twitch.tv/dancojocar
youtube.com/dancojocar

Lecture #12
Architecture Guide Update & Multiplatform Mobile

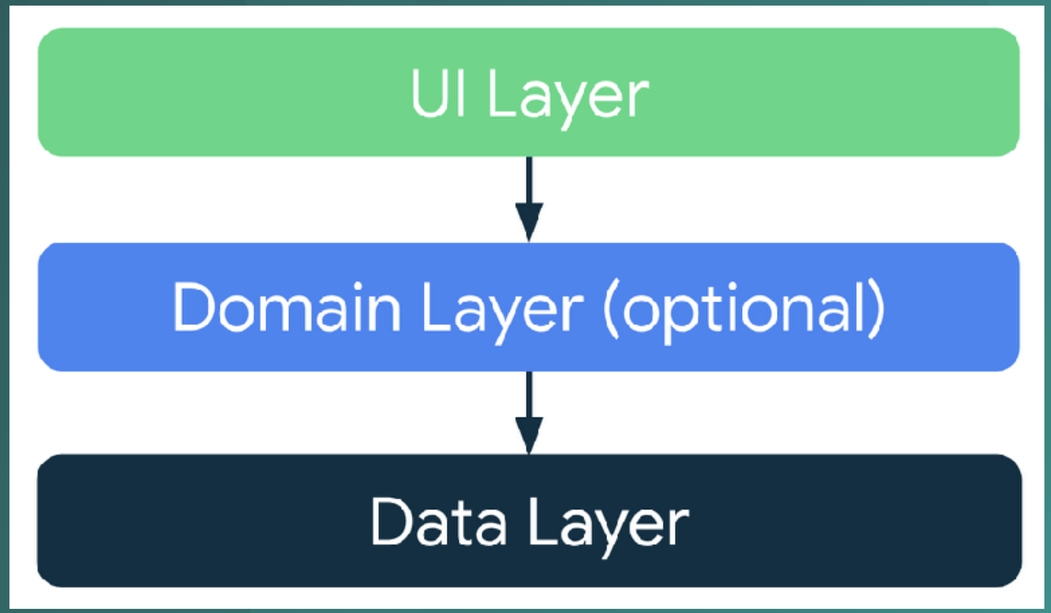
Mobile Applications
Fall 2024

App Architecture

- Scale
- Quality
- Robustness
- Easier to test

Layers

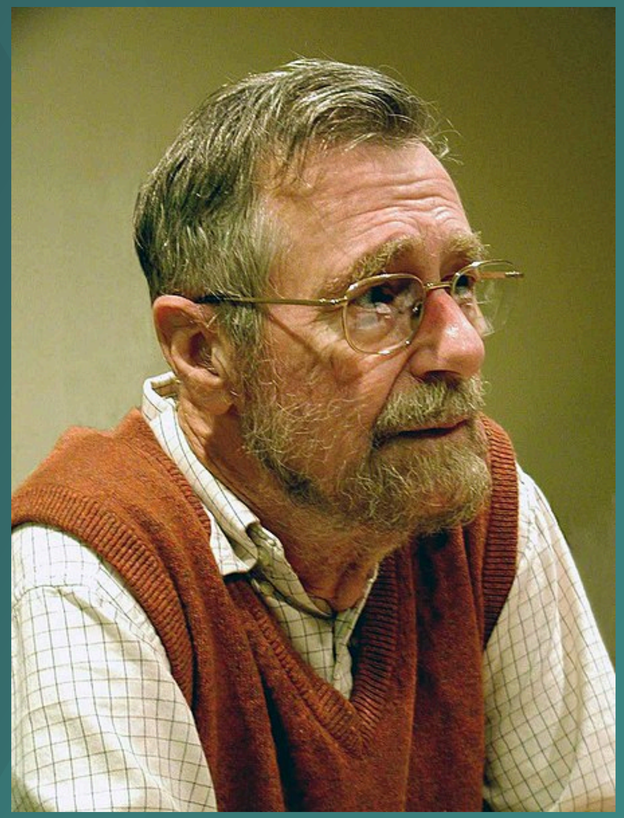
- UI - Display data on the screen.
- Data - Business logic.
- Domain - Reuse interactions.



https://developer.android.com/topic/architecture

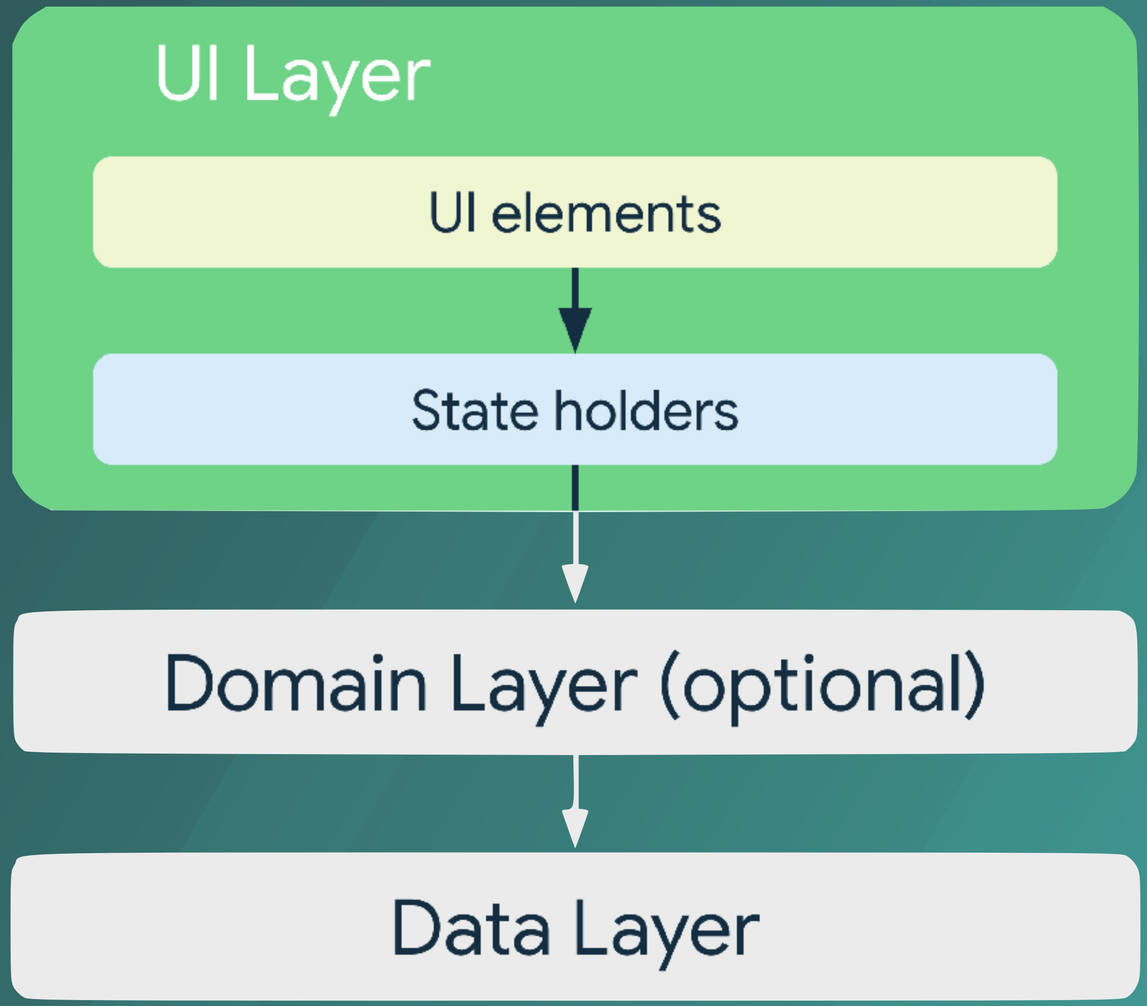
Overview

- Separation of concerns.
- Drive UI from data (persistent) models.
  - Users don’t lose data.
  - App will work even on bad connections or offline.

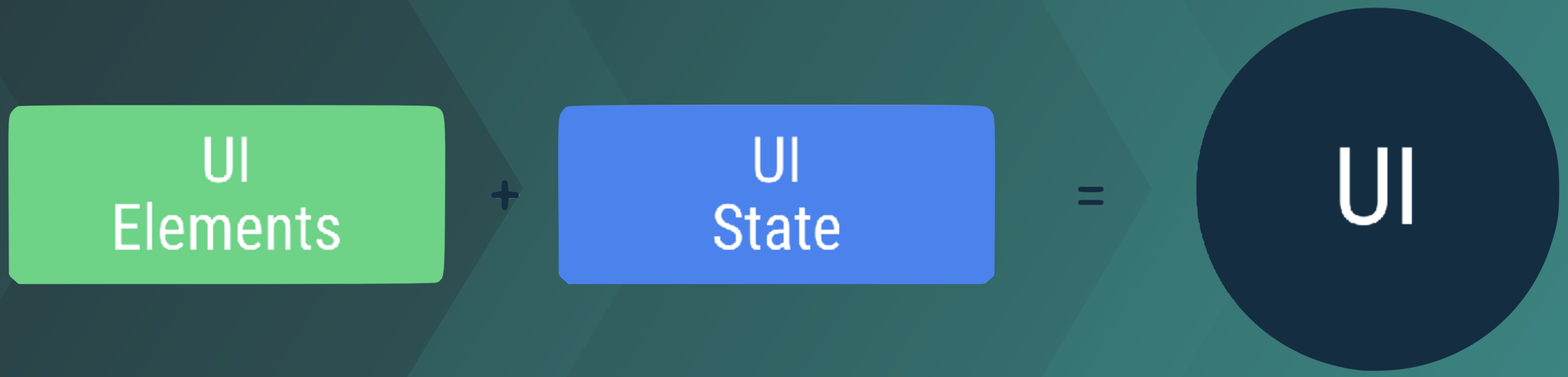


UI Layer

- Ui elements that render the data - Views or Compose functions.
- State holders
  - Hold data.
  - Expose to the UI
  - Handle logic.



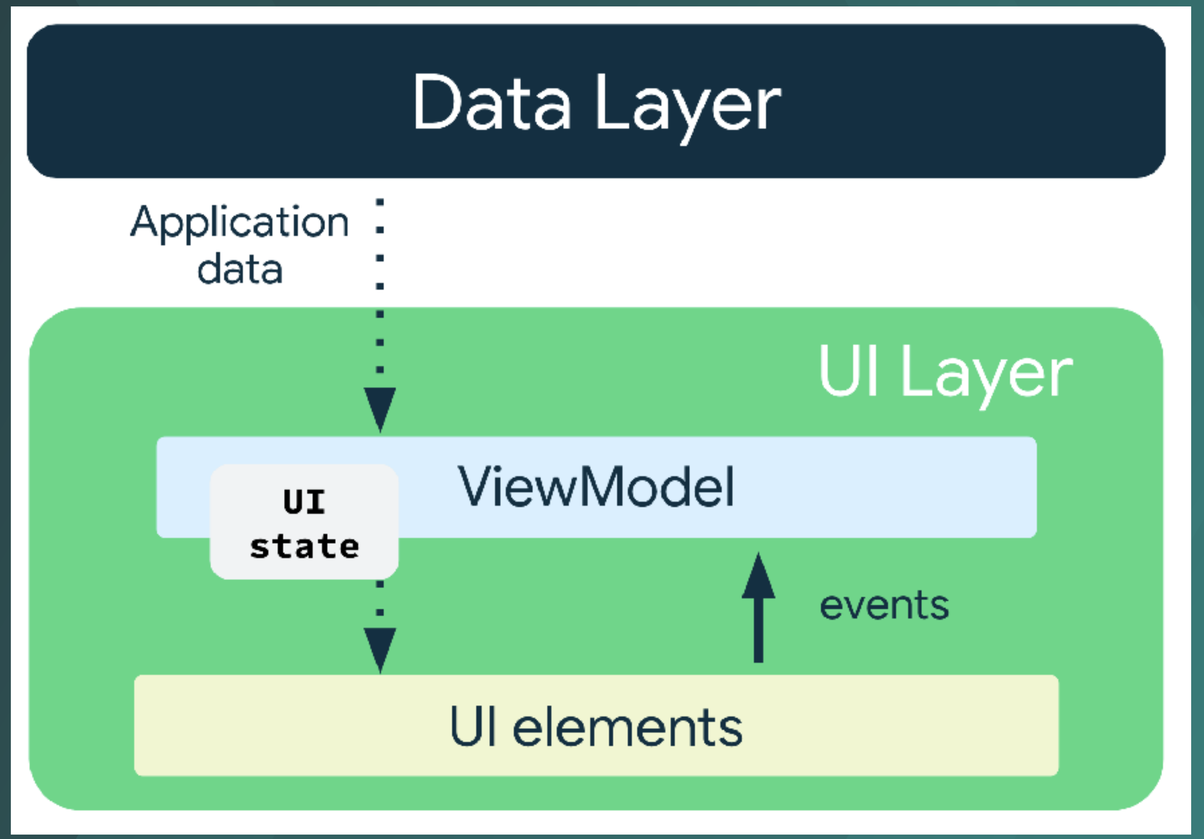
UI Layer



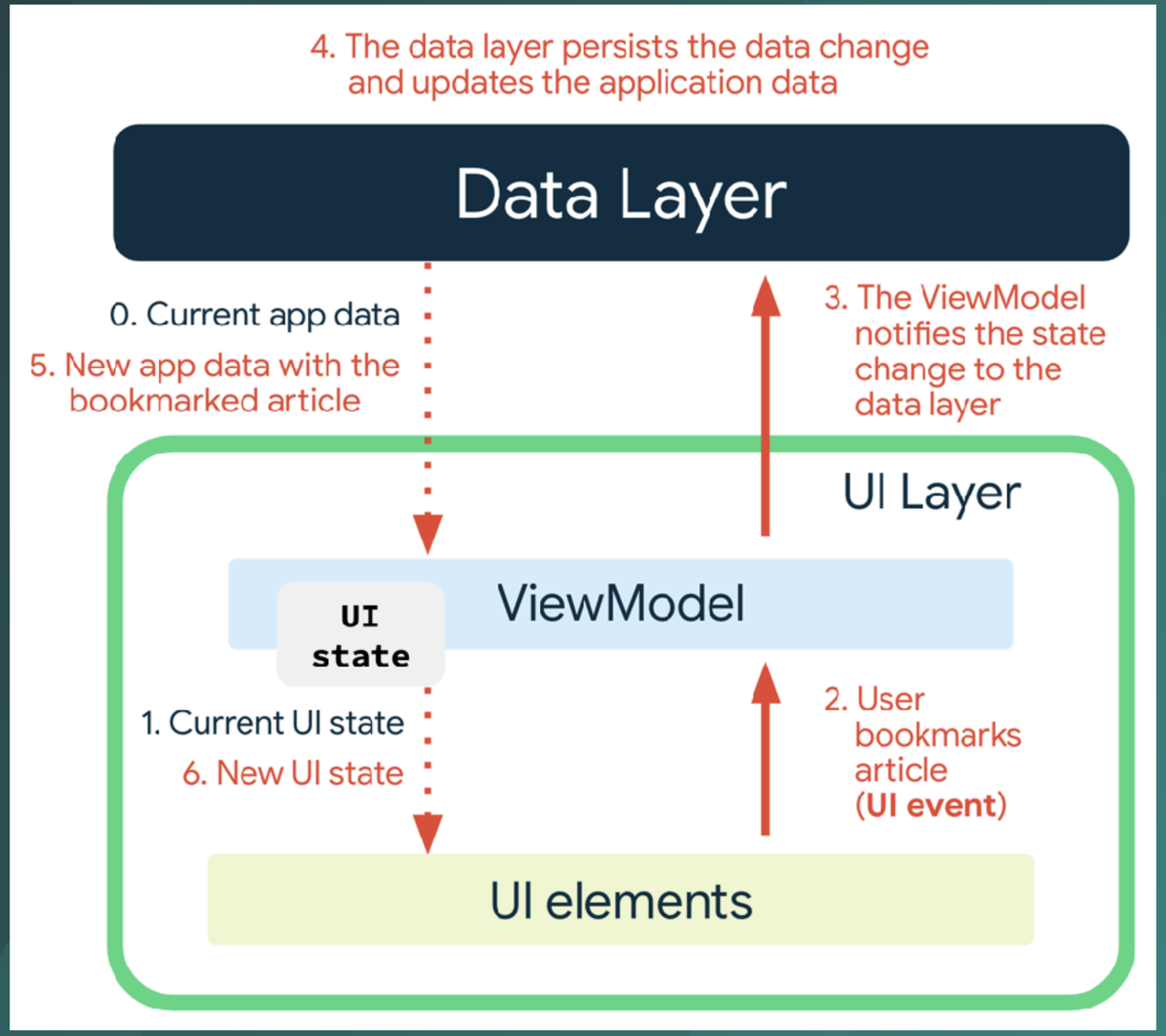
Define UI state

```
data class NewsUiState(
    val isSignedIn: Boolean = false,
    val isPremium: Boolean = false,
    val newsItems: List<NewsItemUiState> = listOf(),
    val userMessages: List<Message> = listOf()
)

data class NewsItemUiState(
    val title: String,
    val body: String,
    val bookmarked: Boolean = false,
    ...
)
```

Define UI state

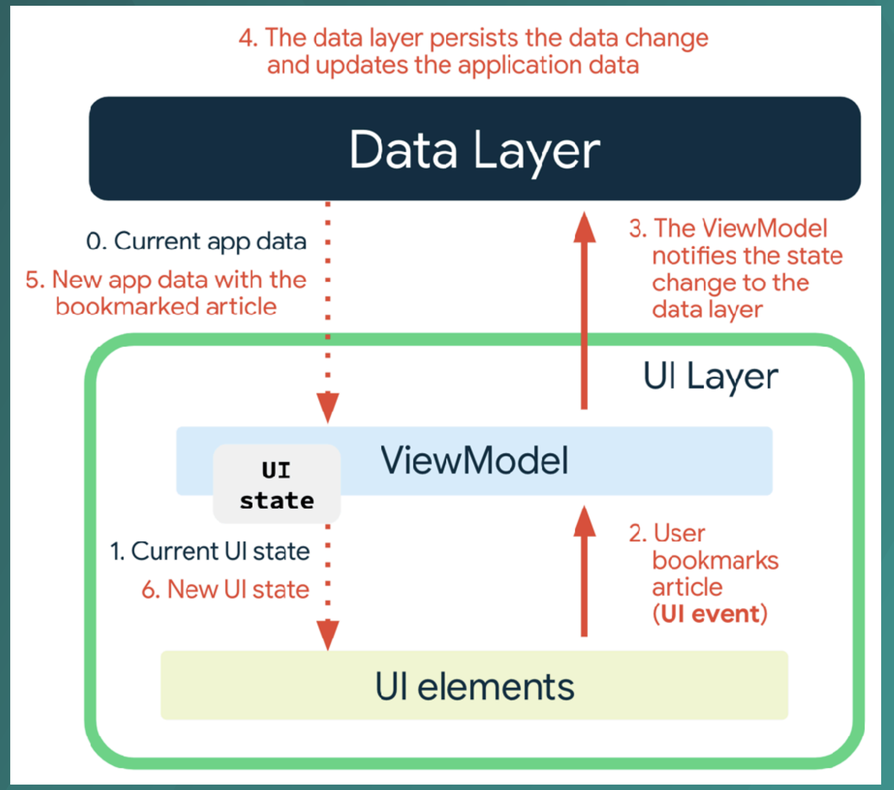


Unidirectional Data Flow



Why use UDF?

- Data consistency - Single source of truth.
- Testability - Isolated state, testable independent of the UI.
- Maintainability - Mutations follow a well-defined pattern.



Expose UI state

```
class NewsViewModel(...) : ViewModel() {

    private val _uiState = MutableStateFlow(NewsUiState())
    val uiState: StateFlow<NewsUiState> = _uiState.asStateFlow()

    ...

}
```

```
class NewsViewModel(...) : ViewModel() {

    var uiState by mutableStateOf(NewsUiState())
        private set

    ...
}
```

Consume UI state

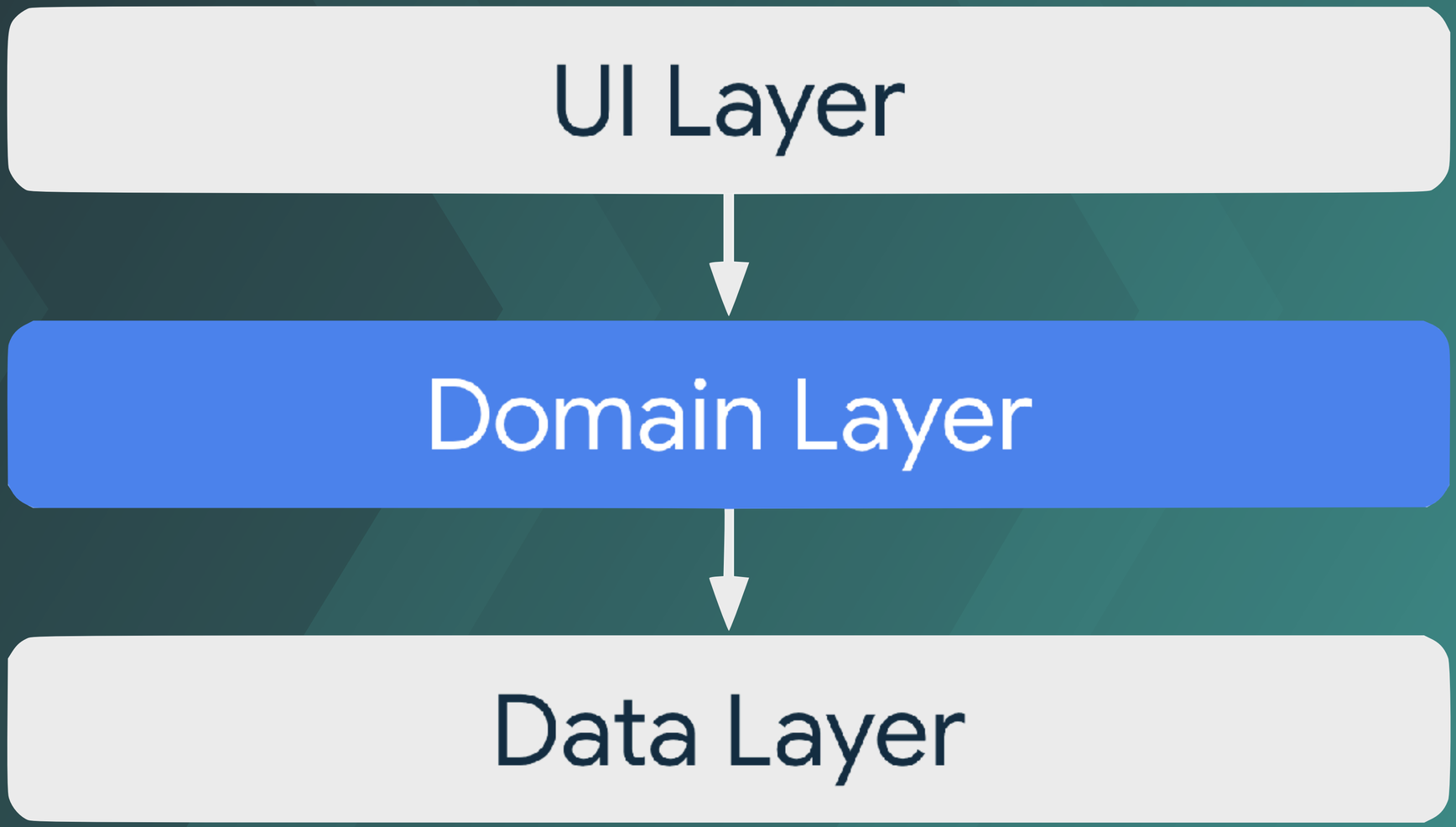
```
class NewsActivity : AppCompatActivity() {

    private val viewModel: NewsViewModel by viewModels()

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

        lifecycleScope.launch {
            repeatOnLifecycle(Lifecycle.State.STARTED) {
                viewModel.uiState.collect {
                    // Update UI elements
                }
            }
        }
    }
}
```

```
@Composable
fun LatestNewsScreen(
    viewModel: NewsViewModel = viewModel()
) {
    // Show UI elements based on the viewModel.uiState
}
```

Show in-progress operations

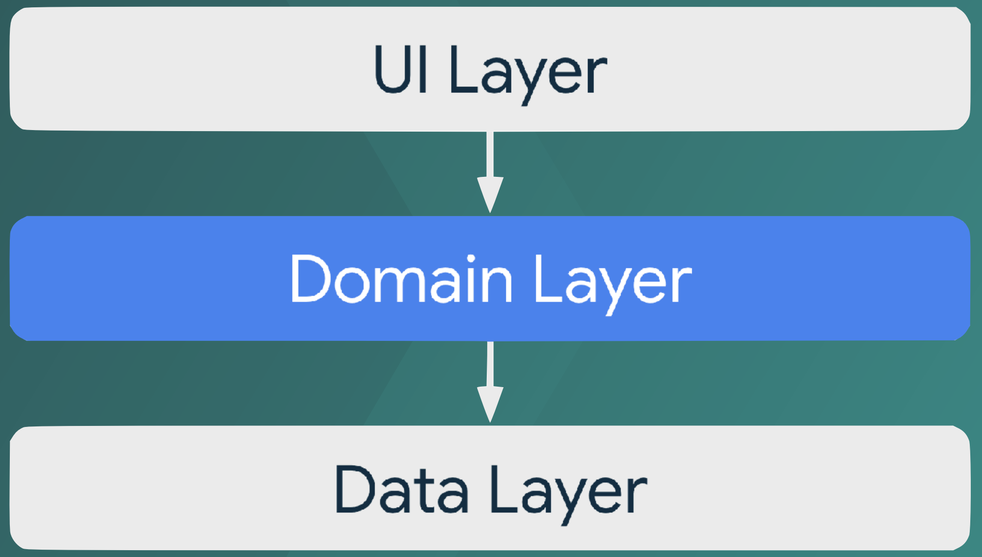
```
data class NewsUiState(
    val isFetchingArticles: Boolean = false,
    ...
)
```

Domain Layer

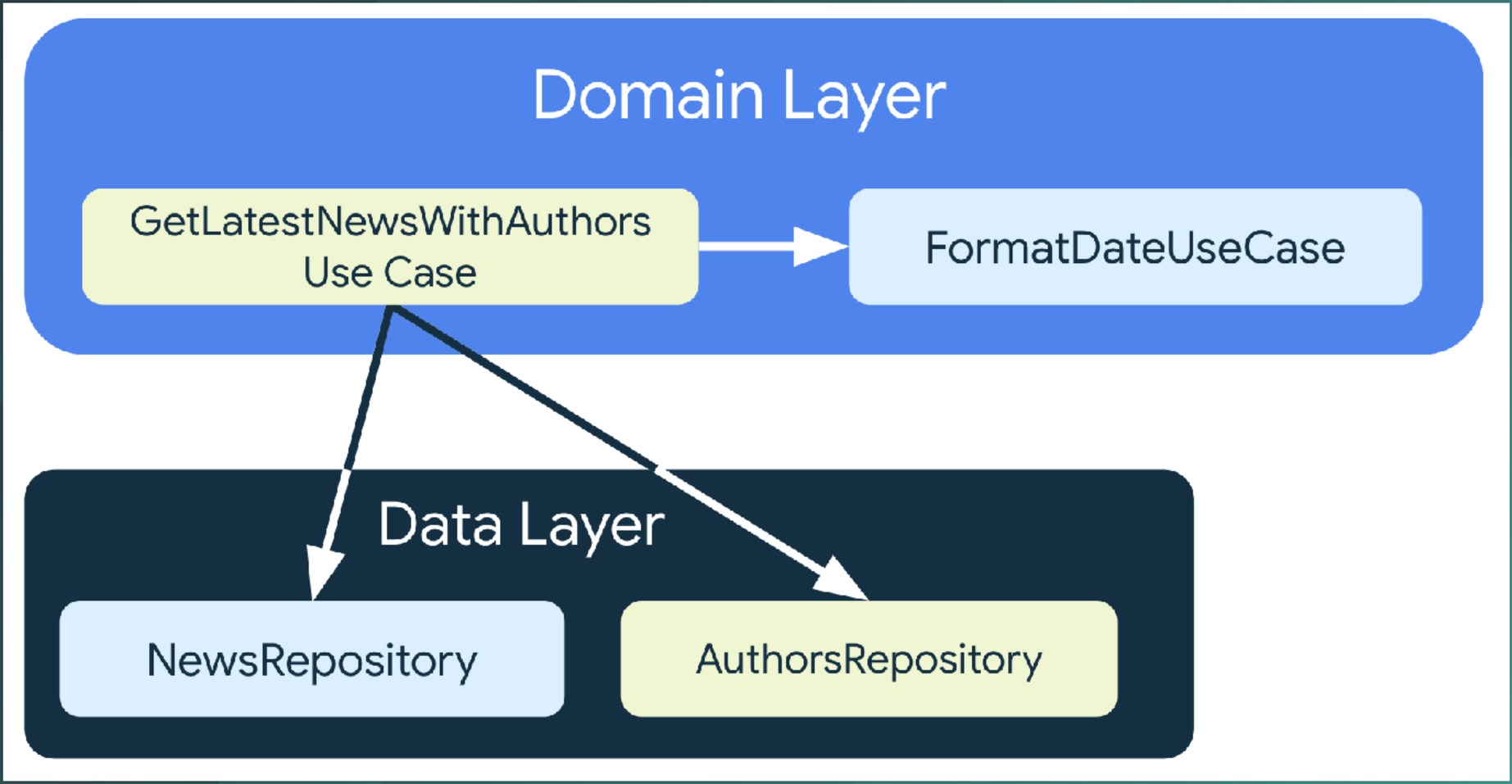


Domain Layer

- Avoids code duplication.
- Improves readability.
- Improves testability.
- Split responsibilities.



Domain Layer



Usage

```
class FormatDateUseCase(userRepository: UserRepository) {

    private val formatter = SimpleDateFormat(
        userRepository.getPreferredDateFormat(),
        userRepository.getPreferredLocale()
    )

    operator fun invoke(date: Date): String {
        return formatter.format(date)
    }
}
```

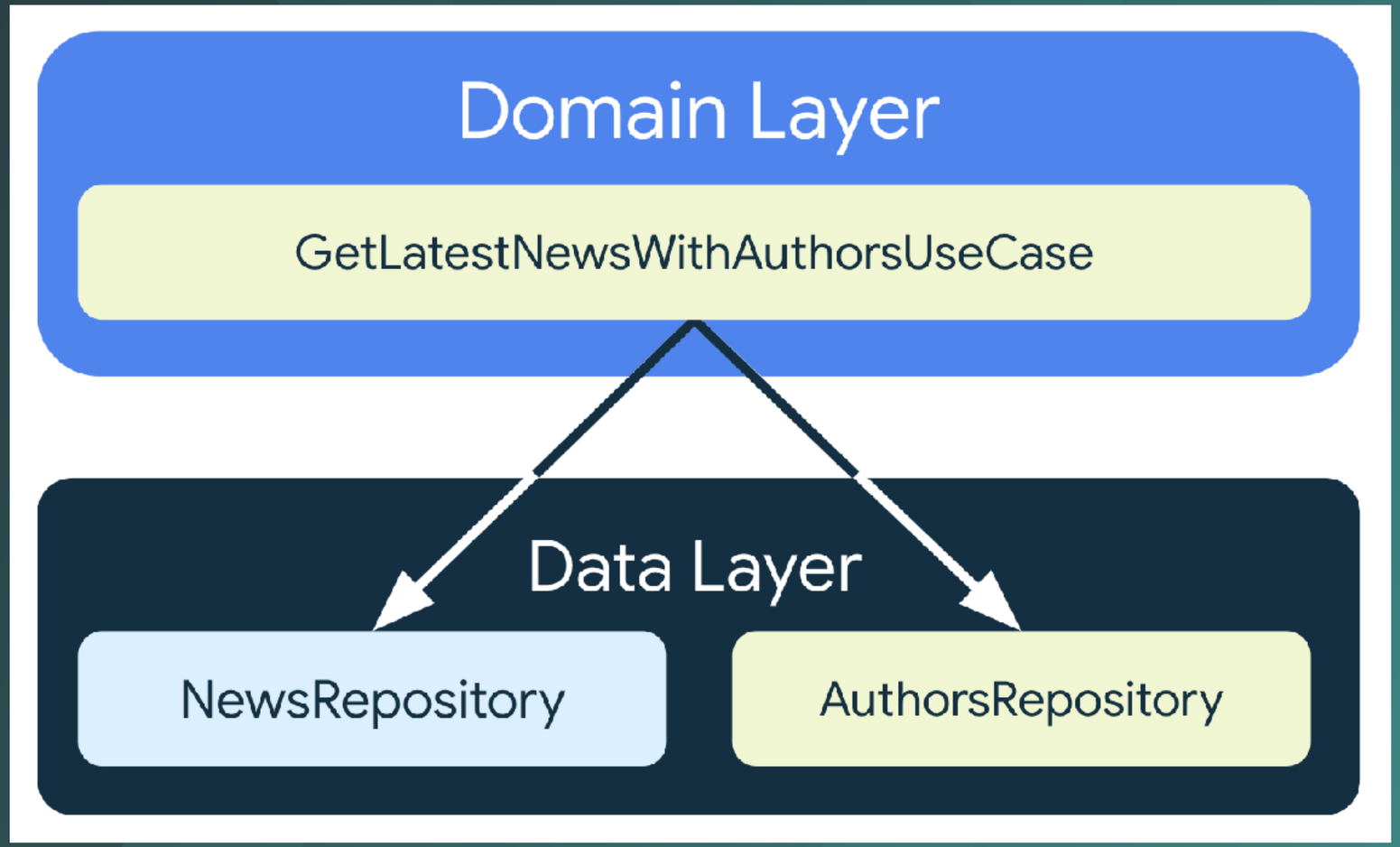
Usage

```
class MyViewModel(formatDateUseCase: FormatDateUseCase) : ViewModel() {
    init {
        val today = Calendar.getInstance()
        val todaysDate = formatDateUseCase(today)
        /* ... */
    }
}
```

Threading

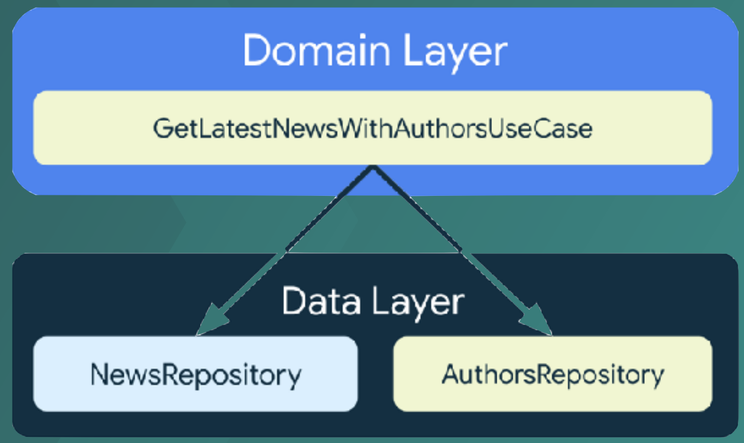
```
class MyUseCase(
    private val defaultDispatcher: CoroutineDispatcher = Dispatchers.Default
) {

    suspend operator fun invoke(...) = withContext(defaultDispatcher) {
        // Long-running blocking operations happen on a background thread.
    }
}
```

Combine repositories

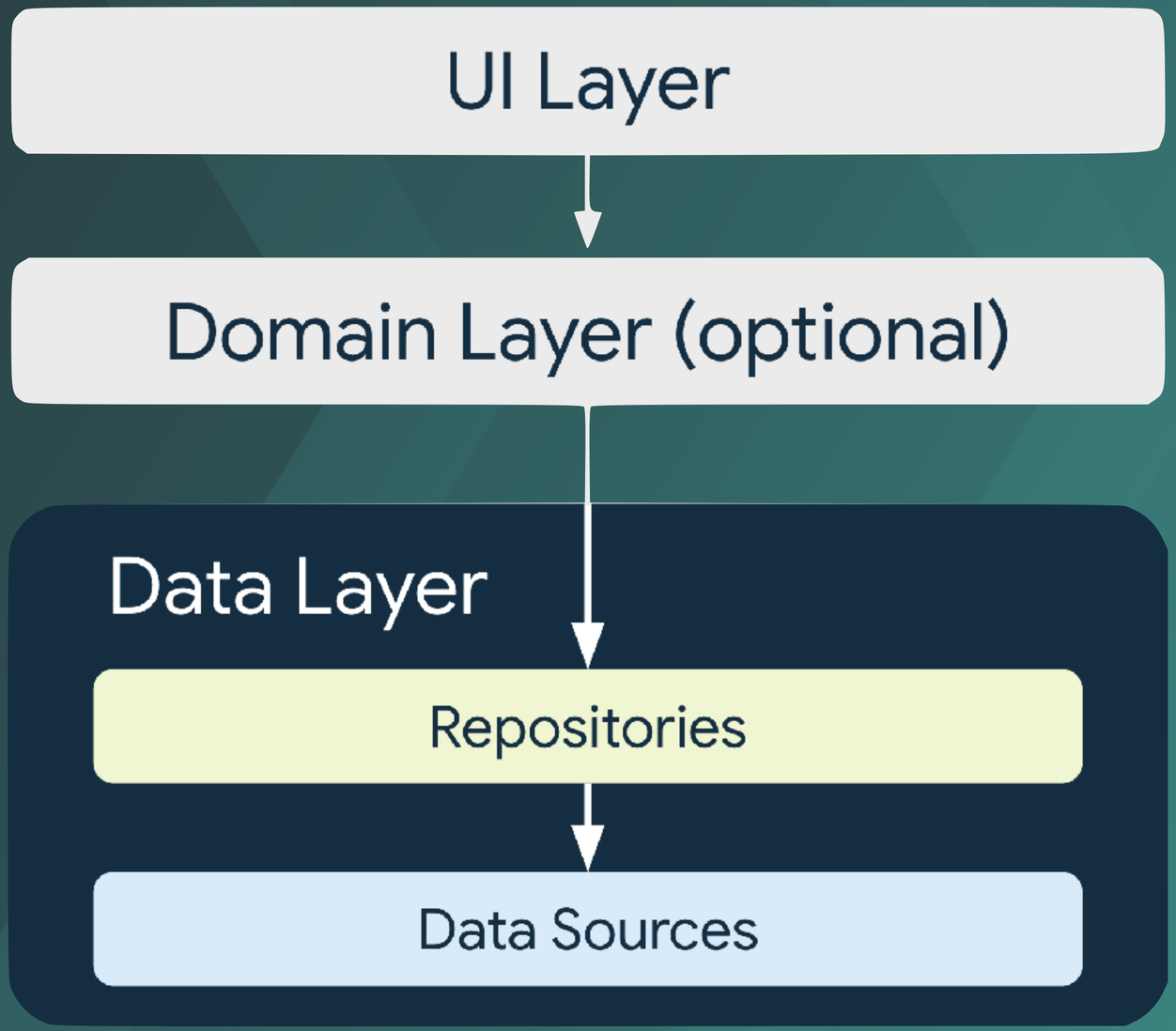


Combine repositories

```
class GetLatestNewsWithAuthorsUseCase(
  private val newsRepository: NewsRepository,
  private val authorsRepository: AuthorsRepository,
  private val defaultDispatcher: CoroutineDispatcher = Dispatchers.Default
) {
    suspend operator fun invoke(): List<ArticleWithAuthor> =
        withContext(defaultDispatcher) {
            val news = newsRepository.fetchLatestNews()
            val result: MutableList<ArticleWithAuthor> = mutableListOf()
            // This is not parallelized, the use case is linearly slow.
            for (article in news) {
                // The repository exposes suspend functions
                val author = authorsRepository.getAuthor(article.authorId)
                result.add(ArticleWithAuthor(article, author))
            }
            result
        }
}
```

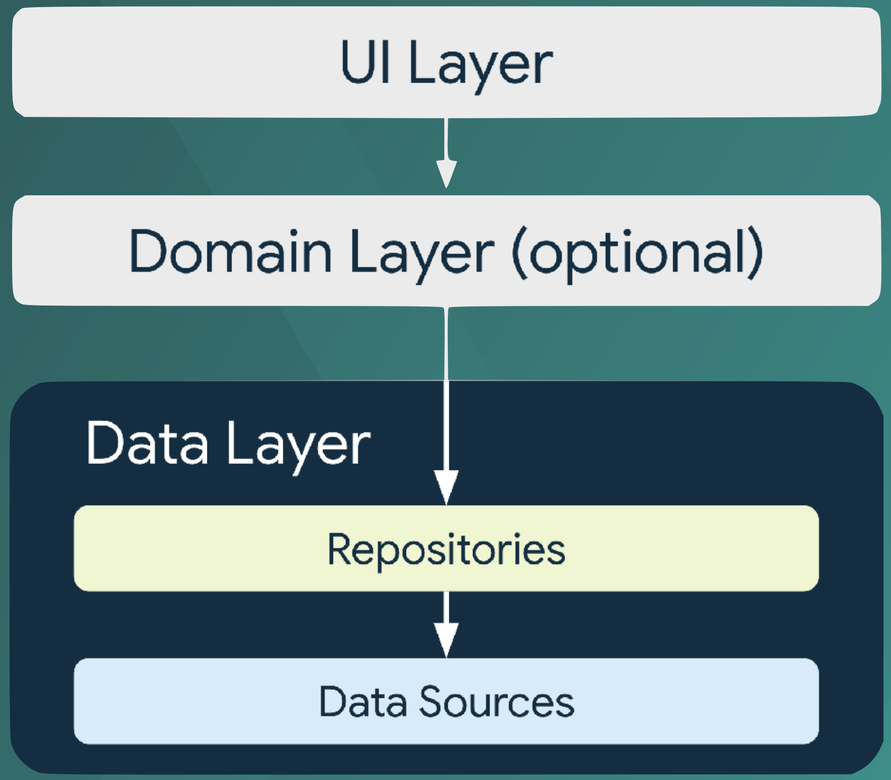


Data Layer


UI Layer
Domain Layer (optional)
Data Layer
Repositories
Data Sources

Data Layer

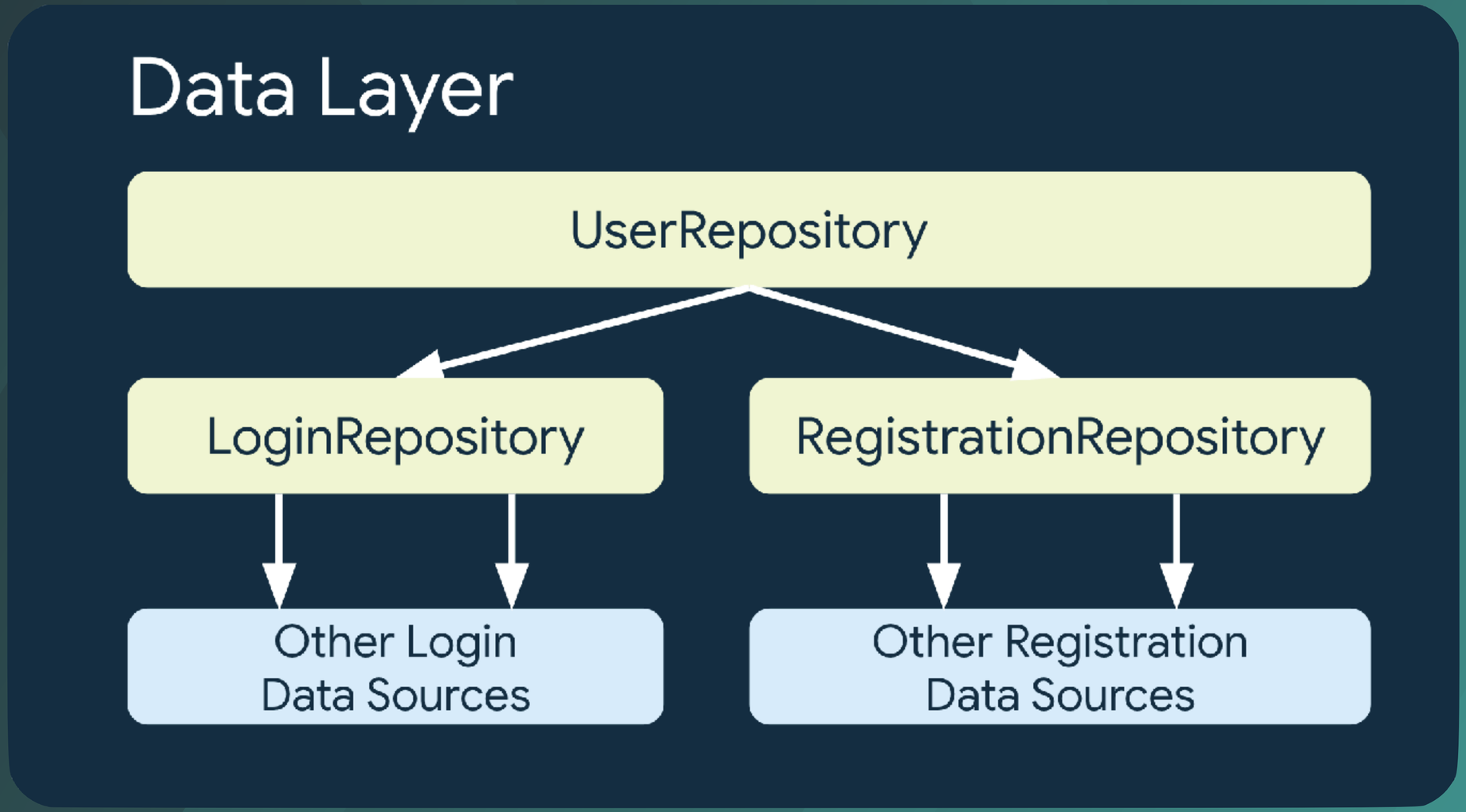
- Exposing data.
- Centralizing data changes.
- Resolving conflicts.
- Abstracting sources of data.
- Combining business logic.



Expose APIs

```
class ExampleRepository(
    private val exampleRemoteDataSource: ExampleRemoteDataSource, // network
    private val exampleLocalDataSource: ExampleLocalDataSource // database
) {

    val data: Flow<Example> = ...

    suspend fun modifyData(example: Example) { ... }
}
```

Multiple levels of repositories



Represent Business Models

```
data class ArticleApiModel(
    val id: Long,
    val title: String,
    val content: String,
    val publicationDate: Date,
    val modifications: Array<ArticleApiModel>,
    val comments: Array<CommentApiModel>,
    val lastModificationDate: Date,
    val authorId: Long,
    val authorName: String,
    val authorDateOfBirth: Date,
    val readTimeMin: Int
),
```

Represent Business Models

- Saves app memory usage.
- Adapts external data types to used data types.
- Provides better separation of concerns.

Create the Repository

```
// NewsRepository is consumed from other layers of the hierarchy.
class NewsRepository(
    private val newsRemoteDataSource: NewsRemoteDataSource
) {
    suspend fun fetchLatestNews(): List<ArticleHeadline> =
        newsRemoteDataSource.fetchLatestNews()
}
```

Cache the Result

```
class NewsRepository(
  private val newsRemoteDataSource: NewsRemoteDataSource
) {
    // Mutex to make writes to cached values thread-safe.
    private val latestNewsMutex = Mutex()
    // Cache of the latest news got from the network.
    private var latestNews: List<ArticleHeadline> = emptyList()
    suspend fun getLatestNews(refresh: Boolean = false): List<ArticleHeadline> {
        if (refresh || latestNews.isEmpty()) {
            val networkResult = newsRemoteDataSource.fetchLatestNews()
            // Thread-safe write to latestNews
            latestNewsMutex.withLock {
                this.latestNews = networkResult
            }
        }
        return latestNewsMutex.withLock { this.latestNews }
    }
}
```

```
class NewsRepository(
    private val newsRemoteDataSource: NewsRemoteDataSource,
    private val externalScope: CoroutineScope
) {
    /* ... */

    suspend fun getLatestNews(refresh: Boolean = false): List<ArticleHeadline> {
        return if (refresh) {
            externalScope.async {
                newsRemoteDataSource.fetchLatestNews().also { networkResult ->
                    // Thread-safe write to latestNews.
                    latestNewsMutex.withLock {
                        latestNews = networkResult
                    }
                }
            }.await()
        } else {
            return latestNewsMutex.withLock { this.latestNews }
        }
    }
}
```

Schedule Tasks

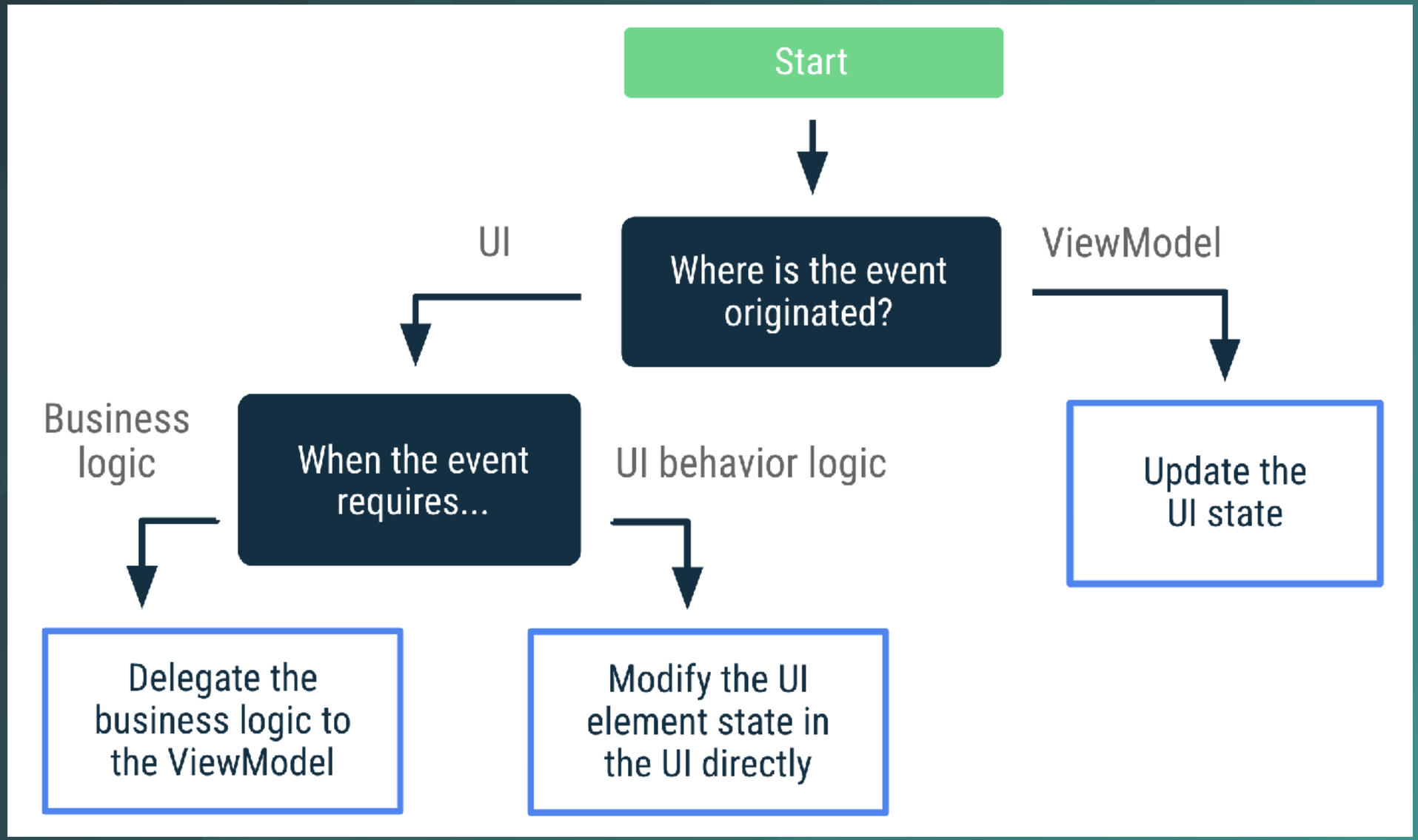
```
class RefreshLatestNewsWorker(
    private val newsRepository: NewsRepository,
    context: Context,
    params: WorkerParameters
) : CoroutineWorker(context, params) {

    override suspend fun doWork(): Result = try {
        newsRepository.refreshLatestNews()
        Result.success()
    } catch (error: Throwable) {
        Result.failure()
    }
}
```

Schedule Tasks

```
private const val REFRESH_RATE_HOURS = 4L
private const val FETCH_LATEST_NEWS_TASK = "FetchLatestNewsTask"
private const val TAG_FETCH_LATEST_NEWS = "FetchLatestNewsTaskTag"

class NewsTasksDataSource(
    private val workManager: WorkManager
) {
    fun fetchNewsPeriodically() {
        val fetchNewsRequest = PeriodicWorkRequestBuilder<RefreshLatestNewsWorker>(
            REFRESH_RATE_HOURS, TimeUnit.HOURS
        ).setConstraints(
            Constraints.Builder()
                .setRequiredNetworkType(NetworkType.TEMPORARILY_UNMETERED)
                .setRequiresCharging(true)
                .build()
        )
            .addTag(TAG_FETCH_LATEST_NEWS)

        workManager.enqueueUniquePeriodicWork(
            FETCH_LATEST_NEWS_TASK,
            ExistingPeriodicWorkPolicy.KEEP,
            fetchNewsRequest.build()
        )
    }
```

```kotlin
private const val FETCH_LATEST_NEWS_TASK = "FetchLatestNewsTask"
private const val TAG_FETCH_LATEST_NEWS = "FetchLatestNewsTaskTag"

class NewsTasksDataSource(
    private val workManager: WorkManager
) {
    fun fetchNewsPeriodically() {
        val fetchNewsRequest = PeriodicWorkRequestBuilder<RefreshLatestNewsWorker>(
            REFRESH_RATE_HOURS, TimeUnit.HOURS
        ).setConstraints(
            Constraints.Builder()
                .setRequiredNetworkType(NetworkType.TEMPORARILY_UNMETERED)
                .setRequiresCharging(true)
                .build()
        )
            .addTag(TAG_FETCH_LATEST_NEWS)

        workManager.enqueueUniquePeriodicWork(
            FETCH_LATEST_NEWS_TASK,
            ExistingPeriodicWorkPolicy.KEEP,
            fetchNewsRequest.build()
        )
    }

    fun cancelFetchingNewsPeriodically() {
        workManager.cancelAllWorkByTag(TAG_FETCH_LATEST_NEWS)
    }
}
```

Handling UI Events



Handling UI Events

```
class LatestNewsActivity : AppCompatActivity() {

    private lateinit var binding: ActivityLatestNewsBinding
    private val viewModel: LatestNewsViewModel by viewModels()

    override fun onCreate(savedInstanceState: Bundle?) {
        /* ... */

        // The expand details event is processed by the UI that
        // modifies a View's internal state.
        binding.expandButton.setOnClickListener {
            binding.expandedSection.visibility = View.VISIBLE
        }

        // The refresh event is processed by the ViewModel that is in charge
        // of the business logic.
        binding.refreshButton.setOnClickListener {
            viewModel.refreshNews()
        }
    }
}
```

```
@Composable
fun LatestNewsScreen(viewModel: LatestNewsViewModel
        = viewModel()) {
    // State of whether more details should be shown
    var expanded by remember { mutableStateOf(false) }
    Column {
        Text("Some text")
        if (expanded) {
            Text("More details")
        }
        Button(
            // The expand details event is processed by the UI that
            // modifies this composable's internal state.
            onClick = { expanded = !expanded }
        ) {
            val expandText = if (expanded) "Collapse" else "Expand"
            Text("$expandText details")
        }
        // The refresh event is processed by the ViewModel
        // that is in charge of the UI's business logic.
        Button(onClick = { viewModel.refreshNews() }) {
            Text("Refresh data")
        }
    }
}
```

User events in RecyclerViews

```
data class NewsItemUiState(
    val title: String,
    val body: String,
    val bookmarked: Boolean = false,
    val publicationDate: String,
    val onBookmark: () -> Unit
)
```

```
class LatestNewsViewModel(
    private val formatDateUseCase: FormatDateUseCase,
    private val repository: NewsRepository
)
    val newsListUiItems = repository.latestNews.map { news ->
        NewsItemUiState(
            title = news.title,
            body = news.body,
            bookmarked = news.bookmarked,
            publicationDate = formatDateUseCase(news.publicationDate),
            // Business logic is passed as a lambda function that the
            // UI calls on click events.
            onBookmark = {
                repository.addBookmark(news.id)
            }
        )
    }
}
```

⚠ **Warning:** It's bad practice to pass the ViewModel into the `RecyclerView` adapter because that tightly couples the adapter with the ViewModel class.

Consuming events and trigger state updates

```
// Models the UI state for the Latest news screen.
data class LatestNewsUiState(
    val news: List<News> = emptyList(),
    val isLoading: Boolean = false,
    val userMessage: String? = null
)
```

Consuming events and trigger state updates

```
// Models the UI state for the Latest news screen.
data class LatestNewsUiState(
    val news: List<News> = emptyList(),
    val isLoading: Boolean = false,
    val userMessage: String? = null
)
```

```
class LatestNewsViewModel(/* ... */) : ViewModel() {

    var uiState by mutableStateOf(LatestNewsUiState())
        private set

    fun refreshNews() {
        viewModelScope.launch {
            // If there isn't internet connection, show a new message on the screen.
            if (!internetConnection()) {
                uiState = uiState.copy(userMessage = "No Internet connection")
                return@launch
            }

            // Do something else.
        }
    }

    fun userMessageShown() {
        uiState = uiState.copy(userMessage = null)
    }
}
```

Consuming events and trigger state updates

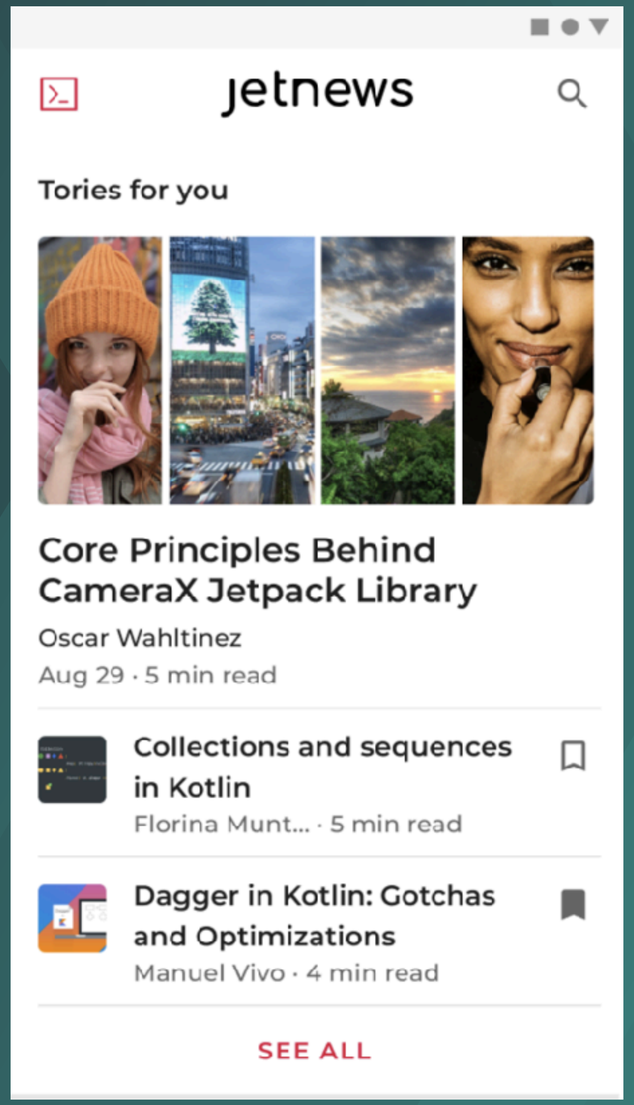
```
@Composable
fun LatestNewsScreen(
    snackbarHostState: SnackbarHostState,
    viewModel: LatestNewsViewModel = viewModel(),
) {
    // Rest of the UI content.

    // If there are user messages to show on the screen,
    // show it and notify the ViewModel.
    viewModel.uiState.userMessage?.let { userMessage ->
        LaunchedEffect(userMessage) {
            snackbarHostState.showSnackbar(userMessage)
            // Once the message is displayed and dismissed, notify the ViewModel.
            viewModel.userMessageShown()
        }
    }
}
```

Navigation Events

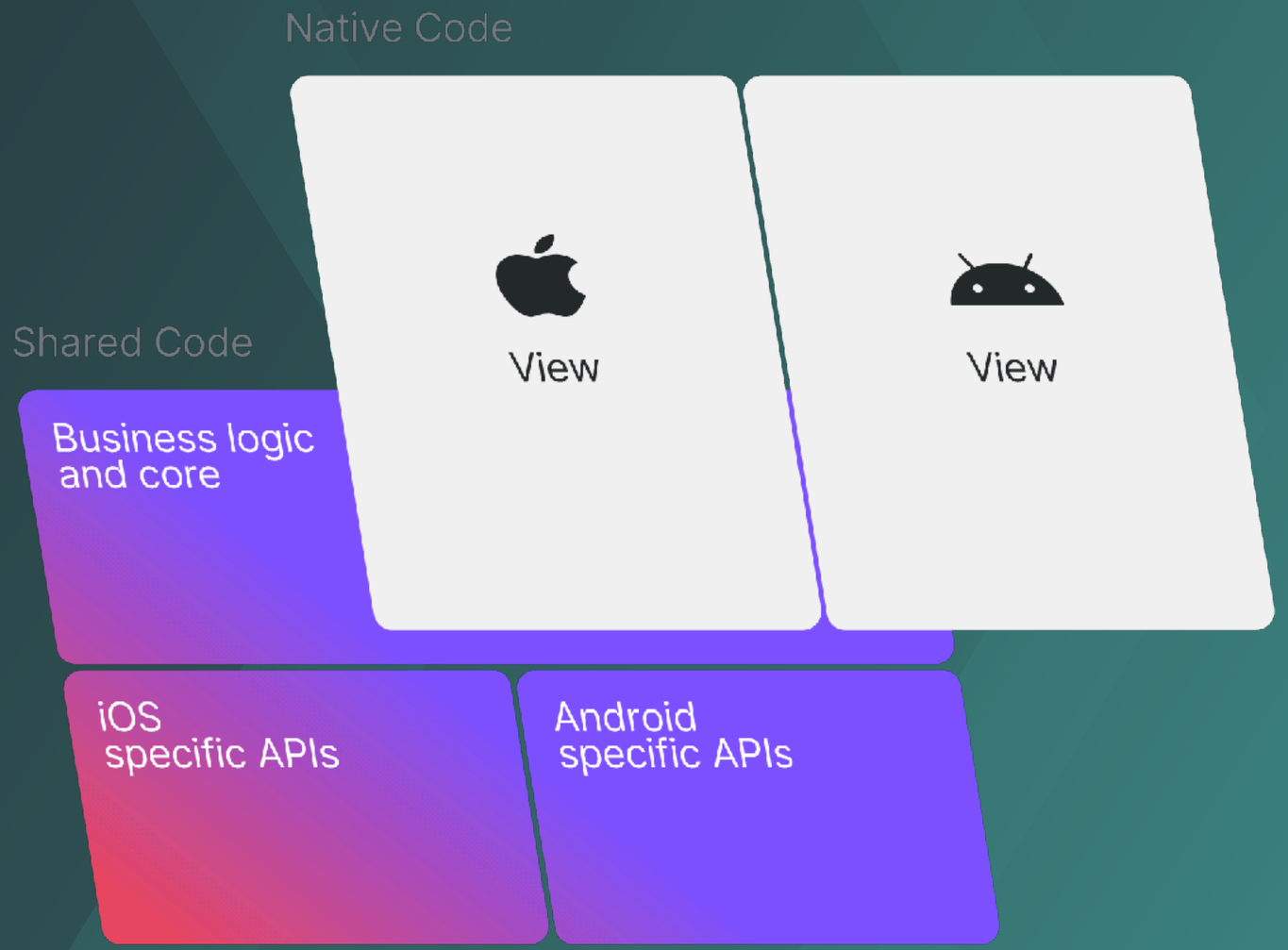
```
@Composable
fun NewsApp() {
    val navController = rememberNavController()
    NavHost(navController = navController, startDestination = "latestNews") {
        composable("latestNews") {
            MyScreen(
                // The navigation event is processed by calling the NavController
                // navigate function that mutates its internal state.
                onProfileClick = { navController.navigate("profile") }
            )
        }
        /* ... */
    }
}

@Composable
fun LatestNewsScreen(
    viewModel: LatestNewsViewModel = viewModel(),
    onProfileClick: () -> Unit
) {
```

Jetnews

DEMO



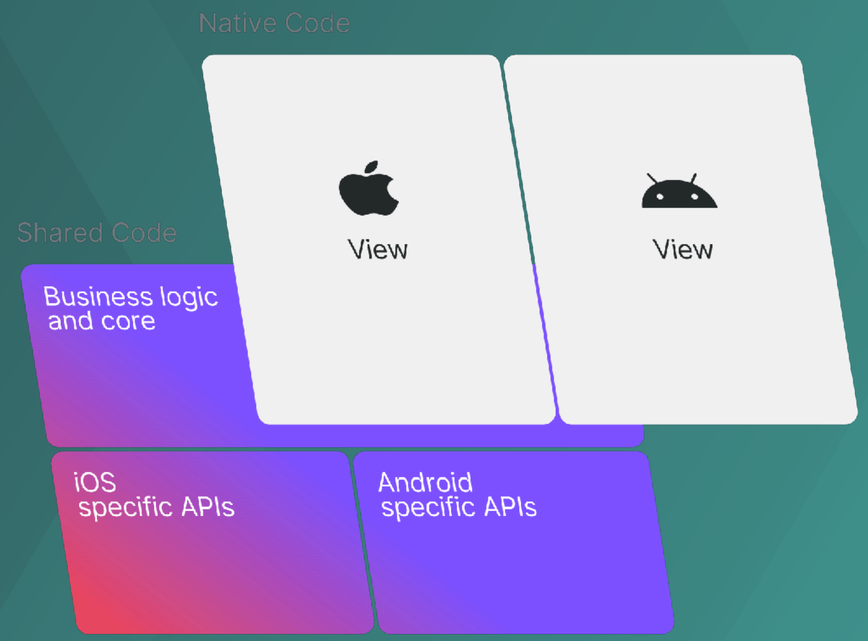
KMM



https://kotlinlang.org/docs/multiplatform-intro.html

Supported platforms

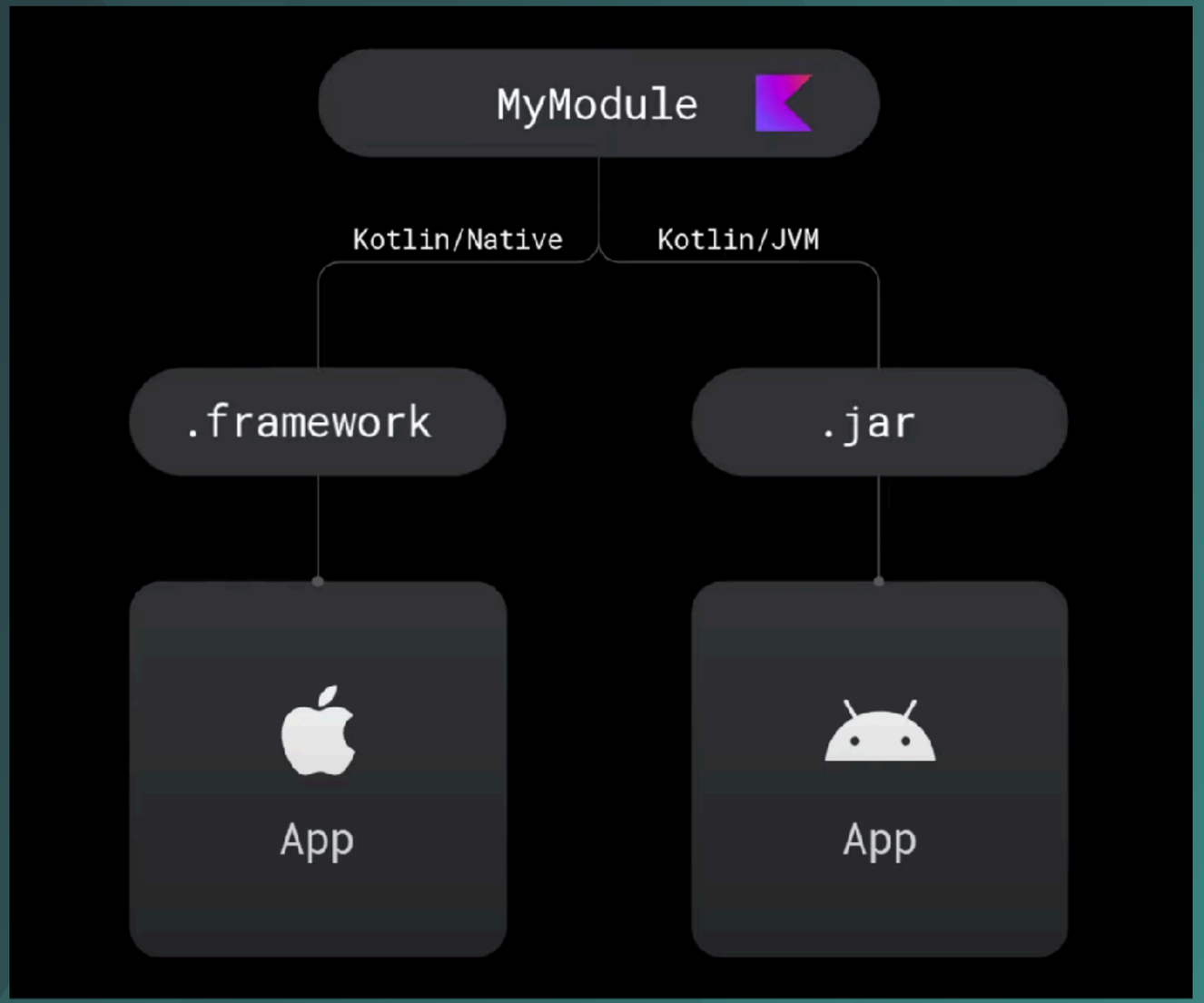
- Android applications and libraries.
- Android Native Development Kit (NDK).
- Apple iOS on ARM64 (>= iPhone 5s).
- Apple watchOs on ARM64 (>= Series 4).



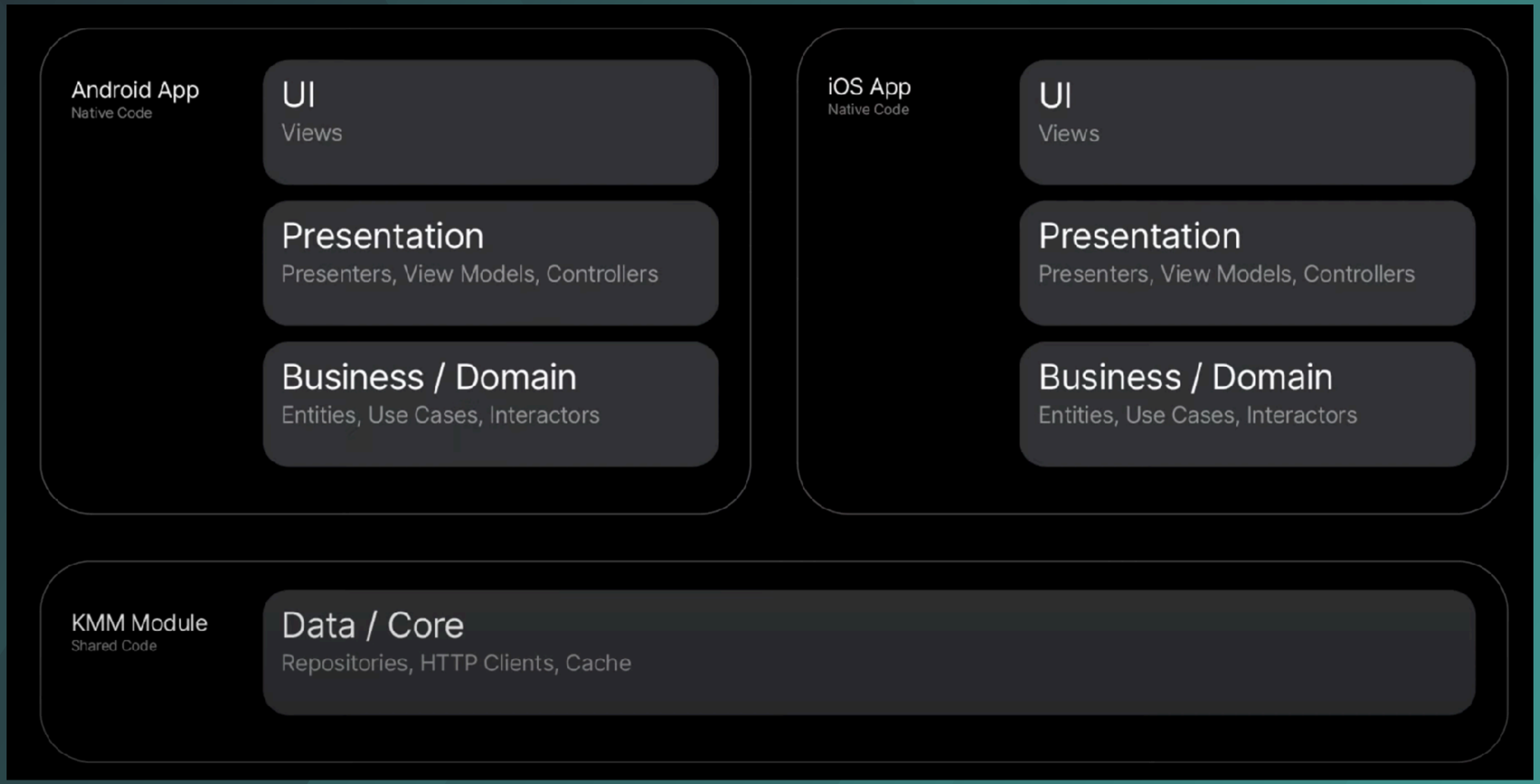
KMP - Supported platforms

Target platform	Target preset
Kotlin/JVM	`jvm`
Kotlin/JS	`js`
Android applications and libraries	`android`
Android NDK	`androidNativeArm32`, `androidNativeArm64`, `androidNativeX86`, `androidNativeX64`
iOS	`iosArm32`, `iosArm64`, `iosX64`, `iosSimulatorArm64`
watchOS	`watchosArm32`, `watchosArm64`, `watchosX86`, `watchosX64`, `watchosSimulatorArm64`
tvOS	`tvosArm64`, `tvosX64`, `tvosSimulatorArm64`
macOS	`macosX64`, `macosArm64`
Linux	`linuxArm64`, `linuxArm32Hfp`, `linuxMips32`, `linuxMipsel32`, `linuxX64`
Windows	`mingwX64`, `mingwX86`
WebAssembly	`wasm32`

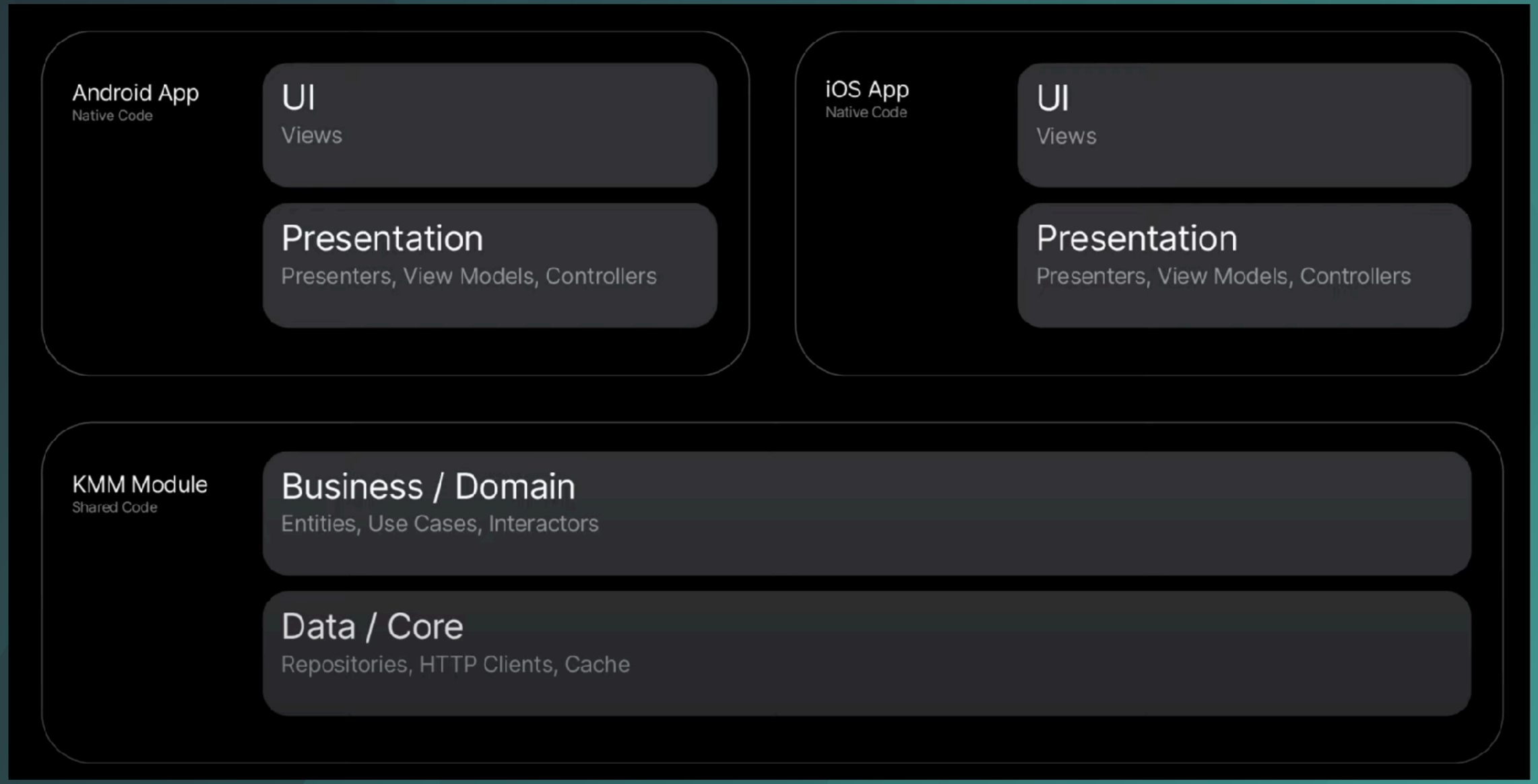
Compilation



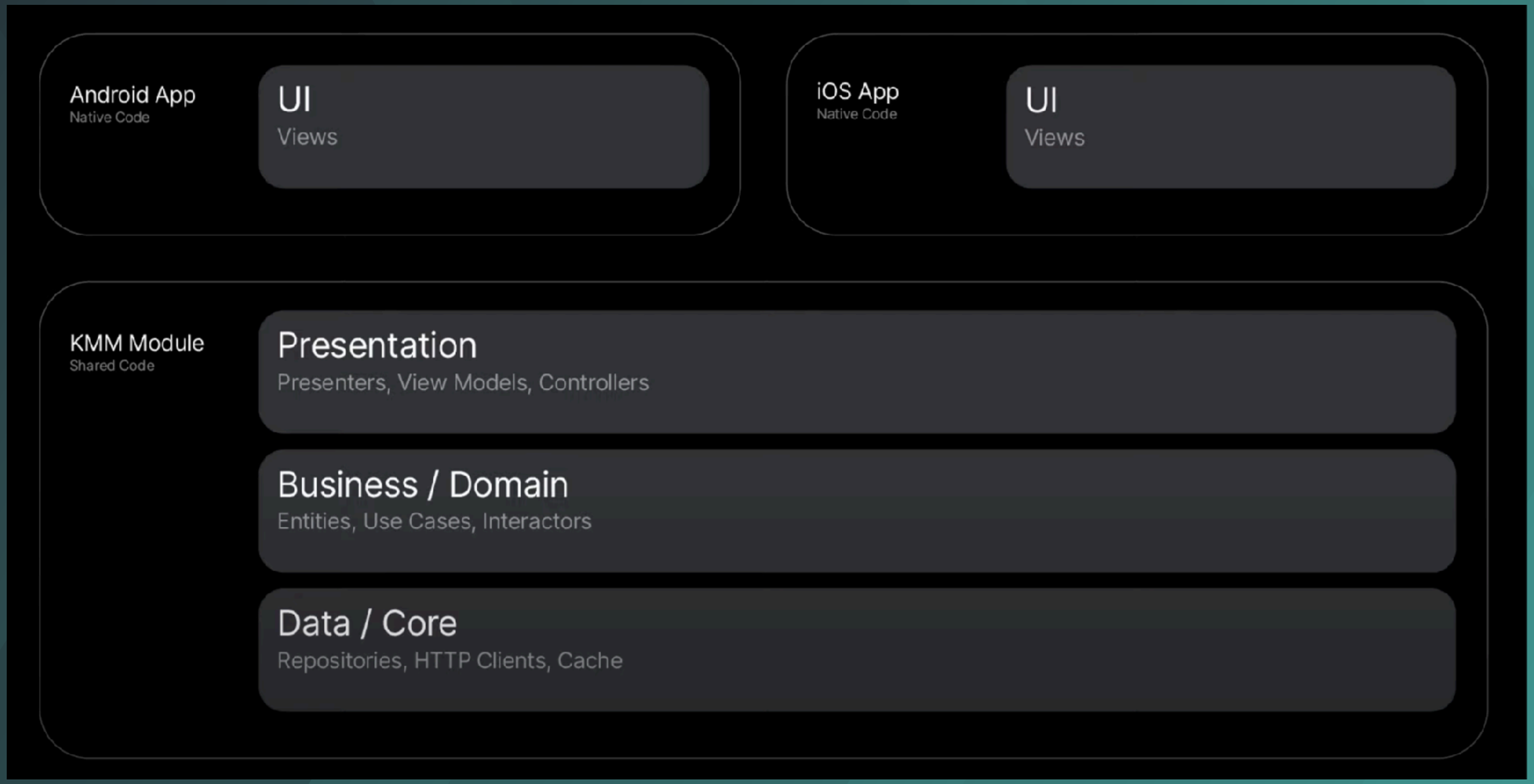
Architecture



Architecture

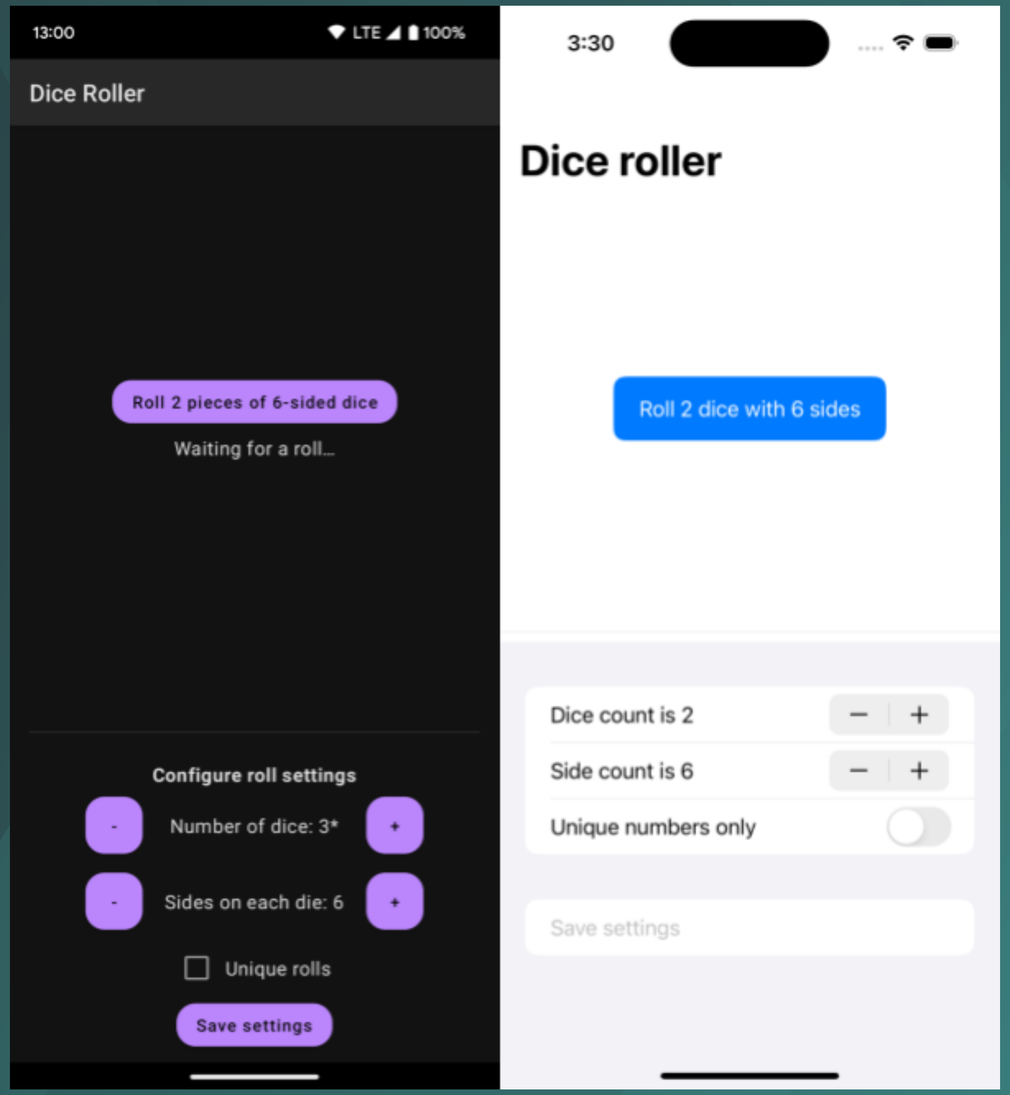


Architecture



DiceRoller

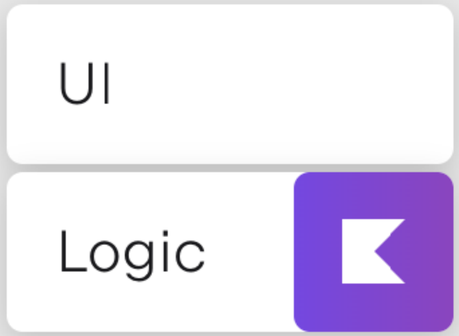
DEMO



Suitable for all kinds of projects

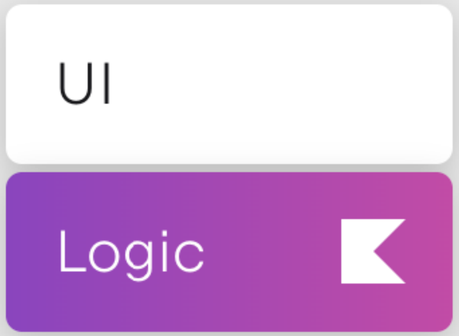
Share a piece of logic

Improve your app's stability by sharing an isolated and critical part of the app. Reuse the Kotlin code you already have to keep the applications in sync.



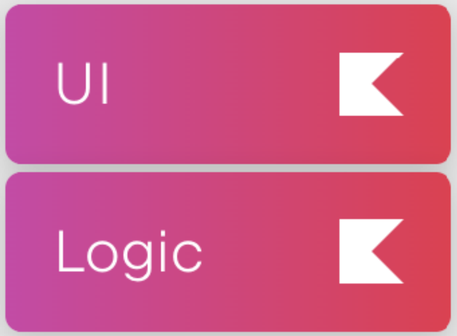
Share logic and keep the UI native

Use Kotlin Multiplatform when you start a new project, and implement data handling and business logic just once. Keep the UI native to meet the most stringent requirements.



Share up to 100% of the code

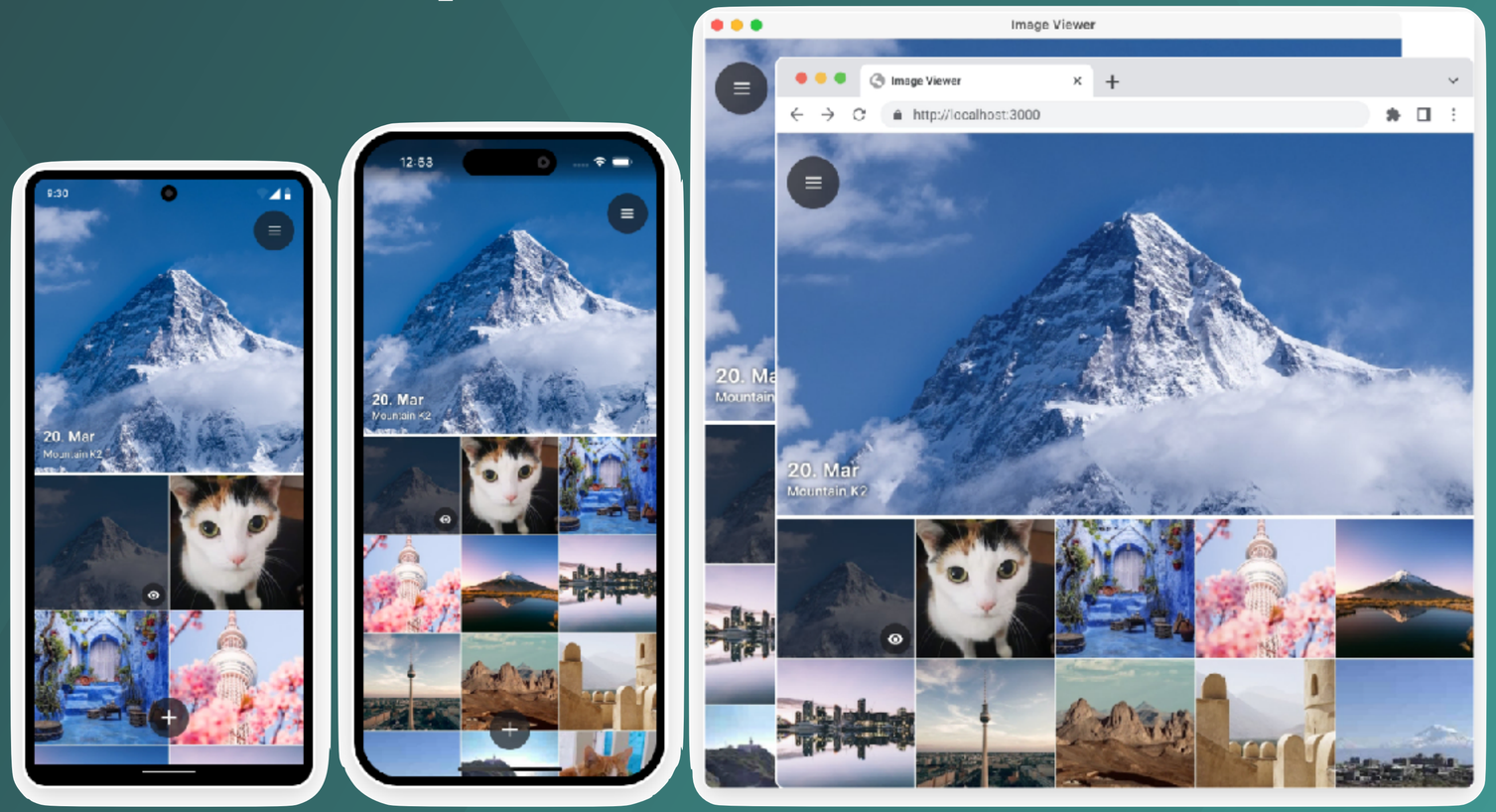
Elevate development efficiency and share up to 100% of your code with Compose Multiplatform – a modern declarative framework by JetBrains for sharing UI across multiple platforms.



https://www.jetbrains.com/kotlin-multiplatform/

Build UI once with Compose Multiplatform

- Accelerated UI development
  - Save yourself the effort of keeping different UI implementations in sync and get your app into the hands of your users more quickly.
- Component-level reuse
  - Build your UIs with customizable widgets that you can use on all of your target platforms. Use premade themes to get started quickly, or create your own visual style down to the very pixel.
- Use native components when needed
  - When you need native UI widgets or want to embed your shared UI in existing native apps, you can do so easily.



https://www.jetbrains.com/kotlin-multiplatform/

DEMO

Real-world success stories

McDonald's leverages Kotlin Multiplatform for their Global Mobile App, enabling them to build a codebase that can be shared across platforms, removing the need for codebase redundancies.

9GAG

9GAG opted for Kotlin Multiplatform after trying both Flutter and React Native. They gradually adopted the technology and now ship features faster, while providing a consistent experience to their users.

Quizlet

Kotlin Multiplatform drives global learning platform Quizlet's web and mobile apps, which boast a combined 100 million active installs. By transitioning their shared code from JavaScript to Kotlin, they significantly improved the performance of both their Android and iOS applications.

NETFLIX

Kotlin Multiplatform helps tech giant Netflix optimize product reliability and delivery speed, which is crucial for serving their customers' constantly evolving needs.

PHILIPS

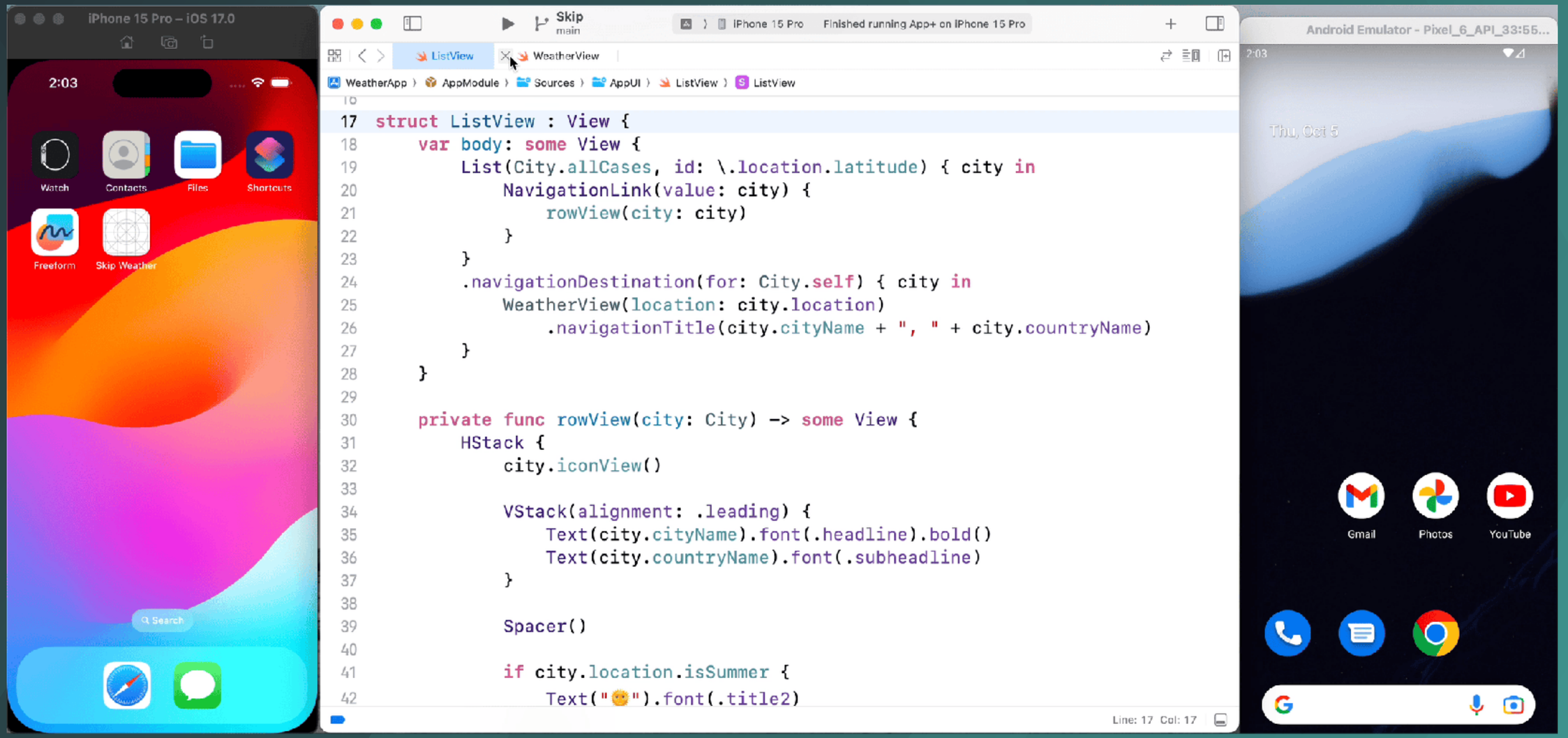
Philips utilizes Kotlin Multiplatform to develop the HealthSuite Digital Platform mobile SDK. With KMP, they not only accelerated the implementation of new features but also fostered increased collaboration between Android and iOS developers.

vmware

VMware is using Kotlin Multiplatform in various modules to enable different use cases in a consistent, cross-platform way across their Workspace ONE productivity app portfolio.

https://www.jetbrains.com/kotlin-multiplatform/

Build SwiftUI apps for iOS and Android with Skip



https://skip.tools/

https://skip.tools/

Installation

Install Skip by running the Terminal command:

```
brew install skiptools/skip/skip
```

This will download and install the `skip` tool itself, as well as the `gradle` and JDK dependencies that are necessary for building and testing the Kotlin/Android side of your apps.

Ensure that the development prerequisites are satisfied by running:

```
skip checkup
```

If the checkup fails, check the FAQ for common solutions.

Once the checkup passes, you're ready to start developing with Skip!

App Template

DEMO

Creating an App

Create a new app project with the command:

```
skip init --open-xcode --appid=bundle.id project-name AppName
```

For example:

```
skip init --open-xcode --appid=com.xyz.HelloSkip hello-skip HelloSkip
```

https://skip.tools/

Lecture outcomes

- Learn about App Architecture.
- How to handle events.
- KMM/KMP.
- skip.tools