Сучасні технології мобільного програмування

Архітектура Jetpack Compose застосунку

Слайди до лекцій (3 змістовий модуль)

Архітектура програми

- **Архітектура програми** передбачає розподіл обов'язків між частинами програми. Добре продумана архітектура програми допомагає масштабувати програму, та розширювати її додатковими функціями. Архітектура також може спростити співпрацю в команді.
- Архітектура програми реалізується розподілом функцій програми, реалізованих програмними частинами, за різними шарами, які часто виділяють як окремі пакети.

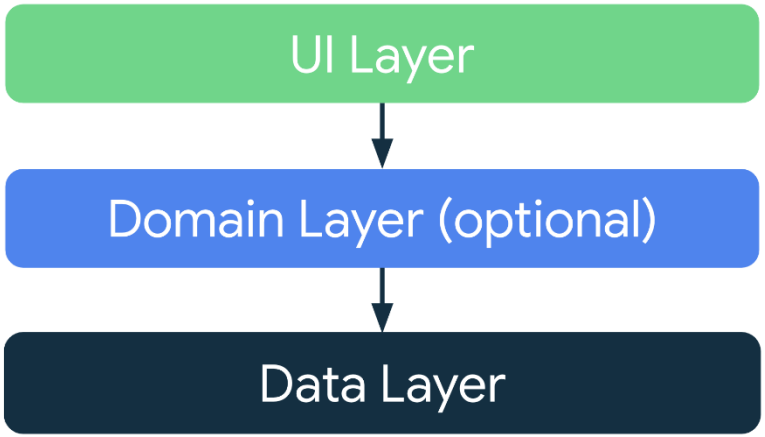
BRANCH LAB3



Архітектура програми - шари

Рекомендована архітектура програми передбачає щонайменше два шари:

- **Шар інтерфейсу користувача (UI Layer)**: шар, який відображає дані програми на екрані, але є незалежним від даних.
- **Шар даних (Data Layer)**: шар, який зберігає, отримує та надає доступ до даних програми.
- Також можна додати ще один рівень, який називається **шаром домену (Domain Layer)**, щоб спростити та повторно використовувати взаємодію між рівнями інтерфейсу користувача та даних.

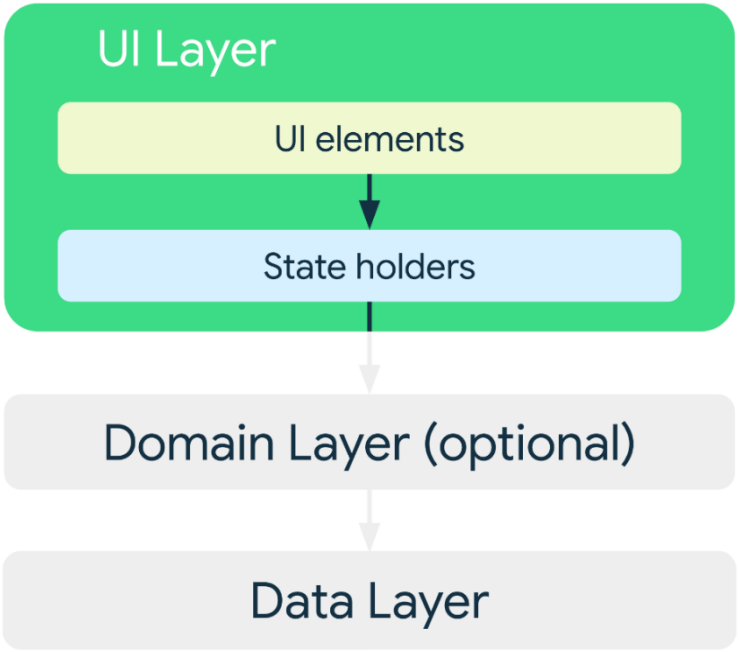


Стрілки визначають залежності між класами. Наприклад, рівень домену залежить від класів рівня даних.

Архітектура програми - UI Layer

Шар інтерфейсу користувача складається з таких компонентів:

- ***Елементи інтерфейсу користувача (UI Elements)***: компоненти, які відображають дані на екрані (створюються за допомогою Jetpack Compose).
- ***Власники станів (State Holders)*:** компоненти, які зберігають стан UI та логіку, у якій використовується стан (***UI-логіку***).
- Таким чином шар елементів UI залежить від шару власників стану.

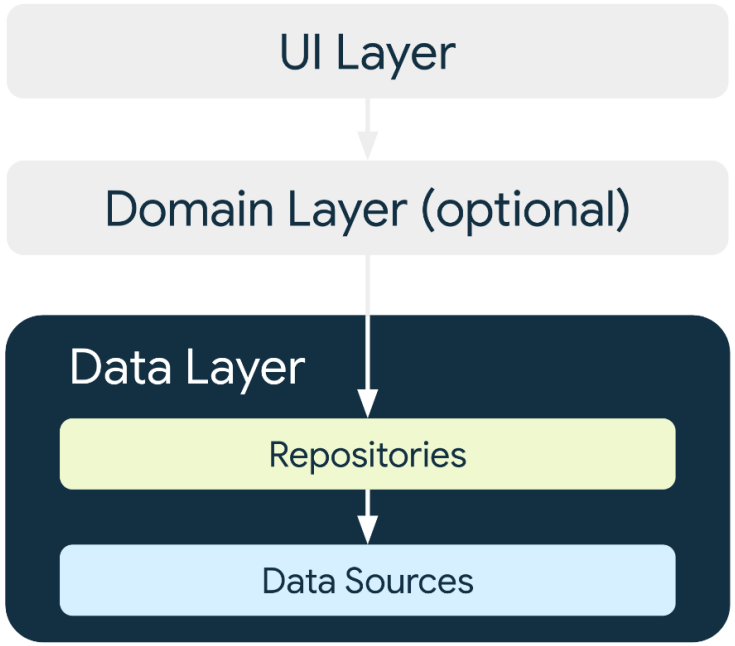


Приклади UI-логіки:

- отримання тексту, який повинен з'явитися на екрані, з використанням рядкових ресурсів,
- виведення повідомлень за допомогою Toast/Snackbar,
- навігація до нового екрану після кліку на кнопці
- тощо.

Архітектура програми - Data Layer

- **Шар даних** містить **дані програми (application data)** та **бізнес-логіку (Buisiness logic)**, які визначають, як дані мають створюватися, змінюватися, зберігатися та видалятися. (фактично це реалізація вимог до програми).
- **Шар даних** складається з **репозиторіїв (Repositories)**, кожен з яких може містити від нуля до багатьох **джерел даних (Data Source)**. Необхідно створювати репозиторії для кожного типу даних, які обробляться програмою.

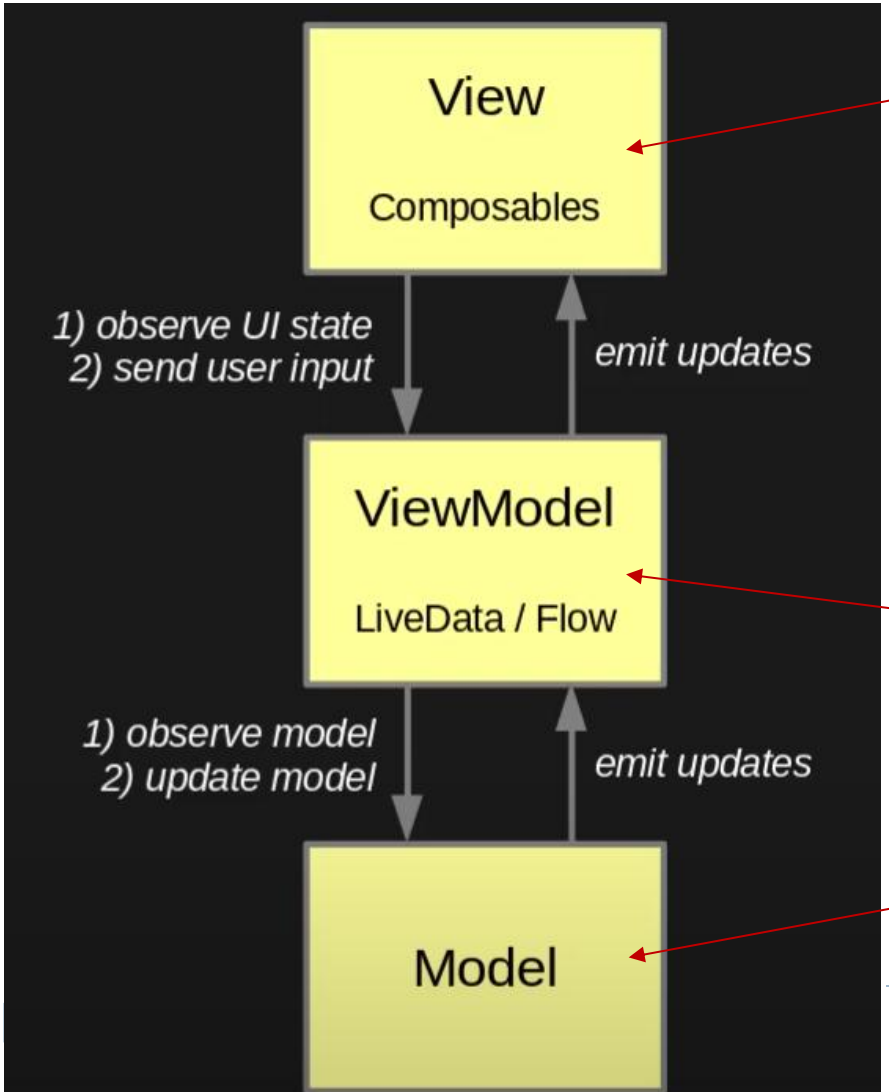


Приклади бізнес-логіки:

- додання/редагування/видалення питання тесту до банку питань,
- підрахунок правильних відповідей з розрахунком загальної оцінки за тест,
- тощо.

Архітектура MVVM

- **MVVM - Model View ViewModel** - архітектурний шаблон

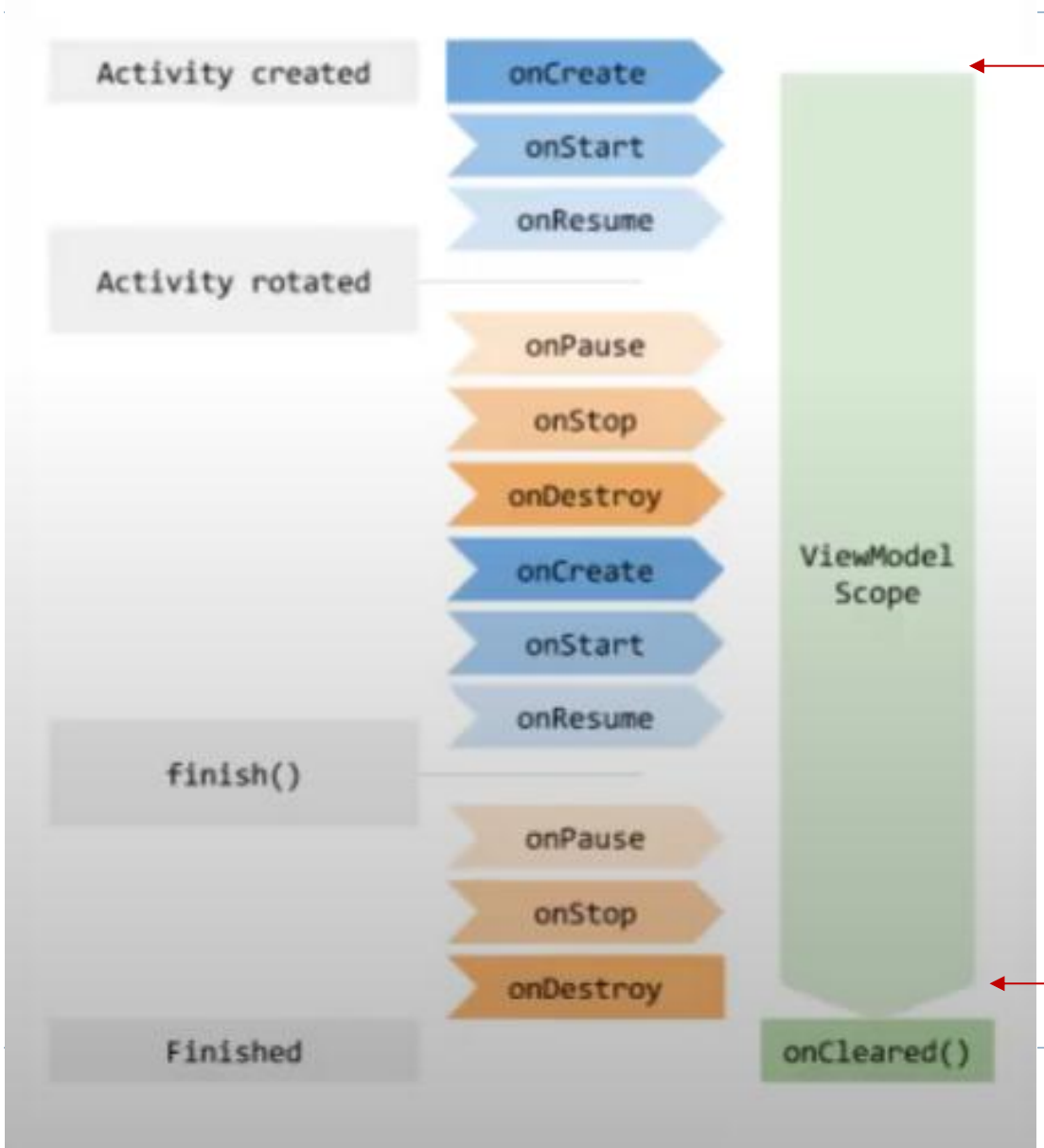


Відображає екран на основі даних від ViewModel

Отримує дані від Model і перетворює їх до вигляду, зручного для використання у View, переживає зміни конфігурації, має спрощений життєвий цикл (див. наст. слайд)

Не регламентується патерном. Часто - репозиторій з даними

Життєвий цикл активності та ViewModel

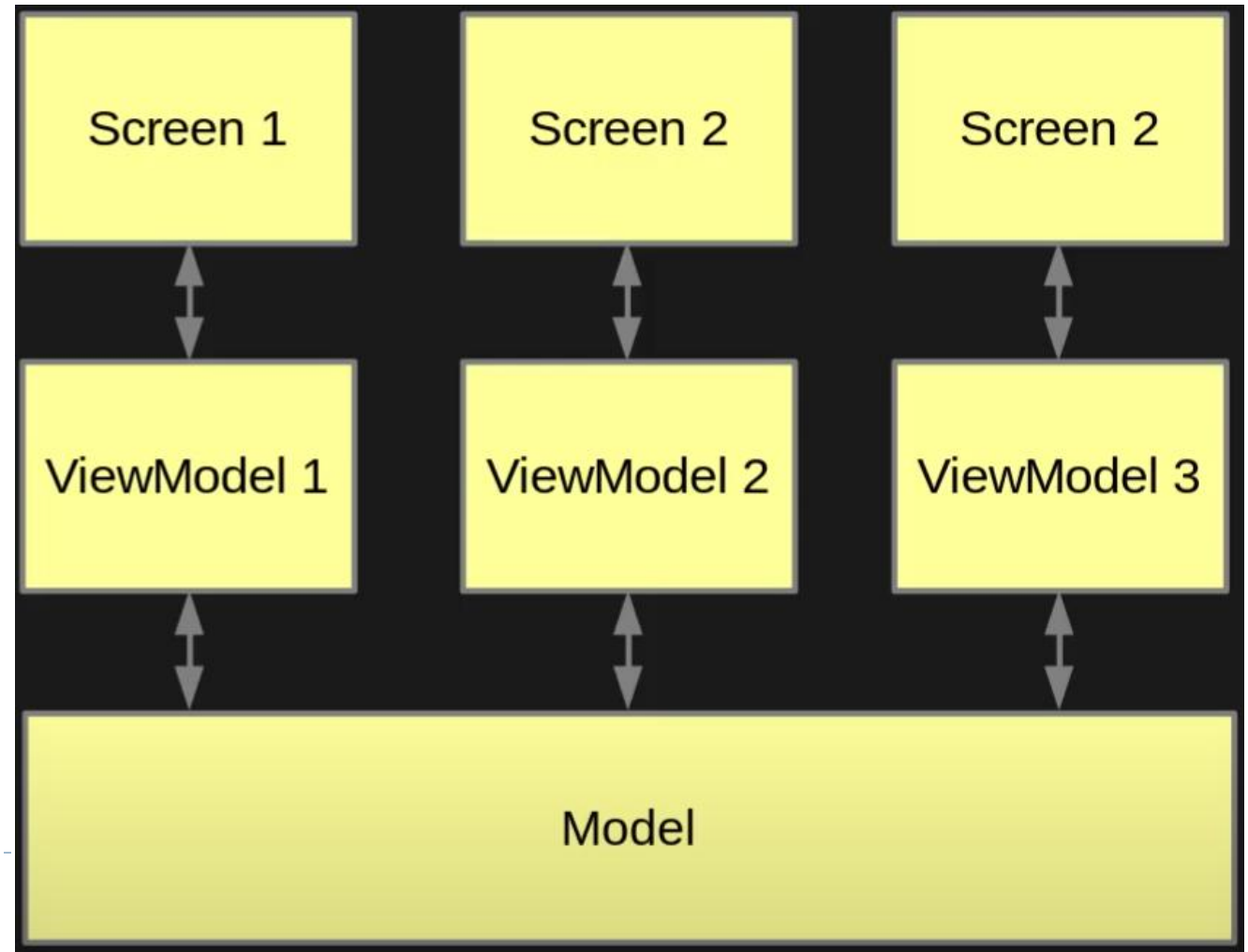


ViewModel created

ViewModel destroyed

ViewModel

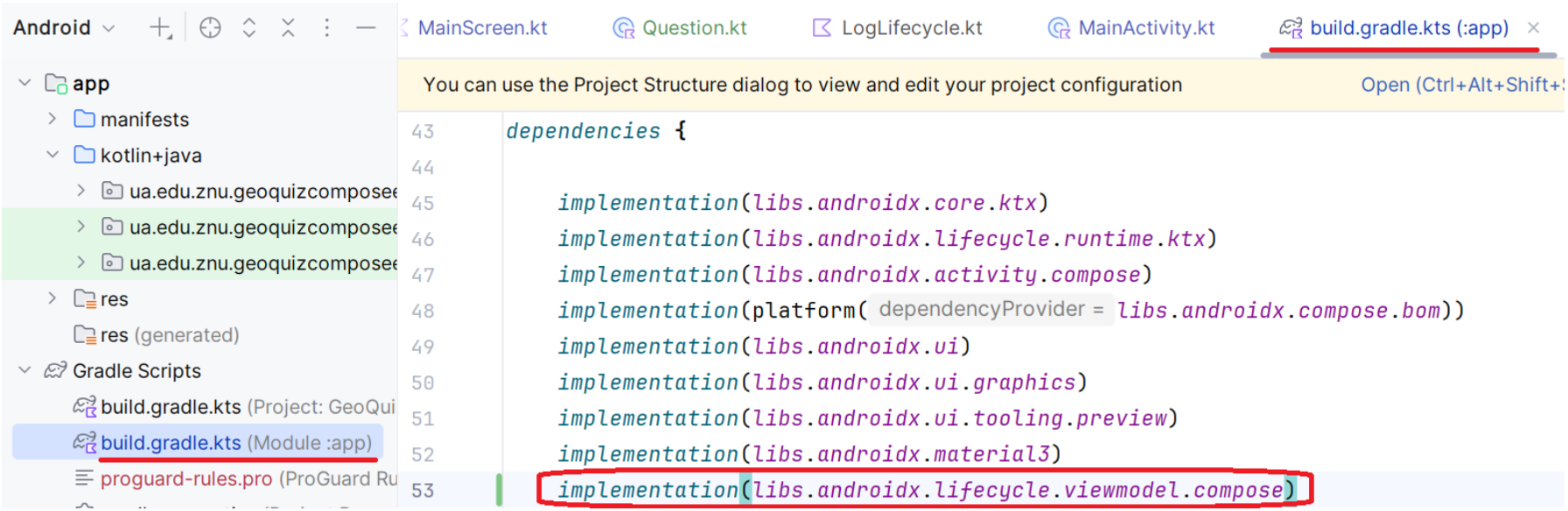
- Зазвичай кожна ViewModel асоціюється з одним екраном



Add dependency

androidx.lifecycle.lifecycle-viewmodel-compose





MutableState vs StateFlow

- ***MutableState*** **-** це механізм спостереження за станом, тісно пов'язаний із Jetpack Compose: коли його значення змінюється, Compose запускає рекомпозицію.
- ***StateFlow*** - представляє стан лише з одним оновлюваним значенням даних (value), надсилає оновлення значення своїм колекторам.
- У ViewModel зазвичай використовують приватний змінюваний **MutableStateFlow**, а назовні надають лише його незмінне значення **StateFlow**:

```
private val _mainScreenState = MutableStateFlow(MainScreenState())
val mainScreenState: StateFlow<MainScreenState> = _mainScreenState.asStateFlow()
```

- Це забезпечує контроль стану тільки об'єктом ViewModel і не дозволяє UI напряму змінювати стан (UI може тільки читати значення StateFlow).
- Така організація коду називається **принципом інкапсуляції** стану.



MainScreenViewModel

```
package ua.edu.znu.geoquizcomposeedu.ui.viewmodel

import …

class MainScreenViewModel : ViewModel() {
    private val questionBank = listOf(
        Question(R.string.question_australia, true),
        …
    )
    private val _mainScreenState = MutableStateFlow(
        MainScreenState(
            currentQuestionId = questionBank.first().textResId
        )
    )
    val mainScreenState: StateFlow<MainScreenState> = _mainScreenState.asStateFlow()

…
```

успадковує androidx.lifecycle.ViewModel

перелік питань переміщений з UI

змінюваний стан MainScreen у ViewModel, ініціалізується першим питанням

незмінюваний стан - надається Composable

- Це шаблон інкапсуляції: єдине джерело достовірної інформації, запобігає зовнішнім мутаціям, безпечніший односторонній потік даних.

MainScreenViewModel

використовується update функція StateFlow для атомарного оновлення MutableState.value зазначеною функцією

```
...
    fun onNextQuestionButtonClick() {
        _mainScreenState.update { currentState ->
            val nextIndex = (currentState.currentIndex + 1) % questionBank.size
            currentState.copy(
                currentIndex = nextIndex,
                currentQuestionId = questionBank[nextIndex].textResId
            )
        }
    }

    fun onAnswerButtonClick(isTrue: Boolean): Int {
        val currentQuestion = questionBank[_mainScreenState.value.currentIndex]
        return if (isTrue == currentQuestion.answer) {
            R.string.correct_toast
        } else {
            R.string.incorrect_toast
        }
    }
}
```

Збір стану в Composable (Міст до UI)

- StateFlow - це реактивний потік на базі корутин, але Compose не вміє читати потоки напряму, йому потрібен звичайний об'єкт стану (State).
- Для цього у Composable використовується функція розширення collectAsStateWithLifecycle().
- Функція collectAsStateWithLifecycle() під капотом запускає окрему корутину, прив'язану до життєвого циклу екрана.
- Ця корутина автоматично призупиняє збір даних, коли застосунок згортається (переходить у фоновий режим). Це критично важливо: якщо програма згорнута, корутина засинає і не витрачає заряд батареї та ресурси процесора.
- Щойно у ViewModel через _uiState.update змінюється значення, StateFlow «проштовхує» нове значення крізь потік.
- UI-корутина ловить це значення, оновлює внутрішній State у Compose, що моментально викликає рекомпозицію (перемальовування) екрана.



MainScreen

```
package ua.edu.znu.geoquizcomposeedu.ui.screens

import ...


data class MainScreenState(
    val currentIndex: Int = 0
    val currentQuestionId: Int
)

@Composable
fun MainScreen(
    innerPadding: PaddingValues,
) {
    val viewModel: MainScreenViewModel = viewModel()
    val mainScreenState by viewModel.mainScreenState.collectAsStateWithLifecycle()
    val context = LocalContext.current

...
```

Видалені анотація @Parcelize та успадкування від Parcelable, оскільки тепер стан створюється та зберігається у ViewModel

Додана властивість, що використовуватиметься у MainScreen

Створює або використовує існуючу MainScreenViewModel з областю видимості - Activity, що запускає цю Composable (або NavBackStackEntry при використанні Jetpack Navigation)

Збирає значення з Flow та конвертує їх у Compose State. Це виконується на етапі життєвого циклу активності, що запускає Composable, між onStart() та onStop().



MainScreen

```
...
    Column(
        modifier = Modifier
            .fillMaxSize()
            .padding(innerPadding),
        verticalArrangement = Arrangement.Center,
        horizontalAlignment = Alignment.CenterHorizontally
    ) {
        Text(
            text = stringResource(id = mainScreenState.currentQuestionId),
            textAlign = TextAlign.Center,
        )
...
```

Додана до MainScreenState властивість



MainScreen

```
...
Row(
    modifier = Modifier.size(width = 200.dp , height = 90.dp),
    horizontalArrangement = Arrangement.SpaceEvenly,
    verticalAlignment = Alignment.CenterVertically
) {
    Button(
        onClick = {
            val messageId = viewModel.onAnswerButtonClick(true)
            Toast.makeText(context, messageId, Toast.LENGTH_SHORT).show()
        }) {
        Text(stringResource(id = R.string.true_button))
    }
    Spacer(modifier = Modifier.width(16.dp))
    Button(
        onClick = {
            val messageId = viewModel.onAnswerButtonClick(false)
            Toast.makeText(context, messageId, Toast.LENGTH_SHORT).show()          }) {
        Text(stringResource(id = R.string.false_button))
    }      } ...
```

MainScreen

```
...
        Spacer(modifier = Modifier.width(16.dp))
        Button(onClick = {
            onClick = viewModel::onNextQuestionButtonClick
        }) {
            Text(stringResource(id = R.string.next_button))
        }
    }
}
```

```
@Preview(showBackground = true)
@Composable
fun MainScreenPreview() {
    val innerPadding = PaddingValues(16.dp)
    MainScreen(innerPadding)
}
```



MainActivity

```
package ua.edu.znu.geoquizcomposeedu

import ...

class MainActivity : ComponentActivity() {
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        enableEdgeToEdge()
        setContent {
            GeoQuizComposeEduTheme {
                Scaffold(modifier = Modifier.fillMaxSize()) { innerPadding ->
                    MainScreen(
                        innerPadding = innerPadding
                    )
                }
            }
        }
    }
}
```



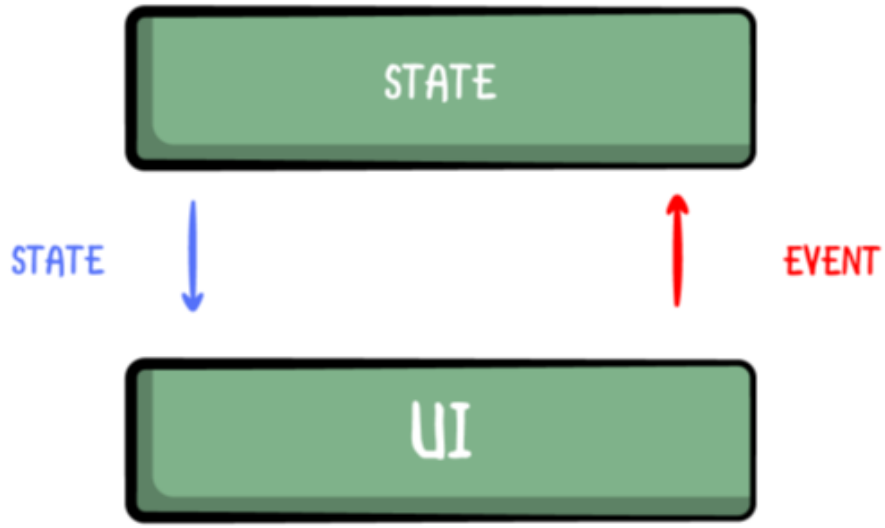
Фундаментальні причини використання ModelView та StateFlow

- **Незалежність логіки отримання даних для інтерфейсу від UI (Jetpack Compose)**. MutableState - це частина бібліотеки Compose Runtime. Якщо використовувати його у ViewModel, логіка отримання даних стає жорстко прив'язаною до Jetpack Compose. StateFlow - це частина стандартної бібліотеки Kotlin (Coroutines/Flow), він нічого не знає про UI.
- **Чисте тестування (Unit Testing)** - тестувати StateFlow у звичайних Unit-тестах значно простіше, оскільки для цього не потрібно запускати середовище виконання Compose.
- **Реактивна трансформація та комбінування потоків** - оскільки StateFlow це потік (об’єктів станів), доступна величезна кількість вбудованих операторів Kotlin Coroutines (map, filter, combine, debounce, flatMapLatest).
- **Архітектурне розділення за принципом UDF (Unidirectional Data Flow)** - коли стан лежав у Composable через remember { mutableStateOf() }, віджет сам керував своїм життям.
  Перемістивши дані у ViewModel, ми розділяємо обов'язки:
  - ViewModel володіє станом, опрацьовує його у корутинах і «транслює» через StateFlow.
  - Composable UI стає stateless - він просто підписується на цей потік за допомогою collectAsStateWithLifecycle(), малює інтерфейс і передає дії користувача (кліки) назад у ViewModel у вигляді подій.

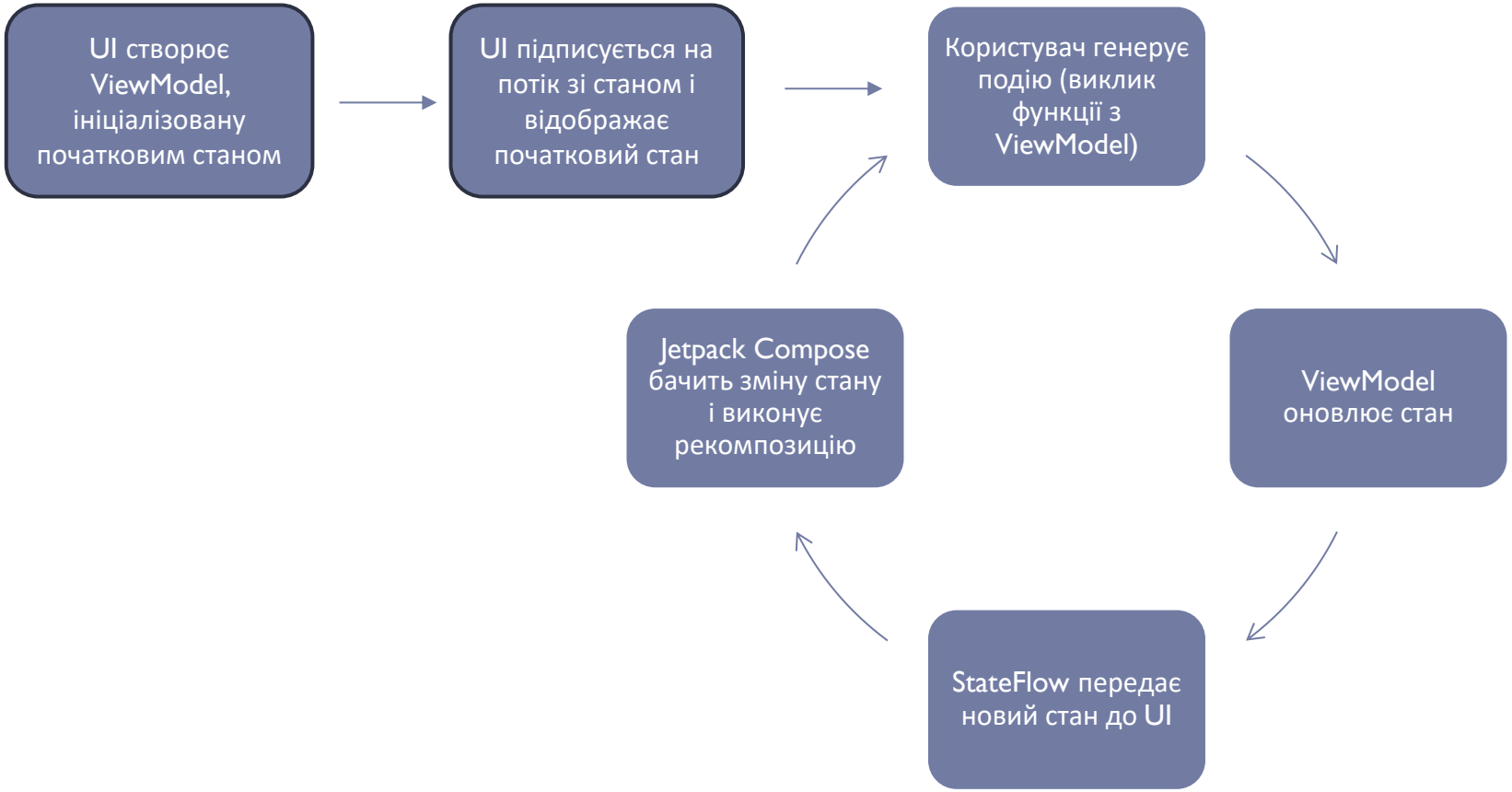


Архітектурний підхід «Односпрямований потік даних» (Unidirectional Data Flow (UDF))

- **Односпрямований потік даних (Unidirectional Data Flow (UDF))** - це архітектурний підхід, де дані рухаються строго в одному напрямку, утворюючи замкнене, але передбачуване коло.
- У системі UDF є два ключові поняття:
  - Стан (State): Дані, які описують, що зараз відображається на екрані (йде «згори донизу» від ViewModel до UI).
  - Події (Events / Actions): Дії користувача (кліки, введення тексту), які надсилаються «знизу догори» від UI до ViewModel.



Життєвий цикл UDF



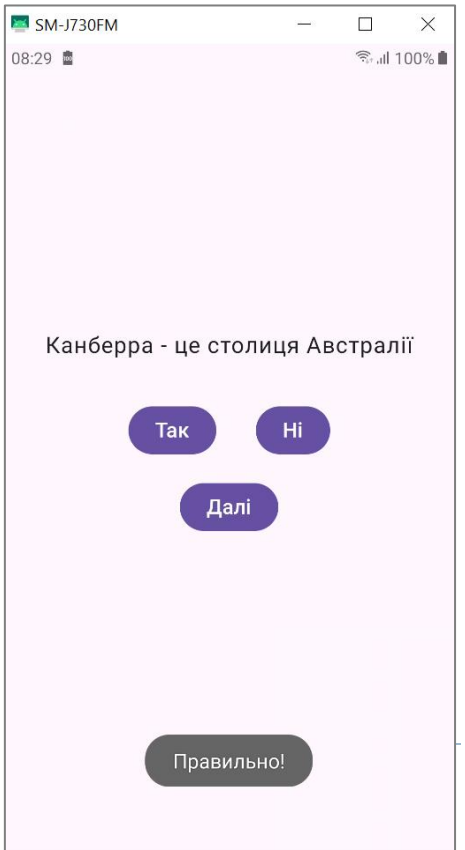
Обробка стану з односпрямованим потоком даних - MainScreenViewModel & MainScreen

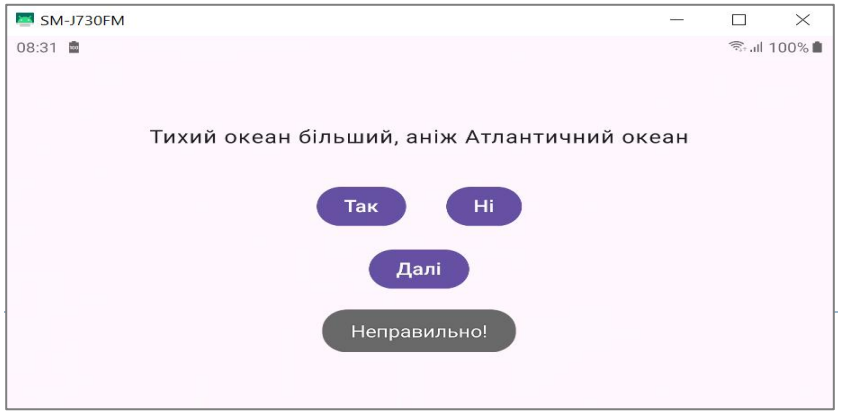


mainScreenState

OnAnswerButtonClick
OnNextQuestionButtonClick







Підняття стану - State Hoisting

- **Підняття стану (State Hoisting)** – це архітектурний шаблон переміщення стану Composable компонента UI до його батьківського компонента (викликаючої сторони – caller). Це робить Composable компонент UI **"без стану" (stateless)**.
- Composable компонент UI **"без стану" (stateless)** не зберігає і не змінює дані, а лише відображає стан, що йому передав батьківський компонент, як параметр, та повідомляє батьківській компонент про події (**Розділення відповідальності (Separation of Concerns**).
- ViewModel залишається власником стану, надаючи стан Composable компонентам UI та обробляючи логіку та події, пов'язані з інтерфейсом користувача.



Підняття стану - State Hoisting

- MainScreenViewModel залишилася без змін.
- 3 MainScreen Composable видалений стан - stateless Composable.

```
@Composable
fun MainScreen(
    mainScreenState: MainScreenState,
    onAnswerButtonClick: (Boolean) -> Int,
    onNextQuestionButtonClick: () -> Unit,
    innerPadding: PaddingValues,
) {
//    val viewModel: MainScreenViewModel = viewModel()
//    val mainScreenState by viewModel.mainScreenState.collectAsStateWithLifecycle()

    val context = LocalContext.current

    Column(
        modifier = Modifier
            .fillMaxSize()
            .padding(innerPadding),
        verticalArrangement = Arrangement.Center,
        horizontalAlignment = Alignment.CenterHorizontally
    ) { ...
```

Тепер стан приймається як параметр, що буде передаватися викликаючою стороною

Callback функції, будуть задані викликаючою стороною

Підняття стану - State Hoisting

```
Text(
    text = stringResource(id = mainScreenState.currentQuestionId),
    textAlign = TextAlign.Center,
    modifier = Modifier.padding(innerPadding)
)
Row(
    modifier = Modifier.size(200.dp, 90.dp),
    horizontalArrangement = Arrangement.SpaceEvenly,
    verticalAlignment = Alignment.CenterVertically
) {
    Button(
        onClick = {
            val messageId = onAnswerButtonClick(true)
            Toast.makeText(
                context,
                messageId,
                Toast.LENGTH_SHORT
            ).show()
        }) {
        Text(text = stringResource(id = R.string.true_button))
    } ...
```

Отримання ідентифікатора відповіді через callback-функцію

Підняття стану - State Hoisting

```
...
    Spacer(modifier = Modifier.size(16.dp))
    Button(
        onClick = {
            val messageId = onAnswerButtonClick(false)
            Toast.makeText(
                context,
                messageId,
                Toast.LENGTH_SHORT
            ).show()
        }) {
        Text(text = stringResource(id = R.string.false_button))
    }
}
Spacer(modifier = Modifier.size(16.dp))
Button(
    onClick = onNextQuestionButtonClick
) {
    Text(text = stringResource(id = R.string.next_button))
}
}
} ...
```

Отримання ідентифікатора відповіді через callback-функцію

Виклик callback-функції

Підняття стану - State Hoisting

```
...

@Preview(showBackground = true)
@Composable
fun MainScreenPreview() {
    MainScreen(
        mainScreenState = MainScreenState(),
        onAnswerButtonClick = {
            R.string.correct_toast
        },
        onNextQuestionButtonClick = {},
        innerPadding = PaddingValues(16.dp)
    )
}
```



Підняття стану - State Hoisting

```
class MainActivity : ComponentActivity() {
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        enableEdgeToEdge()
        setContent {
            GeoQuizComposeDemoTheme {
                Scaffold(modifier = Modifier.fillMaxSize()) { innerPadding ->
                    val viewModel: MainScreenViewModel = viewModel()
                    val mainScreenState by viewModel.mainScreenState.collectAsStateWithLifecycle()
                    MainScreen(
                        mainScreenState = mainScreenState,
                        onAnswerButtonClick = { isCorrect ->
                            viewModel.onAnswerButtonClick(isCorrect)
                        },
                        onNextQuestionButtonClick = {
                            viewModel.onNextButtonClick()
                        },
                        innerPadding = innerPadding
                    )
        }       }       }   } }   ...
```

ВИКЛИКАЮЧА СТОРОНА

створення стану, що зберігається у ViewModel та підписка на його зміни

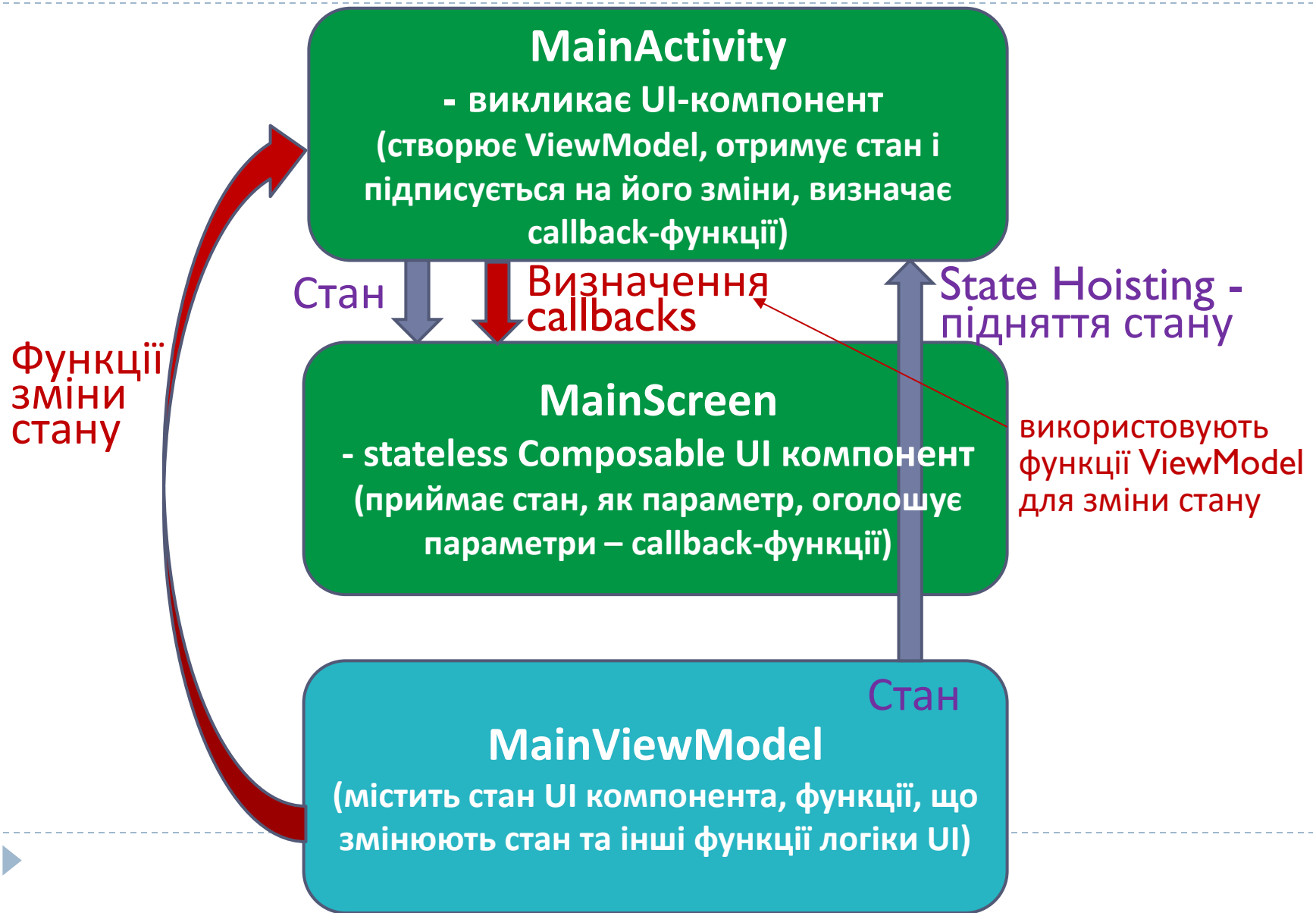
передача стану і визначень callback-функцій до Composable

Підняття стану - State Hoisting

```
...
@Preview(showBackground = true)
@Composable
fun GreetingPreview() {
    GeoQuizComposeDemoTheme {
        MainScreen(
            mainScreenState = MainScreenState(),
            onAnswerButtonClick = { R.string.correct_toast },
            onNextQuestionButtonClick = {},
            innerPadding = PaddingValues())
    }
}
```



Підняття стану - State Hoisting



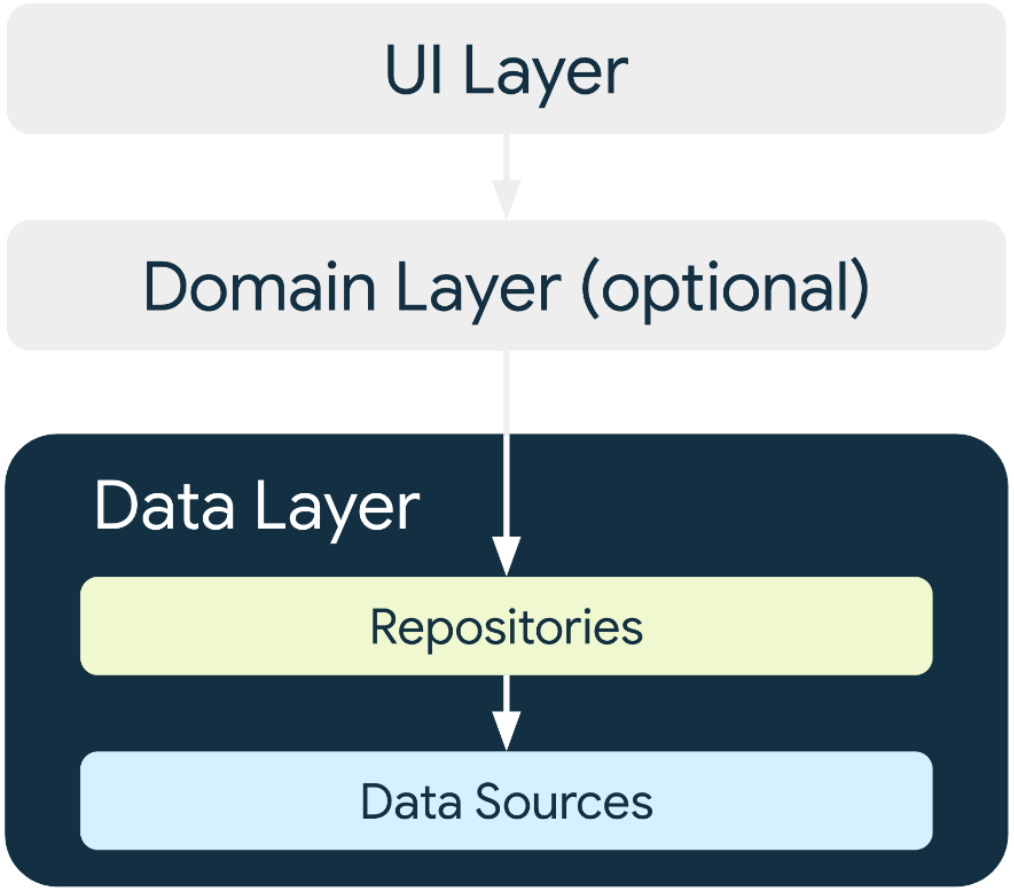
Підняття стану - State Hoisting

- Піднімати стан потрібно **до найближчого спільного батька (Common Parent)** усіх дочірніх компонентів, які читають цей стан або змінюють його.
- Якщо ж стан потрібен лише для візуального ефекту всередині одного компонента (наприклад, чи розгорнута анімація натискання), його можна залишити всередині.
- Composable компонент UI "без стану" легше тестується та легше повторно використовується у різних контекстах.



Додання Data Layer

- Рекомендована архітектура рівня даних у додатку Jetpack Compose, згідно з офіційними рекомендаціями Android, зосереджена навколо **репозиторіїв** та **джерел даних**.

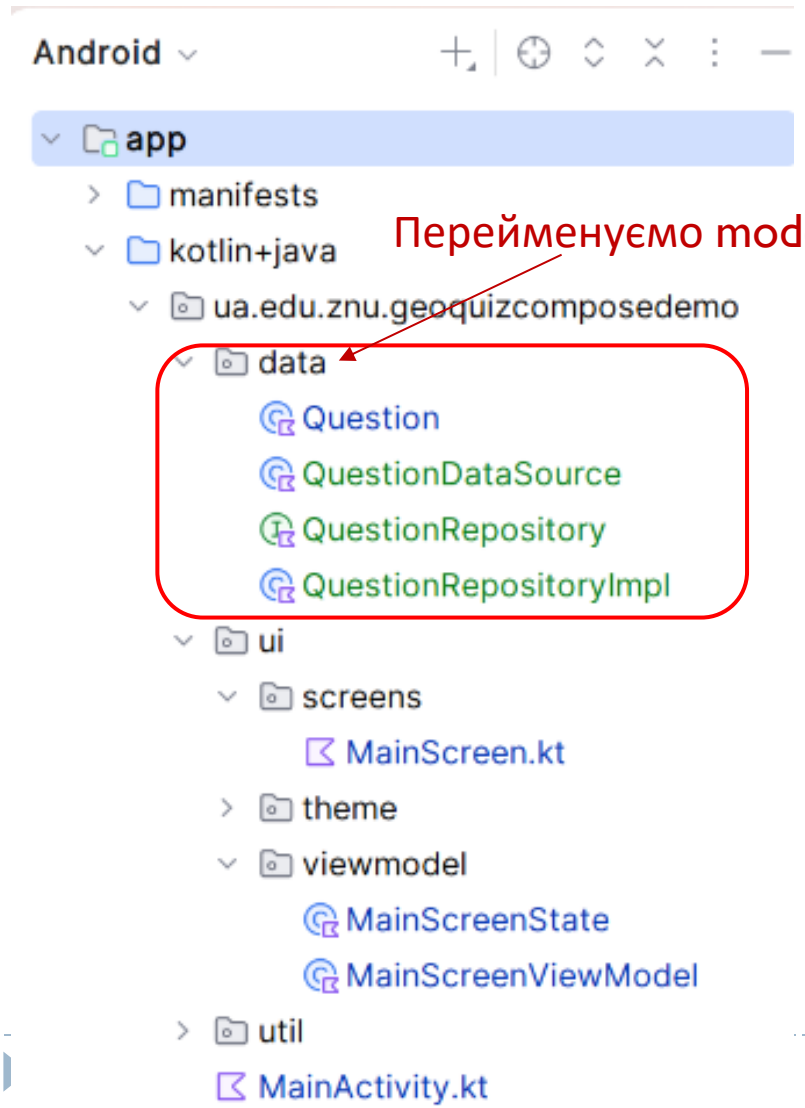


Додання Data Layer

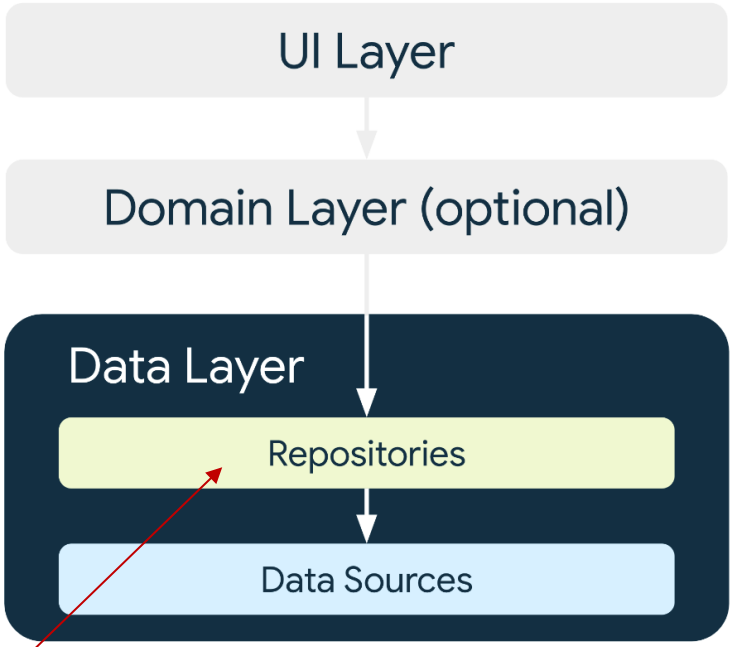
- **Репозиторії -** виступають єдиним джерелом достовірної інформації для певних типів даних. Вони надають дані рівню інтерфейсу користувача (та додатковому рівню домену), а також абстрагують базові джерела даних. Класи репозиторію створюють для кожного окремого типу даних. Наприклад, UserRepository, ProductRepository, OrderRepository. Репозиторії містять бізнес-логіку, пов'язану з отриманням та маніпулюванням даними. Рекомендується організовувати репозиторій як інтерфейс з абстрактними методами та його реалізацію.
- **Джерела даних** - обробляють специфіку зберігання та отримання даних з різних джерел, наприклад, локальної БД, результатів запитів до API тощо. Репозиторій може використовувати одне або кілька джерел даних. Повинна існувати можливість зміни джерела даних без змін у репозиторії.



Додання Data Layer



Перейменуємо model на data



Репозиторій реалізований як інтерфейс з абстрактними методами та клас-реалізація

Додання Data Layer - рефакторинг data-класу

```
package ua.edu.znu.geoquizcomposedemo.data

import java.util.UUID

data class Question(
    val id: String = UUID.randomUUID().toString(),
    val questionText: String,
    val answer: Boolean
)
```

Доданий ідентифікатор з унікальним випадковим значенням, який дозволяє шукати конкретний екземпляр при його оновленні.

Ідентифікатор рядкового ресурсу замінений на рядок для можливості додання нових та редагування старих питань



Додання Data Layer - DataSource

```
package ua.edu.znu.geoquizcomposedemo.data

class QuestionDataSource {

    private val questions = mutableListOf(
        Question(questionText="Канберра - це столиця Австралії", answer = true),
        Question(questionText="Тихий океан більший, аніж Атлантичний океан", answer =
true),
        Question(questionText="Суецький канал поєднує Червоне море та Індійський океан",
answer = false),
        Question(questionText="Витік річки Ніл знаходиться в Єгипті", answer = false),
        Question(questionText="Амазонка є найдовшою річкою у Америці", answer = true),
        Question(questionText="Озеро Байкал є найстарішим та найглибшим прісноводним
озером", answer = true),
    )

    fun getQuestions(): List<Question> = questions
}
```

Тепер список мутабельний для можливості додання нових та редагуванняі видалення старих питань

функція отримання даних

QuestionDataSource - постачальник початкового списку питань. Усе керування станом та оновлення знаходяться в QuestionRepository, що забезпечує єдине джерело достовірної інформації (a single source of truth) для інтерфейсу користувача.

Додання Data Layer - інтерфейс Repository

```
package ua.edu.znu.geoquizcomposedemo.data

interface QuestionRepository {
    fun getQuestions(): List<Question>
    fun getQuestionById(id: String): Question?
}
```





Додання Data Layer - реалізація RepositoryImpl

```
package ua.edu.znu.geoquizcomposedemo.data

import ...

class QuestionRepositoryImpl (val dataSource:QuestionDataSource):QuestionRepository {
    private fun getQuestions(): List<Question> {
        return dataSource.getQuestions()
    }
    override fun getQuestionById (id: String): Question? {
        return dataSource.getQuestions().find { it.id == id }
    }
    override fun getQuestionBankSize() = getQuestions().size
}
```

Приватна функція для зменшення коду у інших функціях

Функції, що використовуватимуться інтерфейсом користувача (екземплярами ViewModel)



Додання Data Layer – MainScreenState refactoring

```
package ua.edu.znu.geoquizcomposedemo.ui.viewmodel

data class MainScreenState(
    val currentIndex: Int = 0,
    val currentQuestion: String = "",
)
```

Тепер стан тримає текст питання



Додання Data Layer - створення фабрики ViewModel

- Оскільки при створенні екземпляра MainScreenViewModel тепер необхідно передавати аргумент QuestionRepository, необхідно використовувати **ViewModelProvider.Factory** (функція viewModel() за замовчуванням працює тільки для ViewModel без аргументів).
- З метою використання для усіх ViewModel застосунку, створимо generic-клас **ViewModelFactory** з аргументом - класом ViewModel

```
package ua.edu.znu.geoquizcomposedemo.ui..viewmodel

import ...

class ViewModelFactory<T : ViewModel>(
    private val modelClass: Class<T>,
    private val creator: () -> T
) : ViewModelProvider.Factory {
    override fun <U : ViewModel> create(modelClass: Class<T>): T {
        if (this.modelClass.isAssignableFrom(modelClass)) {
            @Suppress("UNCHECKED_CAST")
            return creator() as T
        }
        throw IllegalArgumentException("Unknown ViewModel class: ${modelClass.name}")
    }}
```

Додання Data Layer - MainViewModel refactoring

```
class MainScreenViewModel(val QuestionRepository: QuestionRepository) : ViewModel() {
    private val _mainScreenState = MutableStateFlow(
        MainScreenState(
            currentQuestion = questionRepository.getQuestions().first().questionText
        )
    )
    val mainScreenState = _mainScreenState.asStateFlow()
    fun onNextButtonClick() {
        _mainScreenState.update { currentState ->
            val questionBank = questionRepository.getQuestions()
            val nextIndex = (currentState.currentIndex + 1) % questionBank.size
            currentState.copy(
                currentIndex = nextIndex,
                currentQuestion = questionBank[nextIndex].questionText
            )
        }
    }
...
```

Репозиторій передається як параметр, як тип зазначається інтерфейс



Додання Data Layer - MainViewModel refactoring

```
...
    fun onAnswerButtonClick(isTrue: Boolean): Int {
        val questionBank = questionRepository.getQuestions()
        val correctAnswer = questionBank[mainScreenState.value.currentIndex].answer
        return if (isTrue == correctAnswer) {
            R.string.correct_toast
        } else {
            R.string.incorrect_toast
        }
    }
}
```



Додання Data Layer - MainScreen refactoring

```
@Composable
fun MainScreen(
    mainScreenState: MainScreenState,
    onAnswerButtonClick: (Boolean) -> Int,
    onNextQuestionButtonClick: () -> Unit,
    innerPadding: PaddingValues,
) {
    val context = LocalContext.current
    Column(
        modifier = Modifier
            .fillMaxSize()
            .padding(innerPadding),
        verticalArrangement = Arrangement.Center,
        horizontalAlignment = Alignment.CenterHorizontally
    ) {
        Text(
            text = mainScreenState.currentQuestion,
            textAlign = TextAlign.Center,
            modifier = Modifier.padding(innerPadding)
        ) ...
```

Тепер зі стану отримується текст питання

Додання Data Layer - MainScreen refactoring

```
...
    Row(
        modifier = Modifier.size(200.dp, 90.dp),
        horizontalArrangement = Arrangement.SpaceEvenly,
        verticalAlignment = Alignment.CenterVertically
    ) {
        Button(
            onClick = {
                val messageId = onAnswerButtonClick(true)
                Toast.makeText(
                    context,
                    messageId,
                    Toast.LENGTH_SHORT
                ).show()
            }) {
            Text(text = stringResource(id = R.string.true_button))
        }
        Spacer(modifier = Modifier.size(16.dp))
...
```



Додання Data Layer - MainScreen refactoring

```
...
        Button(
            onClick = {
                val messageId = onAnswerButtonClick(false)
                Toast.makeText(
                    context,
                    messageId,
                    Toast.LENGTH_SHORT
                ).show()
            }) {
            Text(text = stringResource(id = R.string.false_button))
        }
    }
    Spacer(modifier = Modifier.size(16.dp))
    Button(
        onClick = onNextQuestionButtonClick
    ) {
        Text(text = stringResource(id = R.string.next_button))
    }
  }
} ...
```

Додання Data Layer - MainScreen refactoring

```
...

@Preview(showBackground = true)
@Composable
fun MainScreenPreview() {
    MainScreen(
        mainScreenState = MainScreenState(),
        onAnswerButtonClick = {
            R.string.correct_toast
        },
        onNextQuestionButtonClick = {},
        innerPadding = PaddingValues(16.dp)
    )
}
```



Додання Data Layer - MainActivity refactoring

ВИКЛИКАЮЧА СТОРОНА

```
class MainActivity : ComponentActivity() {
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        enableEdgeToEdge()
        setContent {
            GeoQuizComposeDemoTheme {
                Scaffold(modifier = Modifier.fillMaxSize()) { innerPadding ->
//                  val viewModel: MainScreenViewModel = viewModel()
                    val questionDataSource = QuestionDataSource()
                    val questionRepository = QuestionRepositoryImpl(questionDataSource)
                    val viewModel: MainScreenViewModel = viewModel(
                        factory = ViewModelFactory(MainScreenViewModel::class.java) {
                            MainScreenViewModel(questionRepository)
                        }
                    )
                    val mainScreenState by viewModel.mainScreenState.collectAsStateWithLifecycle()
...
```

Створення джерела даних та репозиторію

Для ViewModel з параметром використовується ViewModelProvider.Factory

створення стану, що зберігається у ViewModel та підписка на його зміни

Додання Data Layer - MainActivity refactoring

```
...
                MainScreen(
                    mainScreenState = mainScreenState,
                    onAnswerButtonClick = { isCorrect ->
                        viewModel.onAnswerButtonClick(isCorrect)
                    },
                    onNextQuestionButtonClick = {
                        viewModel.onNextButtonClick()
                    },
                    innerPadding = innerPadding
                )
            }
        }
    }
}
...
```

stateless Composable, що викликається

передача стану і визначень callback-функцій до Composable

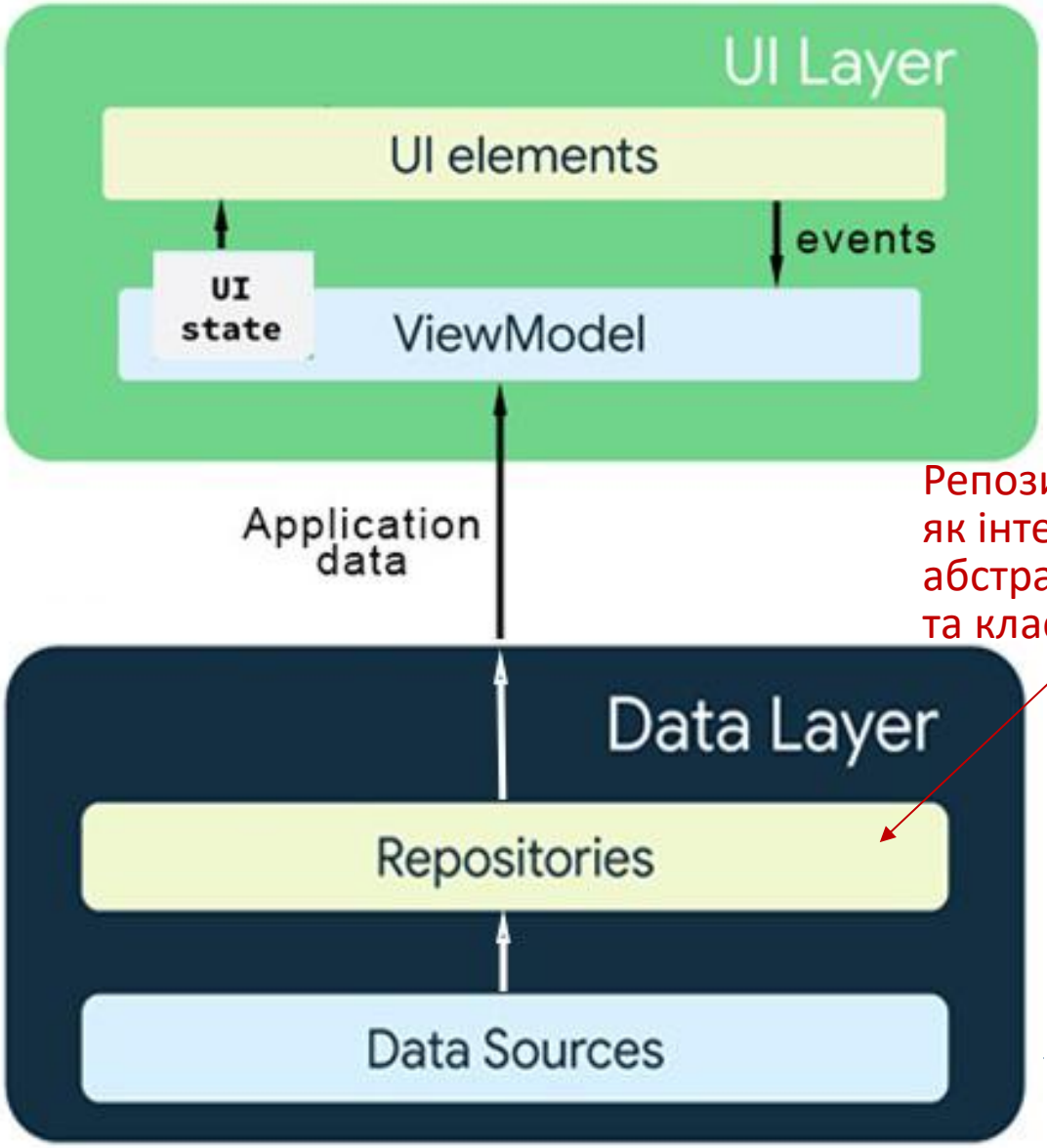


Додання Data Layer - MainActivity refactoring

```
...
@Preview(showBackground = true)
@Composable
fun GreetingPreview() {
    GeoQuizComposeDemoTheme {
        MainScreen(
            mainScreenState = MainScreenState(),
            onAnswerButtonClick = { R.string.correct_toast },
            onNextQuestionButtonClick = {},
            innerPadding = PaddingValues()
        )
    }
}
```



Архітектура Jetpack Compose застосунку



Репозиторій реалізований як інтерфейс з абстрактними методами та клас-реалізація