для реактивного оновлення UI. Його головне призначення — зберігати та відслідковувати значення, які можуть змінюватися з часом, і автоматично викликати рекомпозицію тих частин інтерфейсу, які залежать від цього значення.

Основні цілі State:

1. Зберігати динамічні дані, які можуть змінюватися під час життя Composable.
2. Тригерити рекомпозицію UI, коли значення змінюється.
3. Реактивно оновлювати тільки потрібні частини інтерфейсу, що робить додаток ефективним.

Jetpack Compose використовує обгортку MutableState<T>, яка тримає значення та сповіщає фреймворк про зміни.

Приклад:

```
@Composable
fun Counter() {
    var count by remember { mutableStateOf(0) }
    Button(onClick = { count++ }) {
        Text("Clicked $count times")
    }
}
```

- mutableStateOf(0) створює об’єкт State, який зберігає число.
- remember зберігає цей стан під час рекомпозицій, щоб він не скидалися.
- by — делегування, щоб працювати з count як із простим значенням.
- При зміні count (count++), відбувається рекомпозиція, і Text оновлюється.

Без State інтерфейс був би статичним: зміни в даних не впливали б на зовнішній вигляд.

State Hoisting — це один з ключових принципів архітектури Jetpack Compose, який означає винесення (підняття) стану з дочірнього компонента в батьківський. Це дозволяє створювати чисті та повторно використовувані компоненти, які не керують власним станом, а отримують його та функції зміни "ззовні" — через параметри.

Суть State Hoisting:

У Jetpack Compose стан — це значення, яке зберігається і викликає перекомпоновку (recomposition), коли змінюється. Наприклад, це може бути remember { mutableStateOf(...) }.

State hoisting пропонує передавати цей стан та функції для його зміни з батьківського компонента до дочірнього компонента, замість того, щоб кожен компонент сам керував своїм станом.

Чому це важливо?

- Покращує повторне використання компонентів.
- Дозволяє централізовано керувати станом у ViewModel або в Activity.
- Полегшує тестування компонентів, оскільки вони не мають внутрішньої логіки стану.
- Підтримує односторонній потік даних (Unidirectional Data Flow, UDF), що робить UI передбачуваним.

Приклад без State Hoisting (неправильно, компонент сам керує станом):

```
@Composable
fun Counter() {
    var count by remember { mutableStateOf(0) }

    Button(onClick = { count++ }) {
        Text("Count: $count")
    }
}
```

Цей компонент не може бути гнучко використаний — він сам керує своїм станом.

Приклад з State Hoisting (правильно, стан винесено):

```
@Composable
fun Counter(
    count: Int,
    onIncrement: () -> Unit
) {
    Button(onClick = onIncrement) {
        Text("Count: $count")
    }
}
```

А керування станом тепер в батьківському компоненті:

```
@Composable
fun CounterScreen() {
    var count by remember { mutableStateOf(0) }

    Counter(count = count, onIncrement = { count++ })
}
```

Схема: хто за що відповідає

Рівень	Відповідальність
CounterScreen	Зберігає стан (count) і змінює його
Counter	Тільки показує значення і передає дії нагору

Інший приклад — TextField
Без hoisting:

```
@Composable
fun SearchField() {
    var query by remember { mutableStateOf("") }

    TextField(value = query, onValueChange = { query = it })
}
```

Це обмежує повторне використання, бо компонент сам контролює текст.
З hoisting:

```
@Composable
fun SearchField(
    query: String,
    onQueryChange: (String) -> Unit
) {
    TextField(value = query, onValueChange = onQueryChange)
}

@Composable
fun SearchScreen() {
    var searchQuery by remember { mutableStateOf("") }

    SearchField(query = searchQuery, onQueryChange = { searchQuery = it })
}
```

Коли застосовувати state hoisting?

- Якщо компонент має змінний стан і його потрібно контролювати ззовні.
- Якщо ви хочете, щоб компонент був максимально гнучким і повторно використовуваним.
- Якщо компонент — частина складнішого UI, де станом керує ViewModel або батьківський контейнер.

State Hoisting — це практика "витягування" стану з компонентів, щоб керувати ним зовні. В Jetpack Compose це дає кращий контроль над UI, покращує архітектуру, спрощує тестування та сприяє чистому розподілу відповідальності між компонентами.

Нижче наведено приклад State Hoisting у зв'язці з ViewModel у Jetpack Compose — це типовий і рекомендований підхід для управління станом у додатках Android.

Створимо простий екран з TextField, який дозволяє користувачеві вводити текст. Текст зберігається у ViewModel. Компонент InputField лише показує інтерфейс і передає зміни нагору через параметри — тобто, застосовується state hoisting.

1. ViewModel

```
class MyViewModel : ViewModel() {

    private val _text = mutableStateOf("")
    val text: State<String> = _text

    fun onTextChanged(newText: String) {
        _text.value = newText
    }
}
```

2. Composable, який отримує стан і функцію — hoisted component

```
@Composable
fun InputField(
    value: String,
    onValueChange: (String) -> Unit
) {
    TextField(
        value = value,
        onValueChange = onValueChange,
        label = { Text("Введіть текст") },
        modifier = Modifier.fillMaxWidth().padding(16.dp)
    )
}
```

3. Екран (Composable), що підключає ViewModel і передає стан

```
@Composable
fun MyScreen(viewModel: MyViewModel = viewModel()) {
```

```
    val text by viewModel.text

    Column(modifier = Modifier.fillMaxSize()) {
        InputField(
            value = text,
            onValueChange = viewModel::onTextChanged
        )

        Text(
            text = "Введено: $text",
            modifier = Modifier.padding(16.dp)
        )
    }
}
```

4. MainActivity

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

MyViewModel	Зберігає та оновлює стан (text)
MyScreen	Отримує стан з ViewModel і "піднімає" його до InputField
InputField	Відображає UI, але не має власного стану

Це чистий приклад state hoisting у поєднанні з ViewModel, який дозволяє:

- Винести логіку поза UI-компоненти.
- Повторно використовувати InputField.
- Зберігати односторонній потік даних.

Хід роботи

1. Ознайомитися із теоретичними відомостями (виконується до початку лабораторного заняття).
2. Дослідити функціонал, що розглядається у теоретичних відомостях.
3. Виконати практичне завдання, поставлене викладачем (зазвичай суть практичного завдання полягає у розробці невеликого сценарію, функціональність якого визначається темою, що розглядається, ти стислими термінами розробки під час лабораторного заняття).

4. Зробити висновки.
5. Захистити роботу та відповісти на контрольні запитання.

Контрольні запитання

1. Що таке композиція в контексті Jetpack Compose?
2. Що таке рекомпозиція і коли вона відбувається?
3. Яка роль анотації @Composable у процесі композиції?
4. Яким чином зміни стану (State) викликають рекомпозицію?
5. Що таке remember і як він допомагає контролювати рекомпозицію?
6. Як Compose визначає, яку частину UI потрібно оновити під час рекомпозиції?
7. У чому різниця між remember і rememberSaveable з точки зору рекомпозиції?
8. Яку роль відіграє mutableStateOf() у реактивному UI?
9. Чи може рекомпозиція впливати на продуктивність? Як її оптимізувати?
10. Які інструменти або підходи можна використати для відлагодження небажаної рекомпозиції?

Довідкова інформація

1. https://developer.android.com/compose
2. https://developer.android.com/develop/ui/compose/mental-model
3. https://developer.android.com/develop/ui/compose/lifecycle

Лабораторна робота №3. ViewModel

Мета: набути теоретичних знань та практичних навичок з побудови архітектури Android-додатку з використанням ViewModel у контексті Compose.

Завдання: розробити додаток відповідно до отриманої специфікації, продемонструвати оновлення інтерфейса користувача з використанням State, State Hoisting та ViewModel.

ViewModel — це компонент архітектури Android (із Jetpack), призначений для зберігання та управління UI-станом у життєвому циклі Activity або Fragment. У Jetpack Compose ViewModel використовується так само, але частіше — у зв'язці з Composable-функціями, щоб логіка стану була відділена від інтерфейсу.

Основні цілі ViewModel:

1. Відділення логіки від UI (чистота коду).
2. Збереження стану при зміні конфігурації (наприклад, поворот екрана).
3. Спільне використання стану між кількома Composable-функціями.
4. Інкапсуляція бізнес-логіки, запитів до бази даних чи API.

Суть у Compose:

Jetpack Compose — реактивний фреймворк, тому ViewModel найчастіше використовує State або StateFlow / LiveData для надання стану, який може автоматично оновлювати UI при зміні.

Приклад використання ViewModel у Compose

```
class CounterViewModel : ViewModel() {
    private val _count = mutableStateOf(0)
    val count: State<Int> = _count

    fun increment() {
        _count.value++
    }
}
@Composable
fun CounterScreen(viewModel: CounterViewModel = viewModel()) {
    val count by viewModel.count

    Column(horizontalAlignment = Alignment.CenterHorizontally) {
        Text(text = "Count: $count")
        Button(onClick = { viewModel.increment() }) {
            Text("Increment")
        }
    }
}
```

ViewModel() — функція Compose, яка автоматично створює або під'єднує ViewModel, прив'язану до життєвого циклу Activity/Fragment.
Кожна зміна count викликає рекомпозицію Text.

Інші приклади використання ViewModel у Compose:

1. Збереження вводу користувача (форми).
2. Робота з API через Coroutine у ViewModel + LaunchedEffect у Compose.
3. Зберігання вибраного елемента списку або вкладки.

4. Спостереження за Room базою через Flow.

Нижче наведено приклад використання ViewModel із API-запитом у Jetpack Compose з використанням StateFlow і Coroutine. Це — сучасний підхід, який поєднує реактивність Compose та асинхронну обробку даних.

Мета: створити екран, який отримує список користувачів із фейкового API і показує їх у Compose UI.

```
class UserViewModel : ViewModel() {

    private val _users = MutableStateFlow<List<String>>(emptyList())
    val users: StateFlow<List<String>> = _users.asStateFlow()

    private val _isLoading = MutableStateFlow(false)
    val isLoading: StateFlow<Boolean> = _isLoading.asStateFlow()

    init {
        fetchUsers()
    }

    private fun fetchUsers() {
        viewModelScope.launch {
            _isLoading.value = true

            delay(2000) // імітація API виклику

            // Фейкові дані (імітація відповіді з API)
            _users.value = listOf("Alice", "Bob", "Charlie")
            _isLoading.value = false
        }
    }
}
```

2. Composable, який спостерігає за ViewModel

```
@Composable
fun UserListScreen(viewModel: UserViewModel = viewModel()) {
    val users by viewModel.users.collectAsState()
    val isLoading by viewModel.isLoading.collectAsState()

    Column(modifier = Modifier.padding(16.dp)) {
        if (isLoading) {
            CircularProgressIndicator()
        } else {
            users.forEach { name ->
                Text(text = name, style = MaterialTheme.typography.bodyLarge)
            }
        }
```

```
    }
}
```

- viewModelScope.launch {} — запускає корутину всередині ViewModel, яка автоматично скасовується при знищенні ViewModel.
- StateFlow — реактивне джерело даних. Compose автоматично рекомпонує UI при зміні значення.
- collectAsState() — адаптує StateFlow до Compose, щоб можна було спостерігати за значенням.
- delay() — симуляція очікування відповіді від API (в реальному житті — це був би Retrofit або Ktor).

Цей шаблон легко розширити на:

- справжні HTTP-запити (Retrofit, Ktor),
- обробку помилок (try/catch, sealed class Result),
- пагінацію,
- збереження стану (scroll position, selected item).

Хід роботи

6. Ознайомитися із теоретичними відомостями (виконується до початку лабораторного заняття).
7. Дослідити функціонал, що розглядається у теоретичних відомостях.
8. Виконати практичне завдання, поставлене викладачем (зазвичай суть практичного завдання полягає у розробці невеликого сценарію, функціональність якого визначається темою, що розглядається, ти стислими термінами розробки під час лабораторного заняття).
9. Зробити висновки.
10. Захистити роботу та відповісти на контрольні запитання.

Контрольні запитання

1. Що таке ViewModel і яка його роль у Jetpack Compose?
2. Як забезпечується збереження стану у ViewModel при зміні конфігурації в Compose (наприклад, поворот екрану)?
3. У чому різниця між remember / rememberSaveable і ViewModel при управлінні станом?
4. Як підключити ViewModel до Composable-функції у Compose?
5. Як використовувати State або StateFlow у ViewModel для оновлення UI в Compose?
6. Як працює viewModel() функція в Compose, і як вона знає, який ViewModel використовувати?
7. Як передати параметри у ViewModel, якщо він використовується в Composable-функції?
8. Як використовувати ViewModel разом з Navigation у Jetpack Compose?
9. Як тестувати ViewModel, що використовується для керування станом у Compose UI?
10. У яких випадках слід обирати ViewModel замість локального стану (remember) у Jetpack Compose?

Довідкова інформація

1. https://developer.android.com/topic/libraries/architecture/viewmodel

Лабораторна робота №4. Runtime Permission

Мета: набути теоретичних знань та практичних навичок з побудови архітектури Android-додатку з використанням Runtime Permission у контексті Compose.

Завдання: розробити додаток відповідно до отриманої специфікації, продемонструвати функціонування, що передбачає сценарії обробки задоволених чи відхилених запитів на надання дозволів.

Runtime Permission (дозвіл у часі виконання) — це система в Android, яка вимагає, щоб користувач явно дозволив доступ до "небезпечних" ресурсів (наприклад, камера, мікрофон, місцезнаходження) під час роботи програми, а не лише при встановленні. З'явилося з Android 6.0 (API 23), щоб забезпечити контроль користувача над приватними даними.

Небезпечні permission-и, які вимагають підтвердження під час виконання, наприклад:

- CAMERA
- RECORD_AUDIO
- ACCESS_FINE_LOCATION
- READ_CONTACTS
- READ_EXTERNAL_STORAGE (до Android 10)

Основна логіка:

1. Перевірити, чи вже надано дозвіл.
2. Якщо ні — запросити дозвіл.
3. Обробити результат (надано / відмовлено / заборонено назавжди).

AndroidManifest.xml:

```
<uses-permission android:name="android.permission.ACCESS_FINE_LOCATION" />
```

2. Код у Compose:

```
@Composable
fun LocationPermissionScreen() {
    val context = LocalContext.current
    val permission = Manifest.permission.ACCESS_FINE_LOCATION
    val permissionState = rememberPermissionState(permission = permission)

    Column(
        modifier = Modifier
            .fillMaxSize()
            .padding(16.dp),
        horizontalAlignment = Alignment.CenterHorizontally
    ) {
        when {
            permissionState.status.isGranted -> {
                Text("Доступ до місцезнаходження надано")
            }
            else -> {
                Text("Дозвіл не надано")
```

```
            Spacer(modifier = Modifier.height(8.dp))
            Button(onClick = { permissionState.launchPermissionRequest() }) {
                Text("Надати дозвіл")
            }
        }
    }
}
```

Використовується бібліотека:

```
implementation "com.google.accompanist:accompanist-permissions:<version>"
```

Нижче наведено приклад того, як одночасно запросити кілька Runtime Permissions у Jetpack Compose — наприклад, доступ до камери та мікрофона, а також обробити випадок, коли дозвіл відхилено назавжди.

AndroidManifest.xml

```
<uses-permission android:name="android.permission.CAMERA" />
<uses-permission android:name="android.permission.RECORD_AUDIO" />
```

Код у Compose із accompanist-permissions. Використовуємо поняття rememberMultiplePermissionsState() із бібліотеки:

```
implementation "com.google.accompanist:accompanist-permissions:0.28.0"
```

```
@OptIn(ExperimentalPermissionsApi::class)
@Composable
fun MultiPermissionScreen() {
    val permissions = listOf(
        Manifest.permission.CAMERA,
        Manifest.permission.RECORD_AUDIO
    )

    val permissionState = rememberMultiplePermissionsState(permissions = permissions)

    val allGranted = permissionState.allPermissionsGranted
    val permanentlyDenied = permissionState.permissions.any {
        !it.status.isGranted && !it.status.shouldShowRationale
    }

    Column(
```

```
        modifier = Modifier
            .fillMaxSize()
            .padding(16.dp),
        horizontalAlignment = Alignment.CenterHorizontally
    ) {
        when {
            allGranted -> {
                Text("Усі дозволи надано")
            }
            permanentlyDenied -> {
                Text("Деякі дозволи були відхилені назавжди")
                Spacer(Modifier.height(8.dp))
                Button(onClick = {
                    // відкриває системні налаштування
                    val intent = Intent(Settings.ACTION_APPLICATION_DETAILS_SETTINGS)
                    intent.data = Uri.parse("package:" + LocalContext.current.packageName)
                    LocalContext.current.startActivity(intent)
                }) {
                    Text("Відкрити налаштування")
                }
            }
            else -> {
                Text("Потрібні дозволи: камера та мікрофон")
                Spacer(Modifier.height(8.dp))
                Button(onClick = { permissionState.launchMultiplePermissionRequest() }) {
                    Text("Запросити дозволи")
                }
            }
        }
    }
}
```

rememberMultiplePermissionsState() — створює стан для кількох дозволів.

Якщо всі дозволи надано — показується повідомлення про успішність надання.

Якщо хоч один дозвіл відхилено назавжди (shouldShowRationale == false) — пропонується перейти до налаштувань додатку.

Інакше — користувачеві пропонується надати дозволи через системне вікно.

Застереження:

- У Android 13+ деякі дозволи мають окремі правила (наприклад, POST_NOTIFICATIONS).
- У shouldShowRationale == false система розуміє, що дозвіл відхилено назавжди (тобто користувач обрав "не питати знову").

Хід роботи

1. Ознайомитися із теоретичними відомостями (виконується до початку лабораторного заняття).
2. Дослідити функціонал, що розглядається у теоретичних відомостях.

3. Виконати практичне завдання, поставлене викладачем (зазвичай суть практичного завдання полягає у розробці невеликого сценарію, функціональність якого визначається темою, що розглядається, ти стислими термінами розробки під час лабораторного заняття).

4. Зробити висновки.

5. Захистити роботу та відповісти на контрольні запитання.

Контрольні питання

1. Що таке Runtime Permissions у контексті Android і чому вони важливі?
2. Як у Android Compose перевірити, чи надано користувачу необхідний дозвіл?
3. Як запитати дозвіл на використання камери в додатку, створеному з використанням Jetpack Compose?
4. Які відмінності між Normal Permissions та Dangerous Permissions у Android?
5. Як правильно обробляти відмову користувача у наданні дозволу в Android Compose?
6. Які кроки потрібно виконати, щоб працювати з дозволами на геолокацію в Android Compose?
7. Як реалізувати функціональність запиту дозволу для доступу до мікрофона в додатку на Jetpack Compose?
8. Які бібліотеки можна використовувати для спрощення роботи з Runtime Permissions в Android Compose?
9. Як показати користувачу пояснення щодо запиту дозволу на доступ до чутливих даних у додатку?
10. Які методи обробки дозволів доступні в Jetpack Compose для забезпечення зручного UX?

Довідкова інформація

1. https://developer.android.com/training/permissions/requesting

Лабораторна робота №5. Бібліотека Room

Мета: набути теоретичних знань та практичних навичок з побудови архітектури Android-додатку з використанням бібліотеки Room.

Завдання: розробити додаток відповідно до отриманої специфікації, продемонструвати функціонування, що передбачає створення Entity, Dao та Database та використання їх для реалізації CRUD-операцій.

Бібліотека Room в Android — це бібліотека, яка забезпечує зручний та ефективний доступ до бази даних SQLite, спрощуючи роботу з даними на пристрої. Вона є частиною архітектурних компонентів Android і допомагає зберігати, обробляти та запитувати дані в локальній базі даних.

Основне призначення Room:

1. Абстракція над SQLite: Room надає абстракцію, яка дозволяє зручно працювати з базою даних, замість того, щоб вручну писати SQL-запити. Це дозволяє уникнути помилок у SQL-коді і спрощує роботу з базою даних.
2. Підтримка об'єктно-орієнтованого підходу: Room дозволяє зберігати об'єкти Java/Kotlin (наприклад, класи) у базі даних. Кожен клас відображається на таблицю, а поля класів — на стовпці таблиці. Room автоматично конвертує ці об'єкти в записи бази даних і навпаки.
3. Автоматичне оновлення бази даних: Room забезпечує автоматичне оновлення бази даних, коли зміни в структурі даних (наприклад, додавання нових стовпців у таблицю) відображаються у коді без необхідності вручну змінювати SQL-запити.
4. Простий доступ до даних через DAO (Data Access Object): Для роботи з базою даних Room використовує інтерфейси DAO, які визначають методи для взаємодії з базою даних, такі як збереження, видалення, оновлення та запити. Це дозволяє зробити код чистим і організованим.
5. Підтримка асинхронних операцій: Room підтримує асинхронну обробку запитів до бази даних за допомогою LiveData або Flow. Це забезпечує ефективне використання ресурсів та покращує користувацький досвід.
6. Інтеграція з іншими архітектурними компонентами: Room добре інтегрується з іншими компонентами архітектури Android, такими як *ViewModel*, *LiveData* та *Lifecycle*, що дозволяє ефективно працювати з даними та керувати життєвим циклом додатку.

Ключові компоненти бібліотеки Room:

1. @Entity: Означає клас, який буде відображатися на таблицю бази даних.
2. @Dao: Інтерфейс, який містить методи для доступу до даних в базі даних.
3. @Database: Оголошення бази даних, яка включає в себе список всіх Entity та DAO.

Room робить доступ до локальних даних простішим, безпечнішим і ефективнішим. Це дозволяє розробникам зосередитись на бізнес-логіці замість того, щоб писати складні SQL-запити вручну.

Щоб створити Entity, DAO і Database у Android за допомогою бібліотеки Room в контексті Android Compose, потрібно виконати кілька кроків. Ось як це зробити:

1. Додати залежності в build.gradle

Спершу потрібно додати залежності для Room у файлі build.gradle (Module: app):

```
dependencies {
    def room_version = "2.5.0"  // Заміни на останню версію

    implementation "androidx.room:room-runtime:$room_version"
```

```
    annotationProcessor "androidx.room:room-compiler:$room_version" // Для Java
    kapt "androidx.room:room-compiler:$room_version" // Для Kotlin

    // Для підтримки корутин (необов'язково)
    implementation "androidx.room:room-ktx:$room_version"
}
```

Не забудьте додати плагін для Kotlin KAPT у верхній частині файлу build.gradle:

```
apply plugin: 'kotlin-kapt'
```

2. Створення Entity класу

Entity — це клас, який буде відображатися на таблицю в базі даних. Для цього потрібно використовувати анотацію @Entity і визначити поля, які будуть стовпцями цієї таблиці.

```
import androidx.room.Entity
import androidx.room.PrimaryKey

@Entity(tableName = "users")
data class User(
    @PrimaryKey(autoGenerate = true) val id: Long = 0,
    val name: String,
    val age: Int
)
```

У цьому прикладі ми створили таблицю users, де id — це унікальний ключ, а name та age — це стовпці таблиці.

3. Створення DAO (Data Access Object)

DAO визначає методи для доступу до бази даних. DAO містить методи для додавання, оновлення, видалення та запитів до даних.

```
import androidx.room.Dao
import androidx.room.Insert
import androidx.room.Query

@Dao
interface UserDao {

    @Insert
    suspend fun insert(user: User)

    @Query("SELECT * FROM users WHERE id = :id")
    suspend fun getUserById(id: Long): User?
```

```
    @Query("SELECT * FROM users")
    suspend fun getAllUsers(): List<User>

    @Query("DELETE FROM users WHERE id = :id")
    suspend fun deleteUserById(id: Long)
}
```

У цьому прикладі ми створили кілька методів для роботи з таблицею users: вставка нового користувача, отримання користувача за ID, отримання всіх користувачів і видалення користувача.

4. Створення Database класу

Клас Database є абстракцією для роботи з базою даних. Він об'єднує всі Entity та DAO.

```
import android.content.Context
import androidx.room.Database
import androidx.room.Room
import androidx.room.RoomDatabase

@Database(entities = [User::class], version = 1)
abstract class AppDatabase : RoomDatabase() {

    abstract fun userDao(): UserDao

    companion object {
        @Volatile
        private var INSTANCE: AppDatabase? = null

        fun getDatabase(context: Context): AppDatabase {
            return INSTANCE ?: synchronized(this) {
                val instance = Room.databaseBuilder(
                    context.applicationContext,
                    AppDatabase::class.java,
                    "app_database"
                ).build()
                INSTANCE = instance
                instance
            }
        }
    }
}
```

Тут створено базу даних, яка містить таблицю users (через Entity) і доступ до DAO через метод userDao().

5. Використання бази даних у Compose

Тепер, коли є Entity, DAO і Database, можна використовувати їх у Compose-додаткові.

Нижче нведено приклад використання Room для додавання і отримання даних в Jetpack Compose:

```
import android.os.Bundle
import androidx.activity.ComponentActivity
import androidx.activity.compose.setContent
import androidx.compose.foundation.layout.Column
import androidx.compose.foundation.layout.padding
import androidx.compose.material3.Button
import androidx.compose.material3.Text
import androidx.compose.runtime.Composable
import androidx.compose.runtime.LaunchedEffect
import androidx.compose.ui.Modifier
import androidx.compose.ui.unit.dp
import androidx.lifecycle.ViewModel
import androidx.lifecycle.viewModelScope
import androidx.room.Room
import kotlinx.coroutines.launch

class UserViewModel : ViewModel() {

                    private val db = AppDatabase.getDatabase(context = ApplicationProvider.getApplicationContext())
    private val userDao = db.userDao()

    fun insertUser(user: User) {
        viewModelScope.launch {
            userDao.insert(user)
        }
    }

    fun getAllUsers() {
        viewModelScope.launch {
            val users = userDao.getAllUsers()
            // Do something with users, e.g., update UI
        }
    }
}

@Composable
fun UserScreen(viewModel: UserViewModel) {
    Column(modifier = Modifier.padding(16.dp)) {
        Button(onClick = { viewModel.insertUser(User(name = "John", age = 25)) }) {
            Text("Insert User")
        }
        // Display users from database or other UI updates
    }
}

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

```
            UserScreen(viewModel = UserViewModel())
        }
    }
}
```

Пояснення:

- UserViewModel: містить логіку для взаємодії з базою даних. Використовує корутини для асинхронного виконання операцій (вставка та отримання даних).
- UserScreen: це основний UI-екран, який дозволяє додавати користувачів у базу даних.

Використання Room у Android Compose дозволяє зручно інтегрувати локальну базу даних у додаток, абстрагуючи низькорівневі операції SQL. Ви можете легко створювати Entity, DAO та Database, а також обробляти операції асинхронно за допомогою LiveData або Flow для забезпечення плавного досвіду користувача.

Щоб скористатися бібліотекою Room разом із Hilt в Android Compose, потрібно налаштувати ін'єкцію залежностей за допомогою Hilt, що значно спрощує доступ до бази даних. Hilt дозволяє автоматично управляти створенням і життєвим циклом об'єктів, таких як DAO та база даних, і забезпечує їх ін'єкцію в компоненти вашого додатку.

Нижче наведено покроковий приклад того, як можна налаштувати Room разом з Hilt в Android Compose:

1. Додати залежності для Hilt і Room

Спершу потрібно додати залежності для Room та Hilt у build.gradle (Module: app) файлі.

```
// build.gradle (Module: app)
plugins {
    id 'kotlin-kapt'
    id 'dagger.hilt.android.plugin'
}

dependencies {
    // Залежності для Hilt
    implementation "androidx.hilt:hilt-lifecycle-viewmodel:1.0.0"
    implementation "com.google.dagger:hilt-android:2.45"
    kapt "com.google.dagger:hilt-android-compiler:2.45"

    // Залежності для Room
    def room_version = "2.5.0"  // Заміни на останню версію
    implementation "androidx.room:room-runtime:$room_version"
    implementation "androidx.room:room-ktx:$room_version"
    kapt "androidx.room:room-compiler:$room_version"
}
Не забудьте додати плагін для Hilt:
apply plugin: 'dagger.hilt.android.plugin'
```

2. Налаштування Room Database та DAO з Hilt

Далі, створимо клас Room Database та DAO, і налаштуємо їх для ін'єкції через Hilt.

Створення Entity:

```
import androidx.room.Entity
import androidx.room.PrimaryKey

@Entity(tableName = "users")
data class User(
    @PrimaryKey(autoGenerate = true) val id: Long = 0,
    val name: String,
    val age: Int
)
```

Створення DAO:

```
import androidx.room.Dao
import androidx.room.Insert
import androidx.room.Query

@Dao
interface UserDao {

    @Insert
    suspend fun insert(user: User)

    @Query("SELECT * FROM users")
    suspend fun getAllUsers(): List<User>

    @Query("SELECT * FROM users WHERE id = :id")
    suspend fun getUserById(id: Long): User?

    @Query("DELETE FROM users WHERE id = :id")
    suspend fun deleteUserById(id: Long)
}
```

Створення Database:

```
import android.content.Context
import androidx.room.Database
import androidx.room.Room
import androidx.room.RoomDatabase
import dagger.hilt.android.qualifiers.ApplicationContext
import javax.inject.Inject
import javax.inject.Singleton

@Database(entities = [User::class], version = 1)
abstract class AppDatabase : RoomDatabase() {
    abstract fun userDao(): UserDao
}
```

```
@Singleton
class AppDatabaseProvider @Inject constructor(
    @ApplicationContext private val context: Context
) {
    private var INSTANCE: AppDatabase? = null

    fun getDatabase(): AppDatabase {
        return INSTANCE ?: synchronized(this) {
            val instance = Room.databaseBuilder(
                context.applicationContext,
                AppDatabase::class.java,
                "app_database"
            ).build()
            INSTANCE = instance
            instance
        }
    }
}
```

3. Налаштування Hilt для ін'єкції в ViewModel

Створимо ViewModel, який буде використовувати ін'єкцію залежностей для доступу до UserDao.

Створення ViewModel:

```
import androidx.lifecycle.ViewModel
import androidx.lifecycle.viewModelScope
import dagger.hilt.android.lifecycle.HiltViewModel
import kotlinx.coroutines.launch
import javax.inject.Inject

@HiltViewModel
class UserViewModel @Inject constructor(
    private val userDao: UserDao
) : ViewModel() {

    fun insertUser(user: User) {
        viewModelScope.launch {
            userDao.insert(user)
        }
    }

    fun getAllUsers() {
        viewModelScope.launch {
            val users = userDao.getAllUsers()
            // Update UI with users
        }
    }
```

```
}
```

Важливі моменти:

- @HiltViewModel: ця анотація дозволяє створювати ViewModel, який підтримує ін'єкцію залежностей за допомогою Hilt.
- @Inject constructor: дозволяє Hilt автоматично передавати залежності в конструктор ViewModel.

4. Налаштування Hilt для ін'єкції в Activity або Composable функції

У Activity або Composable функції потрібно використовувати ін'єкцію залежностей через Hilt.

Створення Composable екрану з ін'єкцією через ViewModel:

```
import android.os.Bundle
import androidx.activity.ComponentActivity
import androidx.activity.compose.setContent
import androidx.compose.foundation.layout.Column
import androidx.compose.foundation.layout.padding
import androidx.compose.material3.Button
import androidx.compose.material3.Text
import androidx.compose.runtime.Composable
import androidx.compose.ui.Modifier
import androidx.compose.ui.unit.dp
import androidx.hilt.navigation.compose.hiltViewModel
import dagger.hilt.android.AndroidEntryPoint

@AndroidEntryPoint
class MainActivity : ComponentActivity() {
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContent {
            UserScreen()
        }
    }
}

@Composable
fun UserScreen(userViewModel: UserViewModel = hiltViewModel()) {
    Column(modifier = Modifier.padding(16.dp)) {
        Button(onClick = {
            userViewModel.insertUser(User(name = "John", age = 25))
        }) {
            Text("Insert User")
        }
        // Display users from database or other UI updates
    }
}
```

Важливі моменти:

- @AndroidEntryPoint: ця анотація дозволяє Hilt ін'єктувати залежності в Activity чи Fragment.
- hiltViewModel(): використовується для ін'єкції ViewModel у Composable.

5. Завершення налаштування

Тепер, коли налаштувано Room, DAO, Database та ViewModel з Hilt, додаток буде використовувати ін'єкцію залежностей для автоматичного надання потрібних об'єктів у компоненти, зокрема в ViewModel та Activity.

Це значно спрощує код, оскільки Hilt бере на себе управління життєвим циклом залежностей, і розробникові не потрібно вручну створювати об'єкти або передавати їх через конструктори.

Підсумок

1. Room надає простий API для роботи з базою даних SQLite.
2. Hilt автоматизує ін'єкцію залежностей, спрощуючи керування об'єктами.
3. Завдяки Room та Hilt, ви можете ефективно працювати з базою даних без необхідності вручну створювати екземпляри об'єктів чи керувати їхнім життєвим циклом.

Хід роботи

1. Ознайомитися із теоретичними відомостями (виконується до початку лабораторного заняття).
2. Дослідити функціонал, що розглядається у теоретичних відомостях.
3. Виконати практичне завдання, поставлене викладачем (зазвичай суть практичного завдання полягає у розробці невеликого сценарію, функціональність якого визначається темою, що розглядається, ти стислими термінами розробки під час лабораторного заняття).
4. Зробити висновки.
5. Захистити роботу та відповісти на контрольні запитання.

Контрольні питання

1. Що таке бібліотека Room в Android і як вона допомагає працювати з базами даних?
2. Яка основна відмінність між бібліотеками Room та SQLite в Android?
3. Як створити Entity клас для роботи з таблицею в базі даних за допомогою Room?
4. Як налаштувати DAO (Data Access Object) для взаємодії з базою даних у Room?
5. Як можна використовувати Room з LiveData або Flow для спостереження за змінами в базі даних?
6. Які типи анотацій використовуються в Room для визначення таблиць, стовпців і методів DAO?
7. Як можна реалізувати асинхронне додавання, оновлення та видалення даних за допомогою Room у додатку на Jetpack Compose?
8. Як забезпечити безпечне оновлення схеми бази даних (міграцію) при зміні структури таблиць у Room?
9. Як інтегрувати бібліотеку Room з Hilt для автоматичної ін'єкції залежностей?
10. Як обробляти помилки та виключення, що можуть виникнути при роботі з базою даних за допомогою Room в Android Compose?

Довідкова інформація

1. https://developer.android.com/training/data-storage/room
2. https://developer.android.com/jetpack/androidx/releases/room

Лабораторна робота №6. HTTP-запити. Бібліотека Ktor

Мета: набути теоретичних знань та практичних навичок з побудови архітектури Android-додатку з використанням бібліотеки Ktor.

Завдання: розробити додаток відповідно до отриманої специфікації, продемонструвати функціонування, що передбачає надсилання запитів та отримання відповідей від WEB сервера за заданим API.

Ktor — це фреймворк для роботи з мережевими запитами на Kotlin, який дозволяє створювати HTTP-сервери та клієнти. У контексті Android, Ktor зазвичай використовується для створення HTTP-клієнтів, що взаємодіють з віддаленими сервісами (наприклад, для взаємодії з REST API). Ktor забезпечує асинхронність за допомогою Kotlin корутин і підтримує різні формати серіалізації (наприклад, JSON, XML).

У випадку Android Compose, ви можете використовувати Ktor для асинхронної взаємодії з мережею, отримання даних і їх подальшої обробки для відображення в інтерфейсі користувача. Ktor також дуже зручний для роботи з RESTful API, що є типово необхідним при розробці мобільних додатків.

Основні можливості Ktor в Android:

- HTTP-клієнт для відправки запитів і обробки відповідей.
- Асинхронні запити за допомогою корутин.
- Можливість серіалізації та десеріалізації JSON-даних.
- Підтримка аутентифікації та авторизації (наприклад, Basic Authentication, OAuth2).
- Обробка помилок та робота з різними кодами відповідей HTTP.

Приклад використання Ktor в Android Compose

1. Додавання залежностей у build.gradle

Спершу потрібно додати необхідні залежності у файлі build.gradle:

```
dependencies {
    // Для використання Ktor клієнта
    implementation "io.ktor:ktor-client-android:2.2.0"
    implementation "io.ktor:ktor-client-json:2.2.0"
    implementation "io.ktor:ktor-client-serialization:2.2.0"

    // Для Kotlin корутин
    implementation "org.jetbrains.kotlinx:kotlinx-coroutines-android:1.6.4"
}
```

2. Створення моделі даних для API

Припустімо, що ми працюємо з API, який повертає список користувачів у JSON-форматі. Спершу створимо модель для десеріалізації JSON-даних.

```
import kotlinx.serialization.SerialName
import kotlinx.serialization.Serializable

@Serializable
data class User(
```

```
    val id: Int,
    val name: String,
    val username: String,
    val email: String
)
```

3. Створення клієнта Ktor

Далі створимо Ktor client, який буде виконувати запити до API. Використовуватимемо HttpClient для надсилання запитів і отримання відповідей.

```
import io.ktor.client.*
import io.ktor.client.engine.android.Android
import io.ktor.client.request.*
import io.ktor.client.features.json.JsonFeature
import io.ktor.client.features.json.serializer.KotlinxSerializer
import kotlinx.coroutines.Dispatchers
import kotlinx.coroutines.withContext
import kotlinx.serialization.json.Json

class ApiService {
    private val client = HttpClient(Android) {
        install(JsonFeature) {
            serializer = KotlinxSerializer(Json { ignoreUnknownKeys = true })
        }
    }

    // Функція для отримання списку користувачів
    suspend fun getUsers(): List<User> {
        return withContext(Dispatchers.IO) {
            client.get("https://jsonplaceholder.typicode.com/users")
        }
    }
}
```

4. Створення ViewModel для роботи з даними

Тепер створимо ViewModel, який буде використовувати наш ApiService для отримання даних і передавати їх до UI.

```
import androidx.lifecycle.ViewModel
import androidx.lifecycle.viewModelScope
import kotlinx.coroutines.launch
import kotlinx.coroutines.flow.MutableStateFlow
import kotlinx.coroutines.flow.StateFlow

class UserViewModel : ViewModel() {
```

```
    private val _users = MutableStateFlow<List<User>>(emptyList())
    val users: StateFlow<List<User>> = _users

    private val apiService = ApiService()

    init {
        fetchUsers()
    }

    private fun fetchUsers() {
        viewModelScope.launch {
            val userList = apiService.getUsers()
            _users.value = userList
        }
    }
}
```

5. Відображення даних у Compose UI

Тепер ми можемо використовувати Jetpack Compose, щоб відобразити список користувачів на екрані. Використовуємо collectAsState() для збору даних з StateFlow і оновлення UI.

```
import android.os.Bundle
import androidx.activity.ComponentActivity
import androidx.activity.compose.setContent
import androidx.compose.foundation.layout.Column
import androidx.compose.foundation.layout.padding
import androidx.compose.foundation.lazy.LazyColumn
import androidx.compose.material3.Button
import androidx.compose.material3.Text
import androidx.compose.runtime.Composable
import androidx.compose.runtime.collectAsState
import androidx.compose.ui.Modifier
import androidx.compose.ui.unit.dp
import androidx.hilt.navigation.compose.hiltViewModel
import androidx.lifecycle.viewmodel.compose.viewModel

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

```
@Composable
fun UserScreen(userViewModel: UserViewModel = viewModel()) {
    val users = userViewModel.users.collectAsState()

    Column(modifier = Modifier.padding(16.dp)) {
        Text(text = "Users List:")
        LazyColumn {
            items(users.value) { user ->
                UserItem(user)
            }
        }
    }
}

@Composable
fun UserItem(user: User) {
    Column(modifier = Modifier.padding(8.dp)) {
        Text(text = user.name)
        Text(text = user.email)
    }
}
```

6. Пояснення кроків:

1. ApiService: Клас для налаштування HTTP-клієнта і відправлення запитів до API за допомогою Ktor.
2. ViewModel: Клас для обробки бізнес-логіки та отримання даних від API. Він зберігає список користувачів у StateFlow.
3. Compose UI: Компоненти для відображення списку користувачів на екрані, оновлюючи UI при отриманні нових даних.

7. Переваги використання Ktor з Android Compose:

- Асинхронність: Завдяки корутинам у Kotlin, Ktor дозволяє виконувати мережеві запити без блокування основного потоку, що підвищує продуктивність.
- Легка інтеграція з Compose: Завдяки використанню StateFlow для спостереження за даними, Ktor легко інтегрується з Jetpack Compose для динамічного оновлення UI.
- Серійалізація та десеріалізація: Ktor підтримує Kotlinx.serialization, що полегшує роботу з JSON даними.
- Гнучкість та модульність: Ви можете легко налаштувати різні плагіни і функціональність у Ktor, наприклад, додавати аутентифікацію, керувати заголовками запитів тощо.

Ktor в Android Compose є потужним інструментом для роботи з мережею. Завдяки асинхронному характеру та інтеграції з Kotlin корутинами, він дозволяє ефективно виконувати запити до віддалених серверів без блокування основного потоку, що особливо важливо для мобільних додатків. Ви можете використовувати Ktor для взаємодії з REST API, обробки JSON-даних і оновлення UI в реальному часі.

Хід роботи

1. Ознайомитися із теоретичними відомостями (виконується до початку лабораторного заняття).
2. Дослідити функціонал, що розглядається у теоретичних відомостях.
3. Виконати практичне завдання, поставлене викладачем (зазвичай суть практичного завдання полягає у розробці невеликого сценарію, функціональність якого визначається темою, що розглядається, ти стислими термінами розробки під час лабораторного заняття).
4. Зробити висновки.
5. Захистити роботу та відповісти на контрольні запитання.

Контрольні запитання

1. Що таке Ktor і як його можна використовувати для створення HTTP-клієнтів в Android Compose?
2. Як налаштувати Ktor для роботи з REST API в Android додатку на Jetpack Compose?
3. Як в Ktor здійснюється асинхронне виконання HTTP-запитів, і чому це важливо для мобільних додатків?
4. Як налаштувати Ktor для роботи з JSON за допомогою Kotlinx.serialization в Android Compose?
5. Як обробляти помилки мережевих запитів у Ktor в Android Compose?
6. Як працює аутентифікація в Ktor (наприклад, Basic Authentication або OAuth2) в Android Compose?
7. Як у Ktor реалізувати підтримку SSL/TLS для безпечних з'єднань у Android додатку?
8. Як інтегрувати Ktor з ViewModel та StateFlow для отримання даних з API в Android Compose?
9. Як працювати з багатозадачністю у Ktor в Android, використовуючи Kotlin корутини для ефективного управління запитами?
10. Як тестувати HTTP-запити, виконувані за допомогою Ktor, в Android Compose (наприклад, з використанням моків чи інтеграційних тестів)?

Довідкова інформація

1. https://ktor.io/
2. https://ktor.io/docs/client-create-multiplatform-application.html