KMP

for Mobile Native Developers

A Hands-on Guide to **Kotlin Multiplatform** for Android and iOS

Overview

This book is a comprehensive guide to Kotlin Multiplatform (KMP) aimed at native mobile developers. It begins with an introduction to KMP, explaining how this JetBrains technology enables code sharing across different platforms such as Android, iOS, web, and desktop.

The book covers several fundamental topics, including:

- Basic Kotlin Multiplatform project structure
- Modularization and architectural best practices
- Integration with compatible Jetpack libraries
- Database implementation and local storage
- Testing strategies in multiplatform projects

It includes practical examples with code repositories and detailed references to official documentation. It is a valuable resource for both developers starting with KMP and those looking to deepen their knowledge in multiplatform development.

Who I am

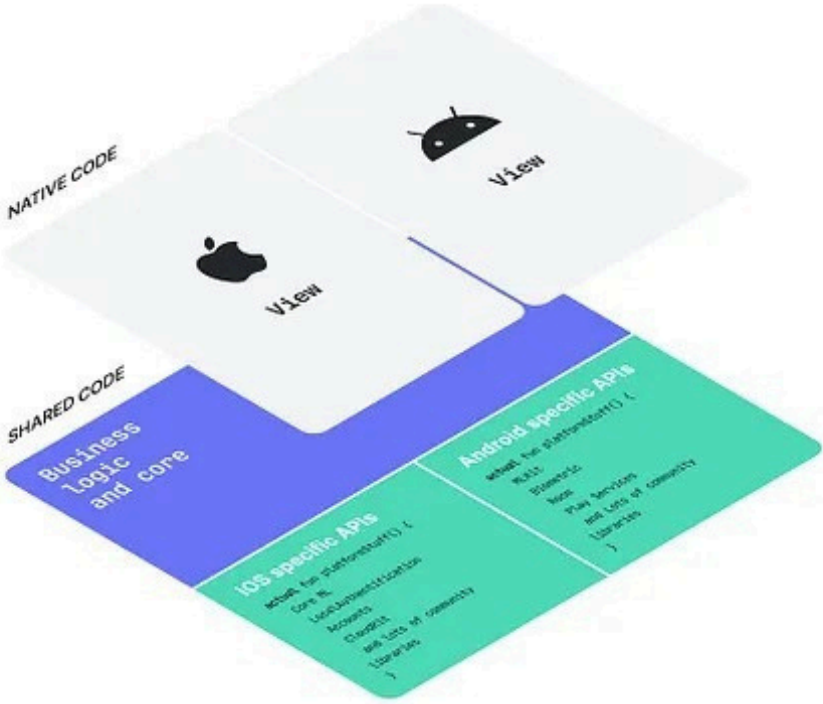
I am a software engineer with experience in mobile and backend development, focused on designing efficient and sustainable solutions, always paying attention to details to achieve clean and functional code.

I have participated in the development of meticulously crafted mobile applications, with an emphasis on delivering solid user experiences. Additionally, I enjoy sharing learnings about Android and Kotlin Multiplatform, creating content that facilitates learning and promotes knowledge exchange within the community.

I am driven by continuous learning and the exploration of new technologies, transforming complex challenges into practical and effective solutions. I believe in collaborative work as the foundation for building robust systems and applications that generate positive impact.

Content

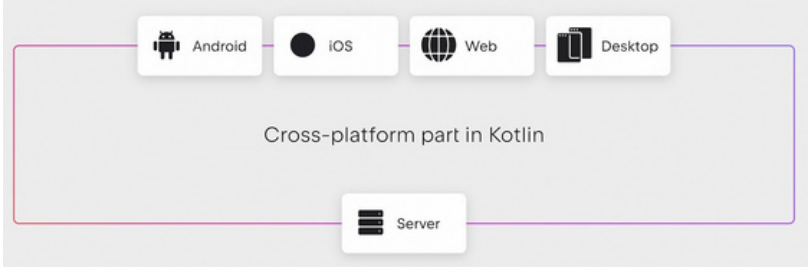
Chapter 1: Introduction to Kotlin Multiplatform



Chapter 1: Introduction to Kotlin Multiplatform

What is Kotlin Multiplatform?

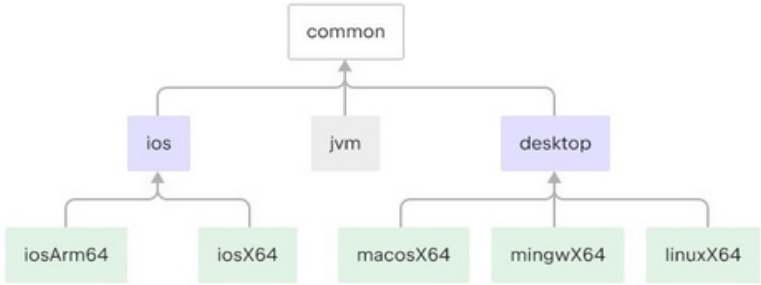
Kotlin Multiplatform is a technology that simplifies cross-platform development by allowing code sharing between different platforms, reducing development and maintenance time while maintaining the advantages of native programming. Developed by JetBrains, this technology enables developers to write code in Kotlin and share it across Android, iOS, web, and desktop. Developers can share business logic, data models, and other components across platforms, minimizing code duplication and facilitating maintenance. While not all elements can be shared due to inherent platform differences, Kotlin Multiplatform provides tools and libraries to optimize the amount of shared code. This code-sharing capability not only reduces development and maintenance time but also preserves the flexibility and advantages of native programming.



Code Sharing Across Platforms

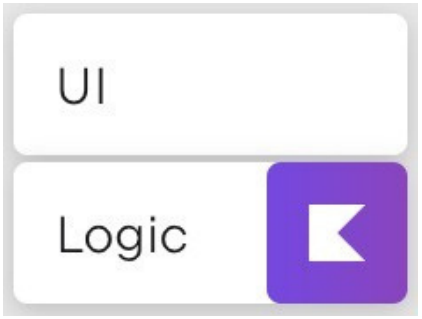
Kotlin Multiplatform enables maintaining a single codebase for application logic across different platforms. Additionally, it leverages the advantages of native programming, including high performance and full access to native SDKs. Kotlin offers two main mechanisms for sharing code:

- Share common code across all project platforms.
- Share code selectively between specific platforms to
- maximize reuse on similar platforms.



Strategies for Sharing Our Code

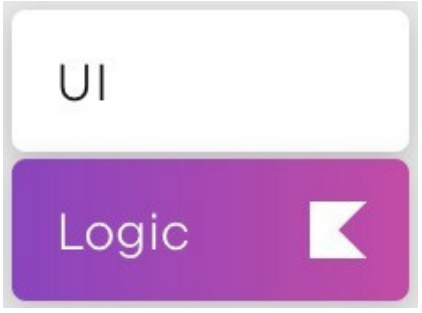
Share a Piece of Logic We can start by sharing an isolated and critical part of the application, reusing existing Kotlin code to keep applications in sync. This strategy aims to share the smallest and most significant logical unit of our application. What is a 'logical unit'? It's a portion of our application that solves a specific problem—such as validations or use cases—and is platform-independent. It's essential to promote or aspire to a good base design from the beginning.



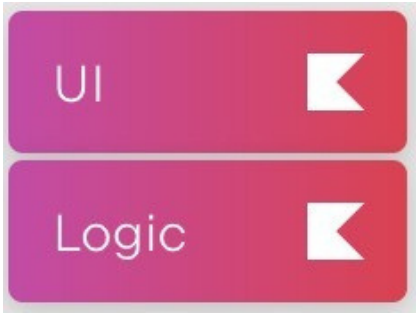
Share Logic and Keep Native UI

When starting a new project, consider using Kotlin Multiplatform to implement data handling and business logic just once. Keep the user interface native to meet the most demanding requirements. While ideal for new projects, we can also leverage existing Android code in Kotlin, reusing already developed implementations.

The strategy is to maintain user interfaces in native frameworks, as they are crucial for user experience, while sharing the application's logic and infrastructure.



Share up to 100% of the code Share up to 100% of your code with Compose Multiplatform (https://bit.ly/3QuV1qL), a modern declarative framework for creating user interfaces across multiple platforms. With Compose Multiplatform, you can develop shared user interfaces for all platforms. While this technology is still evolving, it represents a promising option for new mobile application projects. In its current state, Compose Multiplatform implements Material Design principles, which may have some limitations compared to iOS's Design System. However, the JetBrains team is developing Cupertino support (Apple's Design System) for future versions.

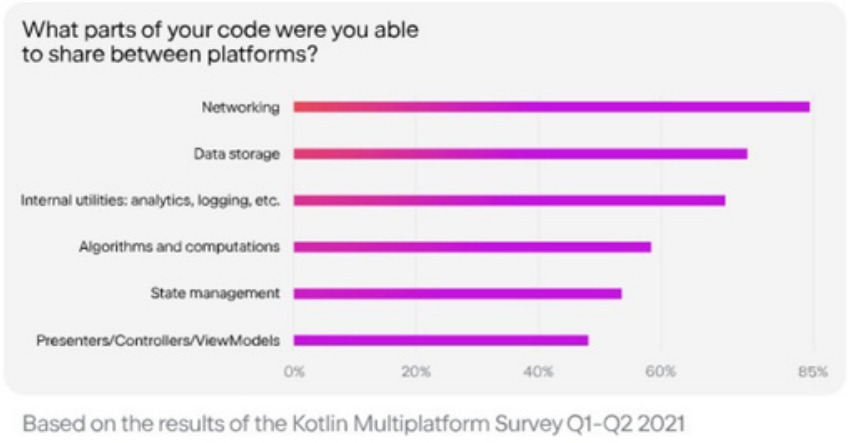


How to Really Benefit from Code Sharing

So far, we've explored the theoretical promises of Kotlin Multiplatform, but now let's examine its practical benefits and the libraries we can use when starting or migrating our development.

What Parts of Your Code Could You Share Across Platforms?

In 2021, JetBrains conducted a survey asking: "*What parts of your code were you able to share across platforms?*". While there isn't precise data about KMP adoption at that time, the survey results appear consistent when examining software architecture and design principles.



What parts of your code were you able to share across platforms?

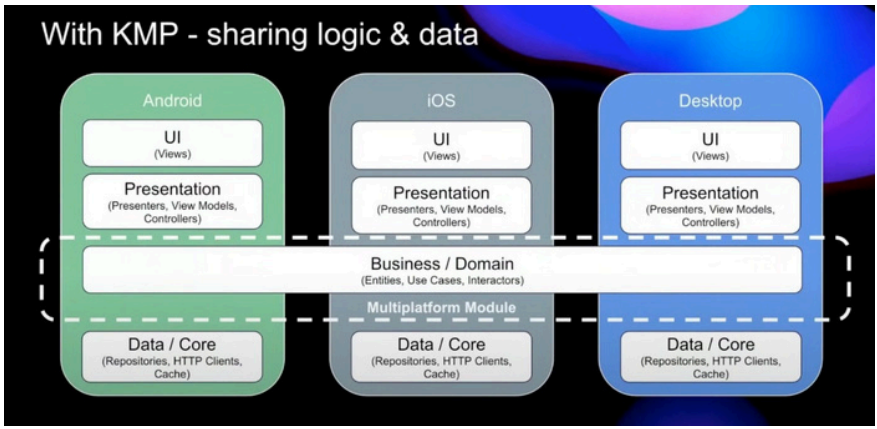
This is why it's important to establish a good design, as we form a common team language when implementing solutions, regardless of the platform.

Defining an Architecture When talking about architecture, we naturally think of **Clean Architecture**. This methodology consists of architectural patterns that separate frameworks and external elements from our domain and business logic. To implement it, we need to understand three fundamental concepts:

Domain
Fundamental concepts of our context (User, Product, Cart, etc.) and business rules defined exclusively by us (domain services).

Application
The layer where our application use cases reside (register user, publish product, add product to cart, etc.).

Infrastructure
Code that varies according to external decisions. This layer contains the implementations of interfaces defined at the domain level. We use the Dependency Inversion Principle (DIP) from SOLID to decouple from external dependencies.
This is where frameworks and external components are integrated, such as Repositories, HTTP Clients, and Caches.

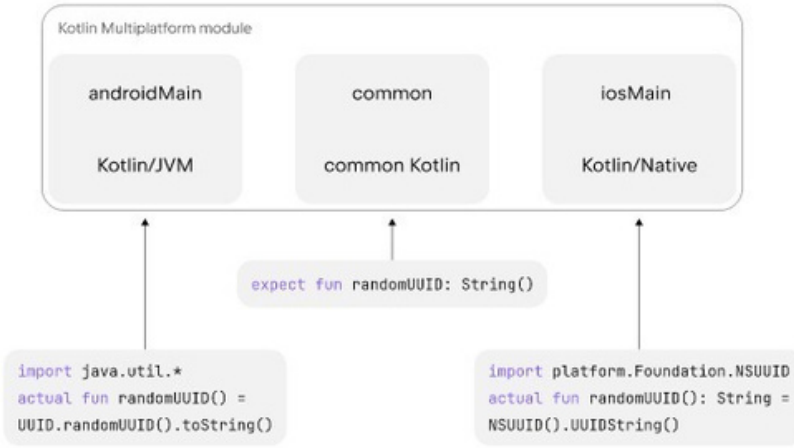


Components in an application design

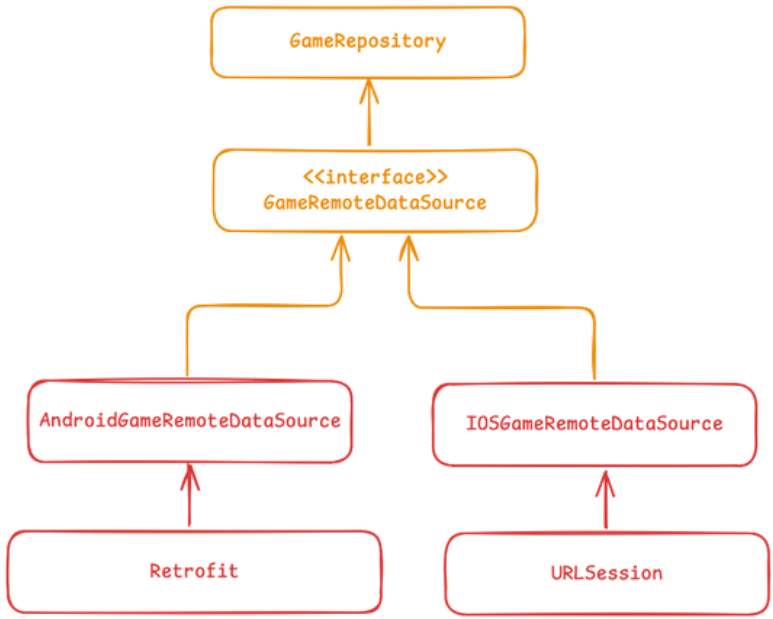
Since our domain and application layers exclusively encapsulate business logic, they constitute the main code to be shared across platforms. Without this code-sharing capability, we would have to duplicate specifications in both Kotlin and Swift for each respective platform.

Infrastructure

In this layer, we can safely implement native solutions. Following the established definitions, we use the Dependency Inversion Principle (D.I.P.) from SOLID to decouple our external dependencies. To achieve this, we define contracts using Kotlin Multiplatform's (KMP) expect-actual pattern.



Networking Example Let's look at an example of how to use native Android and iOS clients to make network requests. To begin, we'll define our repository and a remote data source. For the data source, we'll create an interface that will have two implementations: one for Android and another for iOS, as shown in the following image.



For Android, we'll use Retrofit, and for iOS, we'll use URLSession. Let's see how to implement this in code. First, we'll define an expect function that will provide a platform- specific implementation of the data sources.

```
expect fun provideGameDataSource(): GameRemoteDataSources
```

From here, we'll create our repository and implement the corresponding data sources

```
class GameRepository(
    private val remoteDataSources:
GameRemoteDataSources = provideGameDataSource(),
) {

    suspend fun fetch(): Result<GameResponse> {
        return remoteDataSources.getGames()
    }
}
```

Android Implementation

```
actual fun provideGameDataSource():
GameRemoteDataSources {
    return AndroidGameRemoteDataSources()
}


class AndroidGameRemoteDataSources :
GameRemoteDataSources {

    private val client = RetrofitClient(baseUrl)
    private val services =
client.create<GameServices>()

    override suspend fun getGames():
Result<GameResponse> {
        return runCatching {
            val games = services.getGames()
            val jsonElement =
Json.parseToJsonElement(games)

Json.decodeFromJsonElement<GameResponse>(jsonElement)
        }
    }

    companion object {
        private const val baseUrl = "https://
www.freetogame.com/api/"
    }
}
```

iOS Implementation

```
actual fun provideGameDataSource():
GameRemoteDataSources {
    return IOSGameRemoteDataSources()
}

class IOSGameRemoteDataSources : GameRemoteDataSources
{

    private val client = URLSessionClient()

    override suspend fun getGames():
Result<GameResponse> {
        return runCatching {
            val jsonString = client.fetch(baseUrl)

Json.decodeFromString<GameResponse>(jsonString)
        }
    }


    companion object {
        private const val baseUrl = "https://
www.freetogame.com/api/games"
    }
}
```

Once these configurations are implemented, our KMP project will have the following structure

Spoiler Alert: In the next chapter, we'll explore the structure of a Kotlin Multiplatform project.

As we mentioned that "*we'll rely on SOLID's DIP to decouple from external dependencies*", we can also use native code, that is, code implemented directly in the platform where we're using our multiplatform code. Let's look at an example of this in iOS with Swift. Since GameRemoteDataSource is an interface in Kotlin, it translates to a protocol in Swift.

```
class SwiftGameRemoteDataSources: GameRemoteDataSources {

    func getGames() async throws -> Any? {

        guard let url = URL(string: "https://
www.freetogame.com/api/games") else {
            throw GameServiceError.invalidURL
        }
        let (data, _) = try await
URLSession.shared.data(from: url)
        let result = try
JSONDecoder().decode(GameResponse.self, from: data)
        return result.map{ item in item.asDomainModel() }
    }

}
```

Note that type compatibility is lost here, as we can see in the getGames function signature which returns Any? We can implement this as follows

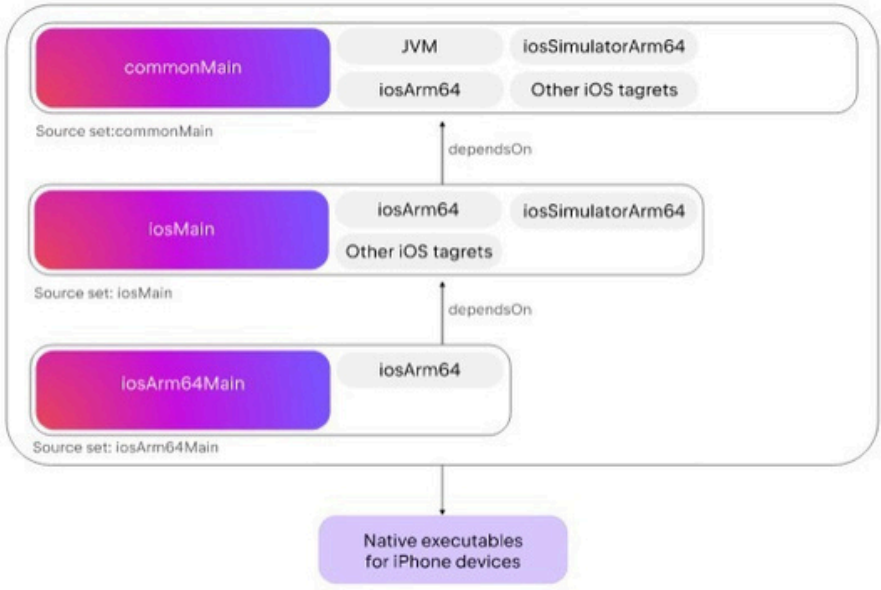
```
import Shared

@Observable
class GameViewModel{

   let repository = GameRepository(remoteDataSources:
SwiftGameRemoteDataSources())

    var data:String = ""

    func load(){
        self.repository.fetch(completionHandler:{response,_
in
            self.data = "\(String(describing: response))"
        })
    }
}
```

You can find the complete example in the following repository: GitHub - KMP for Mobile Native Developers (https://bit.ly/3XgfS4P)

GitHub - santimattius/kmp-for-mobile-native-developers at feature_01_expect_actual KMP for Mobile Native Developers. Contribute to santimattius/kmp-for-mobile-native- developers development by creating an account on GitHub.

While sharing business logic is valuable in Kotlin Multiplatform, reusing native code at the infrastructure level can be complex to maintain due to platform differences. For example, when handling preferences, Android requires a Context while iOS uses UserDefaults, creating platform-specific dependencies. To address this, we'll explore multiplatform libraries that offer common abstractions for storage, networking, and other essential functionalities, enabling more consistent implementation and greater code reuse. In the next chapter, we'll analyze the basic structure of a Kotlin Multiplatform project, including modules, source sets, and targets - fundamental elements for developing efficient multiplatform applications.

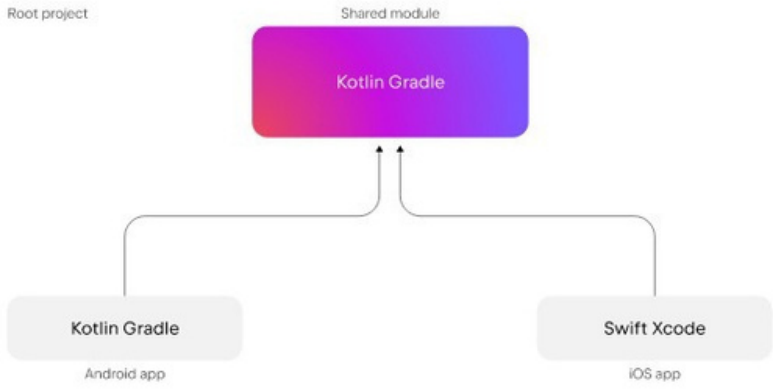
Chapter 2: Understanding the Basic Project Structure



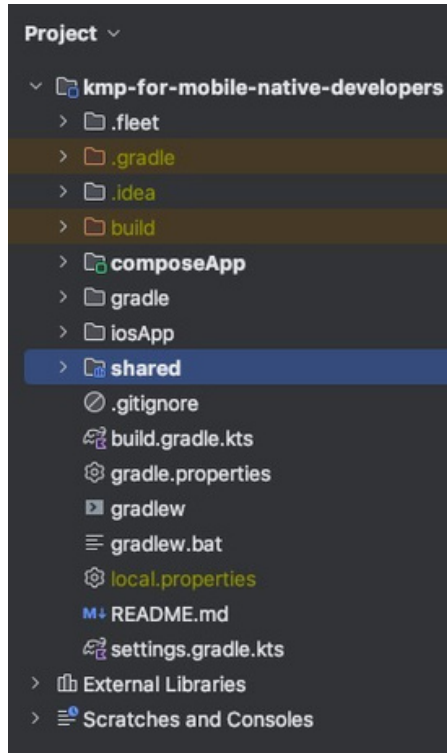
Chapter 2: Understanding the Basic Project Structure

In this chapter, we'll look at the structure of a multiplatform project and the fundamental concepts introduced when sharing code in KMP. Each Kotlin Multiplatform project includes three modules:

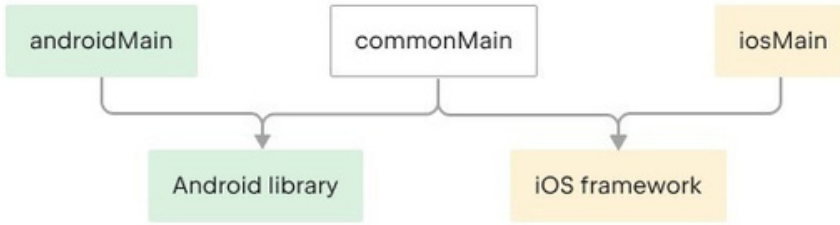
- **shared** is a Kotlin module that contains the common logic for Android and iOS applications: the code that is shared between platforms. It uses Gradle (https://bit.ly/419QejC) as a build system to automate the build process.
- **composeApp** is a Kotlin module that compiles into an Android application. It uses Gradle as a build system. The composeApp module depends on and uses the shared module as a regular Android library.
- **iosApp** is an Xcode project that compiles into an iOS application. It depends on and uses the shared module as an iOS framework. The shared module can be used as a regular framework or as a CocoaPods (https://bit.ly/41eANqa) dependency. By default, the Kotlin Multiplatform wizard creates projects that use the regular framework dependency.



The shared module consists of three source sets: androidMain, commonMain, and iosMain. A "source set" is a grouping of related files in Gradle, where each set handles its own dependencies. In Kotlin Multiplatform, these source sets can target different platforms within the shared module. The common set contains the Kotlin code that is shared, while the platform-specific sets implement specialized Kotlin code for each target. In the case of androidMain, Kotlin/JVM is used, and for iosMain, Kotlin/Native.



When the shared module is integrated as an Android library, the common Kotlin code is compiled to Kotlin/JVM. However, when integrated as an iOS framework, this same code is compiled to Kotlin/Native.



Let's now dive deeper into Source sets and Targets, and how they indicate which platforms we can share our code with.

Basic Concepts of Kotlin Multiplatform Project Structure

Targets

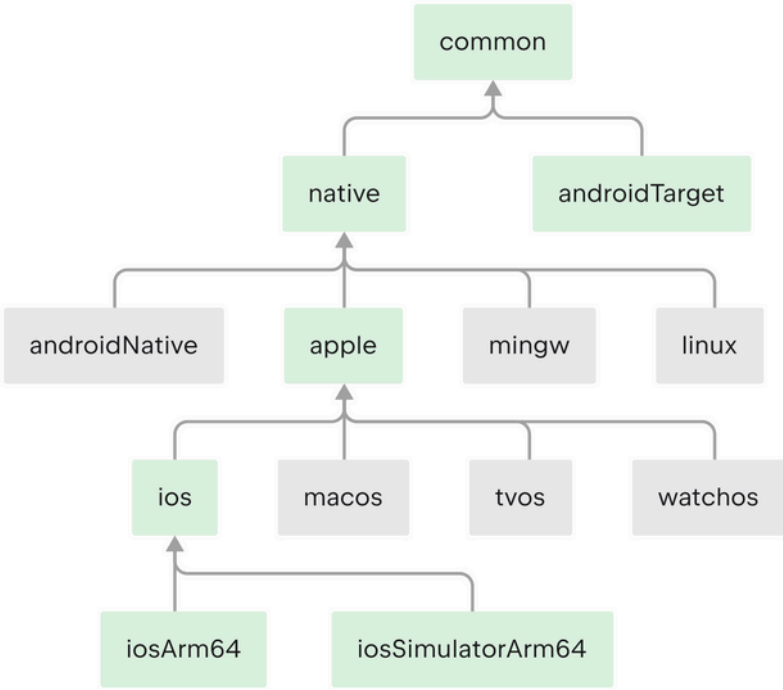
Targets define the specific platforms for which Kotlin will compile the shared code, such as Android and iOS in mobile projects. In KMP, a target is an identifier that specifies the type of compilation. It determines the format of generated binary files, available language constructs, and dependencies that can be used.

```
kotlin {
    androidTarget {
        compilations.all {
            kotlinOptions {
                jvmTarget =
JavaVersion.VERSION_1_8.toString()
            }
        }
    }

    listOf(
        iosX64(),
        iosArm64(),
        iosSimulatorArm64()
    ).forEach { iosTarget ->
        iosTarget.binaries.framework {
            baseName = "Shared"
            isStatic = true
        }
    }
}
```

As we can see in the code above, targets can specify particular configurations for each platform. For example, for Android we are indicating in kotlinOptions that the jvmTarget should be Java 1.8. There is a default hierarchy within the targets, where if our
definition is the following

```
kotlin {
    androidTarget()
    iosArm64()
    iosSimulatorArm64()
}
```

The resulting source sets hierarchy is as follows



The "source sets" shown in green are created and active in the project, while those in gray from the default template are ignored. For example, the Kotlin Gradle plugin doesn't generate code for watchOS because the project has no targets defined for this platform.

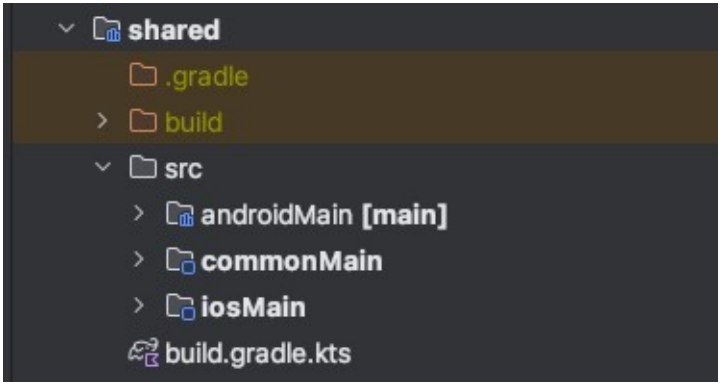
Next, we'll see how to access these source sets and how to define specific dependencies within them.

Source sets

A *Kotlin source set* is a collection of files that share targets, dependencies, and compilation configurations. It is the main mechanism for sharing code in multiplatform projects. Each source set in a multiplatform project:

- Has a unique name within the project.
- Contains files and resources, organized in a directory that bears the source set's name.
- Specifies the targets for which the code compiles, determining which language features and dependencies are available.
- Defines its own dependencies and compilation configurations.

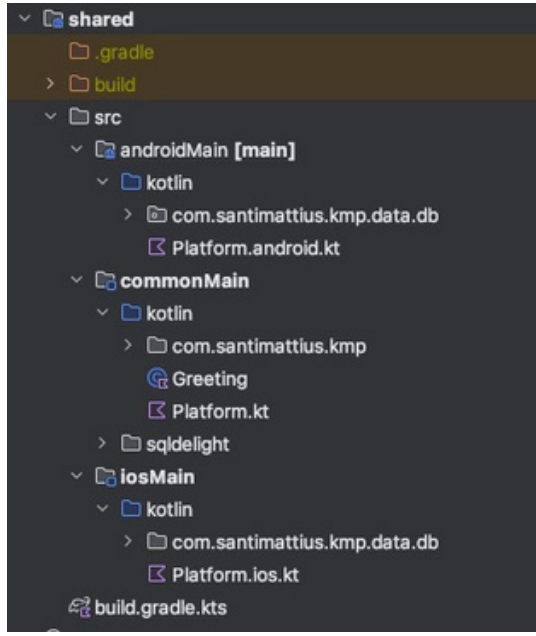
Kotlin offers several predefined source sets. Among them, commonMain stands out, as it is present in all multiplatform projects and brings together all declared targets. In the src directory of our shared module, we'll find the defined source sets. For example, in a project with commonMain, iosMain, and androidMain, the source sets are structured as follows:



In Gradle scripts, source sets are accessed by name within the kotlin.sourceSets {} block:

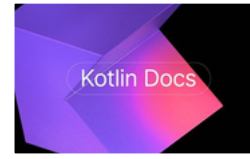
```
kotlin {

  // Targets
    androidTarget {
        compilations.all {
            kotlinOptions {
                jvmTarget = JavaVersion.VERSION_1_8.toString()
            }
        }
    }

    listOf(
        iosX64(),
        iosArm64(),
        iosSimulatorArm64()
    ).forEach { iosTarget ->
        iosTarget.binaries.framework {
            baseName = "Shared"
            isStatic = true
        }
    }

  // Sourcets
    sourceSets {
        commonMain.dependencies {
      //.....
        }

        androidMain.dependencies {
           //.....
        }

        iosMain.dependencies {
            //.....
        }
    }
}
```

Within our source sets, we can define platform-specific code for each supported platform.



The basics of Kotlin Multiplatform project structure | Kotlin
With Kotlin Multiplatform, you can share code among different platforms. This article explains the constraints of the shared code, how to distinguish between shared and platform-specific parts of

https://kotlinlang.org/docs/multiplatform-discover-project.html#source-sets



Advanced Concepts of Multiplatform Project Structure

In this section, we'll explore some advanced concepts of the Kotlin Multiplatform project structure and how they relate to Gradle implementation. This information will be useful if you need to work with low-level abstractions of Gradle builds (configurations, tasks, publications, and others) or if you're creating a Gradle plugin for Kotlin Multiplatform builds.

DependsOn

dependsOn is a specific relationship in Kotlin that connects two source sets. This connection can occur between common and platform-specific source sets, such as when jvmMain depends on commonMain, or iosArm64Main depends on iosMain.

To better understand how it works, let's take two Kotlin source sets A and B. When we write A.dependsOn(B), this means:

- A has access to B's API, including its internal declarations.
- A can implement B's expected declarations. This is fundamental, as A can only provide actuals for B if there is a dependsOn relationship, either direct or indirect.
- B must compile for all of A's targets, in addition to its own targets.
- B inherits all of A's regular dependencies.

This dependsOn relationship generates a tree-like hierarchical structure between
source sets.

```
kotlin {
    // Targets declaration
    sourceSets {
        // Example of configuring the dependsOn relation
        iosArm64Main.dependsOn(commonMain)
    }
}
```

For more information about advanced concepts of the multiplatform project structure, you can refer to the official Kotlin documentation.

Dependencies on Other Libraries or Projects

In multiplatform projects, you can configure dependencies from both published libraries and other Gradle projects. Dependency configuration in Kotlin Multiplatform follows a structure similar to Gradle, where:

- The dependencies {} block is used in the build script.
- The appropriate scope for dependencies is selected, such as implementation or api.
- The dependency is referenced through its coordinates when published in a repository (for example, "org.jetbrains.kotlinx:kotlinx-coroutines-android:1.8.0") or through its path if it's a local Gradle project (like project(":utils:concurrency")).

Dependency configuration in multiplatform projects has a distinctive feature: each Kotlin source set has its own dependencies {} block, allowing you to declare platform-specific dependencies.

```
kotlin {
    // Targets declaration
    sourceSets {
        androidMain.dependencies {

implementation("org.jetbrains.kotlinx:kotlinx-
coroutines-android:1.8.0"
        }
    }
}
```

Let's consider a multiplatform project that uses a multiplatform library, such as kotlinx.coroutines :

```
kotlin {
    androidTarget()     // Android
    iosArm64()          // iPhone devices
    iosSimulatorArm64() // iPhone simulator on Apple
Silicon

    sourceSets {
        commonMain.dependencies {
            implementation("org.jetbrains.kotlinx:kotlinx-
coroutines-core:1.8.0")
        }
    }
}
```

For more information about multiplatform dependencies, you can refer to the Kotlin Multiplatform Dependencies documentation.

Dependency Resolution

In the dependency resolution process for multiplatform projects, three fundamental aspects stand out:

- **Multiplatform Dependencies Propagation**: Dependencies declared in the commonMain source set automatically propagate to other source sets with dependsOn relationships. For example, a dependency added to commonMain extends to iosMain, jvmMain, iosSimulatorArm64Main, and iosX64Main. This prevents duplication and simplifies dependency management.
- **Intermediate and Final State of Dependency Resolution**: The commonMain source set acts as an intermediate state in dependency resolution, while platform-specific source sets represent the final state. After resolution, each multiplatform library is structured as a collection of its individual source sets, allowing for more precise management and ensuring project coherence.
- **Resolution of Dependencies by Compatible Targets**: Kotlin ensures that a dependency's source sets are compatible with those of the consumer. For example, if a source set compiles for androidTarget, iosX64, and iosSimulatorArm64, the dependency must offer source sets compatible with these targets. This ensures dependencies work across all target platforms.

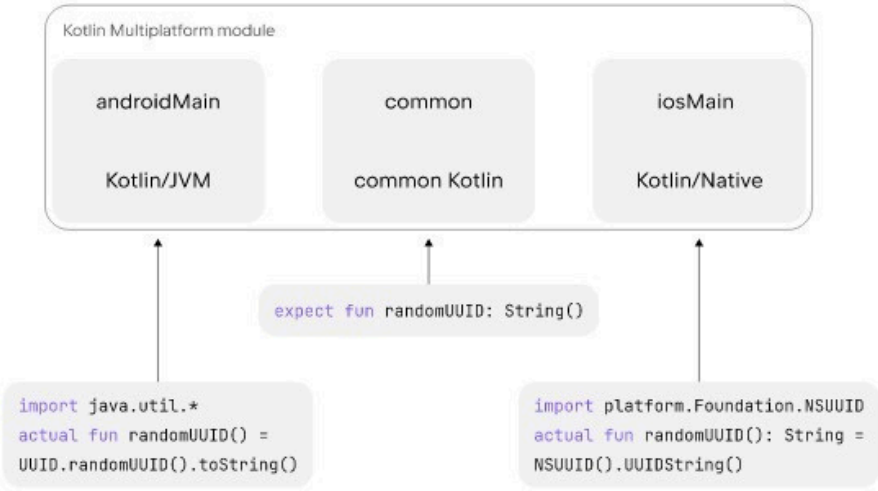
In summary, dependency resolution in multiplatform projects is based on three pillars: automatic propagation from commonMain, management of intermediate and final states, and compatibility between dependencies and consumers. This system ensures efficient and coherent management in Kotlin multiplatform projects.

Sharing Code Across Platforms

If you have business logic common to all platforms, you can avoid code duplication by sharing it in the common source set.
Some dependencies between source sets are established automatically, eliminating the need to manually specify dependsOn relationships:

- Between platform-specific source sets that depend on the common source set (for example, jvmMain, macosX64Main).
- Between main and test source sets of a specific target (such as androidMain and androidUnitTest).

To access platform-specific APIs from shared code, use Kotlin's expect/actual declarations mechanism, which we explored in the previous post.



Sharing Code Across Similar Platforms

Multiplatform projects often require creating multiple native targets that can reuse much of the common logic and third-party APIs. A common example is in iOS projects, where two targets are needed: one for iOS ARM64 devices and another for the x64 simulator. While these have separate specific source sets, they rarely require different code between them, and their dependencies are very similar. This allows sharing iOS-specific code between both targets. Therefore, it's advantageous to have a shared set of sources for both iOS targets, allowing Kotlin/Native code to directly access the common APIs of both iOS device and simulator. To implement this, you can share code between native targets using the hierarchical structure through two options:

- Using the default hierarchy template
- Manually configuring the hierarchical structure

Chapter 3: Dependency Injection

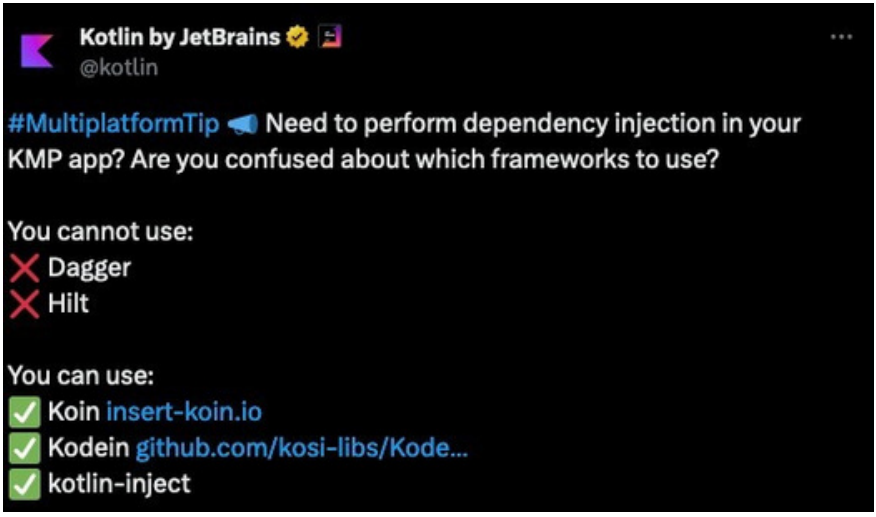
Chapter 3: Dependency Injection

In software development, dependency management is essential. Dependency Injection (DI) is a design pattern that allows application components to receive their dependencies instead of creating them internally. This promotes modularization, facilitates code reuse, and simplifies unit testing. In Kotlin Multiplatform (KMP), DI is fundamental to ensure code cohesion and portability across different platforms.

Benefits of Dependency Injection

- **Code Reuse**: By decoupling components, code can be reused across different platforms without significant changes.
- **Testability**: DI simplifies unit testing by allowing dependencies to be replaced with mock or fake objects.
    - In future articles, we'll explain how to mock objects for testing in KMP.
- **Flexibility**: DI facilitates the incorporation of new functionalities and changes in dependencies without affecting existing code, improving scalability and maintainability.
- **Clarity and Organization**: DI promotes a clearer and more organized code structure by explicitly defining dependencies and their relationships.

Although JetBrains does not provide a native dependency injection solution for the multiplatform ecosystem, they recommend using existing community-developed solutions.



Jetbrains tweet about Dependency Injection tip

Implementing Dependency Injection in Kotlin Multiplatform

To implement Dependency Injection (DI) in Kotlin Multiplatform, there are several libraries and approaches available. Each offers different advantages and disadvantages:

- **Kodein**: A native Kotlin DI library that stands out for its simple syntax and easy configuration. Its platform-independent nature makes it ideal for KMP projects.
- **Koin**: A popular library that uses a declarative syntax and is easy to configure. While it relies on reflection, this can present limitations on some platforms like iOS.
- **Kotlin Inject** is a compile-time DI library designed specifically for Kotlin Multiplatform, ensuring efficient, reflection-free injection across JVM, Native, and JavaScript platforms.
- **Manual DI**: An alternative that, while requiring more implementation effort, offers total control over the process. It involves creating and injecting dependencies externally through constructors or methods.

Let's examine each of these options in detail.

Kodein

Kodein is a dependency injection library for Kotlin. It's designed to simplify dependency management in Kotlin applications across multiple platforms: Android, iOS, Web, and Backend. Thanks to its simple syntax and straightforward configuration, it has become a popular choice for Kotlin Multiplatform projects.

Main Features of Kodein

- **Concise and Declarative Syntax**: Kodein uses an intuitivesyntax that facilitates the definition and resolution of dependencies, improving code readability and maintainability.
- **Platform Independent**: Being developed purely in Kotlin,Kodein works on any platform without worrying about compatibility with specific frameworks or libraries.
- **Kotlin Coroutines Support**: Kodein integrates perfectly with Kotlin Coroutines, facilitating dependency management in asynchronous and reactive code.
- **Modular Configuration**: Offers a modular system for configuring dependencies, simplifying the organization and maintenance of extensive projects.
- **Flexible Injection**: Allows dependency injection through both constructor and property, offering versatility in dependency resolution.
- **Annotation Support**: Includes annotations to identify components and configurations, improving code clarity.

Basic Usage of Kodein

To implement Kodein in a Kotlin Multiplatform project, the first step is to add the dependency to the configuration file. Then, you can define dependency modules and register application components using the Kodein API. Finally, you'll be able to resolve these dependencies in any section of the application where you need them.

Let's look at a basic code example using Kodein:

```
// Define a dependency module
val appModule = DI.Module("appModule") {
    bind<Database>() with singleton { Database() }
    bind<UserRepository>() with singleton {
UserRepository(instance()) }
    bind<MyService>() with singleton { MyService(instance()) }
}

// Configure the DI container
val di = DI {
    importAll(appModule)
}

// Resolve dependencies in a class
class MyService(private val userRepository: UserRepository) {
    // ...
}

// Use resolved dependencies
val myService by di.instance<MyService>()
```

In this example, a dependency module is defined that provides a database implementation and a user repository. Then, Kodein is configured with this module and an instance of MyService that depends on the user repository is resolved.

Getting started with Kodein-DI: Kodein Open Source Initiative Documentation
Kodein-DI is a Dependency Injection library. It allows you to bind your business unit interfaces with their implementation and thus having each business unit being independent.
https://kosi-libs.org/kodein/7.22/getting-started.html

Koin

Koin is a lightweight Dependency Injection library for Kotlin that stands out for its simplicity and ease of use. Compatible with Android, backend, and iOS, it prioritizes clean and readable code.

Main Features of Koin

- **Non-Intrusive**: Uses pure Kotlin functions, integrating naturally without modifying existing architecture.
- **Clear Syntax**: Employs a simple DSL that facilitates dependency definition and resolution.
- **Optimized Performance**: By not using reflection, it improves performance and is compatible with iOS/Kotlin Native.
- **Flexibility**: Allows constructor and property injection, adapting to different needs.
- **Lifecycle Control**: Manages scopes to control instance creation and destruction.
- **High Compatibility**: Easily integrates with Kotlin frameworks and libraries.

Basic Usage of Koin

To implement Koin in a Kotlin Multiplatform project, follow these steps: first, add the Koin dependency to the project configuration file. Then, define dependency modules and register application components using the Koin API. Finally, resolve dependencies where you need them in your application.

Let's look at a basic implementation example with Koin: In this example, a dependency module is defined that provides a database, a user repository, and a ViewModel. Then, Koin is configured with this module and an instance of MyViewModel, which depends on the user repository, is obtained.

```
// Define a dependency module
val myModule = module {
    single { Database() }
    single { UserRepository(get()) }
    factory { MyViewModel(get()) }
}

// Configure Koin with the defined module
startKoin {
    modules(myModule)
}

// Resolve dependencies in a class
class MyActivity : AppCompatActivity() {
    private val viewModel: MyViewModel by viewModel()
    // ...
}
```

Koin: The Kotlin Dependency Injection Framework
The Kotlin Dependency Injection Framework
https://insert-koin.io/

Kotlin Inject for Multiplatform

Kotlin Inject is a lightweight, reflection-free dependency injection (DI) library for Kotlin, designed specifically for Kotlin Multiplatform (KMP). It leverages Kotlin compiler plugins to generate dependency graphs at compile time.

Key Features

- Kotlin Inject provides several advantages over traditional DI frameworks:
- **Multiplatform Support:** Works seamlessly with JVM, Native, and JS, making it ideal for KMP projects.
- **Reflection-Free:** Unlike Dagger, it avoids runtime reflection, making it performant in Kotlin/Native environments.
- **Compile-Time Dependency Injection:** Generates dependency graphs at build time, eliminating runtime overhead.
- **Constructor Injection** – Encourages immutability and clear dependency management.
- **Lightweight & Simple –** Requires minimal setup, reducing boilerplate code.

Example: Dependency Injection in KMP Define Dependencies The first step in setting up dependency injection is to annotate classes with @Inject, signaling that they can be automatically provided by the DI system.

```
import me.tatarka.inject.annotations.Inject

// Service - Basic dependency
class UserService @Inject constructor() {
    fun getUser() = "Santiago Mattiauda"
}

// Repository - Inject dependency
class UserRepository @Inject constructor(private val
userService: UserService) {
    fun fetchUser(): String = userService.getUser()
}

// ViewModel - Inject repository
class UserViewModel @Inject constructor(private val
userRepository: UserRepository) {
    fun getUserName(): String = userRepository.fetchUser()
}
```

By marking each class with @Inject, Kotlin Inject can automatically generate the necessary dependency graph.

Creating a Dependency Container

In Kotlin Inject, a Component is an abstraction that defines the dependencies available in an application.

```
import me.tatarka.inject.annotations.Component

// Define a dependency container using @Component
@Component
abstract class AppComponent {
    abstract val userViewModel: UserViewModel

    companion object {
        fun create(): AppComponent =
AppComponent::class.create()
    }
}
```

This AppComponent class serves as the centralized factory for managing instances. By calling AppComponent.create(), we obtain an instance of the component with all dependencies properly injected.

Using the Dependency Injection System

With the component set up, we can now retrieve an instance of UserViewModel and use it:

```
fun main() {
    val appComponent = AppComponent.create()
    val viewModel = appComponent.userViewModel

    println(viewModel.getUserName()) // Output:
Santiago Mattiauda
}
```

This approach ensures that all dependencies are created and managed automatically without manual instantiation.

Manual Dependency Injection in Kotlin Multiplatform

Manual Dependency Injection (DI) is a simple alternative for managing dependencies in KMP projects without using external libraries. This approach involves creating and injecting dependencies directly into the classes that need them.

Basic Principles

- **Constructor Injection**: Dependencies are passed through constructors, keeping code explicit and clear.
- **OOP Approach**: Aligns with object-oriented programming, using clear interfaces between components.

Implementation

- **Identify**: Determine which dependencies each class needs.
- **Create**: Generate instances at the top level using patterns like Singleton.
- **Inject**: Provide dependencies through constructors.
- **Manage**: Handle the lifecycle of dependencies.

```
// Service: Base dependency
class UserService {
    fun getUser(): String = "Santiago Mattiauda"
}

// Repository: Receives dependency via constructor
class UserRepository(private val userService: UserService) {
    fun fetchUser(): String = userService.getUser()
}

// ViewModel: Also receives dependency via constructor
class UserViewModel(private val userRepository: UserRepository) {
    fun getUserName(): String = userRepository.fetchUser()
}

// Dependency Container (Manages and creates instances)
object AppContainer {
    val userService: UserService by lazy { UserService() }
    val userRepository: UserRepository by lazy {
UserRepository(userService) }
    val userViewModel: UserViewModel by lazy {
UserViewModel(userRepository) }
}

// Application usage
fun main() {
    val viewModel = AppContainer.userViewModel
    println(viewModel.getUserName()) // Output: Santiago
Mattiauda
}
```

Advantages and Challenges

- **Advantages**: Simplicity, total control, and transparency in dependency flow.
- **Challenges**: Higher coupling, manual maintenance, and possible code repetition.

Key Considerations

- **Design**: Create clear interfaces following SOLID principles.
- **Lifecycle**: Properly manage instance creation and destruction.
- **Testing**: Facilitate unit testing through mocks.
- **Scalability**: Maintain a modular and structured approach.
- **Documentation**: Maintain clear documentation and effective team communication.

Off Topic: Creating our own dependency injection framework

As the title suggests, while this is a secondary topic, it represents a viable alternative when we want to avoid external dependencies for dependency injection and reduce code repetition inherent to manual injection. While we won't delve into this topic, I'd like to conclude this article by recommending the following resource about it.

DIY´ your own Dependency Injection library!

Demystifying the internals of DI libraries

https://blog.p-y.wtf/diy-your-own-dependency-injection-ylibrar

Summary of the presented alternatives

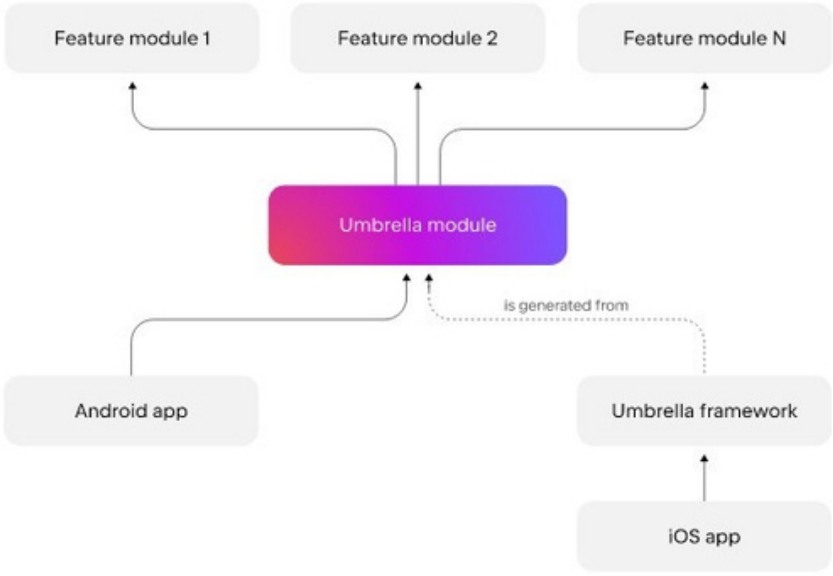
Koin is a lightweight library for implementing Dependency Injection in Kotlin applications. It stands out for its ease of use and efficiency, being highly valued by the developer community. Its cross-platform compatibility and frequent updates make it a reliable option for Kotlin Multiplatform projects.

Kodein is a robust library for managing dependencies in Kotlin Multiplatform projects. It offers an intuitive syntax, cross-platform support, and advanced features such as Kotlin Coroutines integration. Its efficiency, flexibility, and support from an active community position it as a solid solution for dependency injection in Kotlin.

Kotlin Inject is an excellent choice for Kotlin Multiplatform projects requiring efficient and lightweight Dependency Injection. By leveraging compile-time DI, it avoids the pitfalls of reflection-based approaches while providing an intuitive and scalable solution.

Manual Dependency Injection allows total control over dependency management in Kotlin Multiplatform projects without relying on external libraries. While it requires more implementation and maintenance effort, it is ideal for small projects or teams that prefer a direct approach. However, it is crucial to carefully evaluate project requirements and weigh the advantages and challenges before opting for manual DI in a KMP project.

Chapter 4: Modularization



Chapter 4: Modularization

Modularization has gained greater importance in the face of growing complexity in mobile applications and platform diversity. This strategy is fundamental for improving code maintainability, scalability, and reusability. In this scenario, Kotlin Multiplatform emerges as an ideal solution for developing mobile applications across different platforms, such as Android and iOS. Let's see how to modularize a Kotlin Multiplatform project.

Benefits of Modularization in Kotlin Multiplatform

KMP allows sharing business logic, data models, and components across various mobile applications, resulting in more efficient development and greater consistency between application versions. Modularization in a KMP project offers several significant benefits:

- **Code Reusability**: Independent modules facilitate the reuse of components and functionalities across different parts of the application and between platforms.
- **Maintainability**: Well-defined modules simplify code understanding and maintenance. Being able to develop, test, and update each module independently speeds up development and reduces errors.
- **Scalability**: Modularization facilitates project growth, allowing modules to be added or modified without affecting existing code.
- **Decoupling**: Separation into independent modules reduces coupling between components, making the code more flexible and easier to extend.

So far, we've explored the theory behind modularization. But what strategies can we implement to make the most of cross-platform development?

Strategies for Modularizing a Kotlin Multiplatform Project

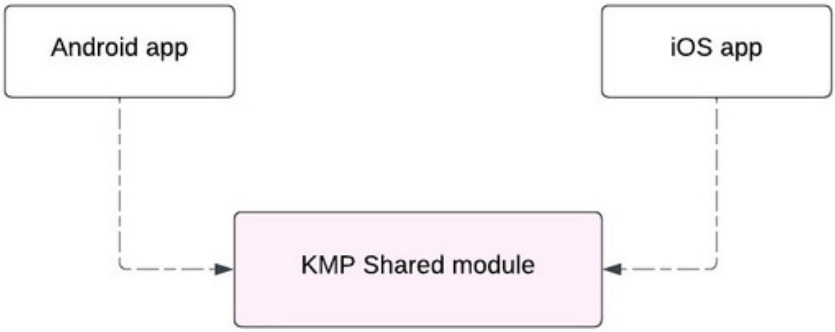
There are various strategies for modularizing a KMP project. The most common ones are:

- **Layer-based Division**: Organizes code into modules representing different architecture layers, such as presentation, business logic, and data access.
- **Feature-based Division**: Groups code related to a specific functionality into a single module, facilitating its reuse and maintenance.
- **Platform-based Division**: Separates platform-specific code into different modules, keeping shared code in a central module.
- **Domain-based Division**: Organizes code into modules representing different application domains, such as user, authentication, and purchases.

This approach is valid for both monorepos and separate repositories, as the fundamental aspects are the configurations of projects using KMP.

Modularization in Practice

When creating a KMP project, whether it's an App or Library type, a Shared module is automatically generated that will function as a shared module between both platforms: Android and iOS.



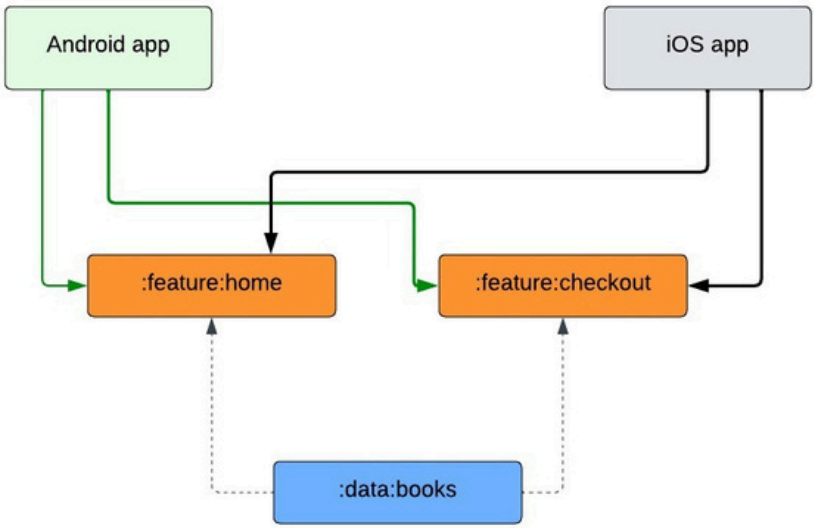
Pros

- A simple design with a single module reduces cognitive load. You don't need to think about where to put your functionality or how to logically divide it into parts.
- Works very well as a starting point.

Cons

- Build time increases as the shared module grows.
- This design doesn't allow for separate features or having dependencies only in the features that the application needs.

When a KMP project grows, it's common to add more shared modules besides the initial one. This happens naturally when implementing new functionalities in KMP instead of using native modules, or when teams gradually adopt this technology. To maintain scalable and manageable code, it's recommended to split the shared module into smaller feature modules. Let's see this represented in the following image.



Example of a book selling app

As shown in the image, we have two modules (features) that represent different flows in our application, along with a shared module (data) that these features use. Additionally, we have sub-modules to manage specific information in our application (in this case, books). While this approach offers clear benefits in the separation of responsibilities, it also presents specific challenges when generating binaries for each platform (especially in iOS, as we'll see later).

Pros

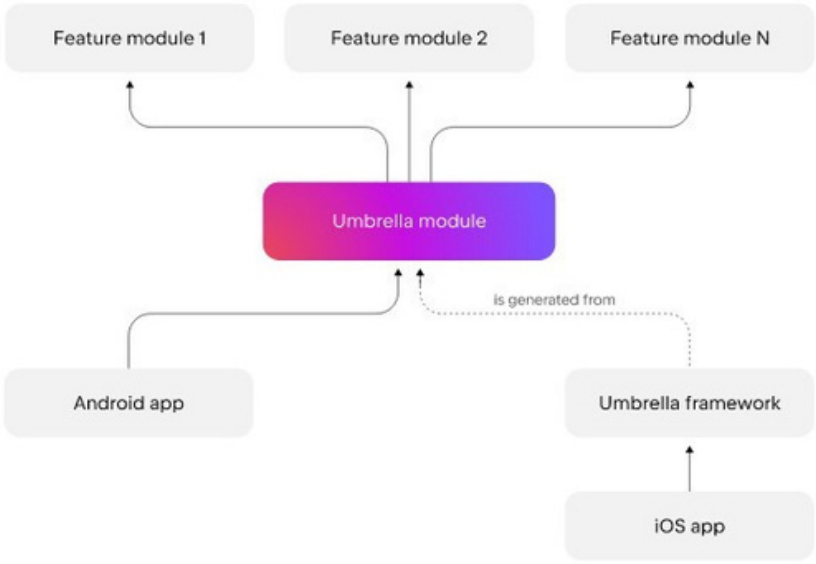
- Separation of concerns for shared code.
- Better scalability.

Cons

- More complicated setup, including umbrella framework configuration.
- More complex dependency management across modules.

Multiple Shared Modules

When working with multiple shared modules, there are important differences between platforms. In Android, the application can directly depend on all or some feature modules as needed, as it uses Gradle modules for its definition. On the other hand, the iOS application can only depend on a single framework generated by the Kotlin Multiplatform module. To handle multiple modules in iOS, it's necessary to create an additional module called the Umbrella module, which depends on all modules in use. This module is configured to generate a framework containing all modules, known as the **Umbrella** framework.

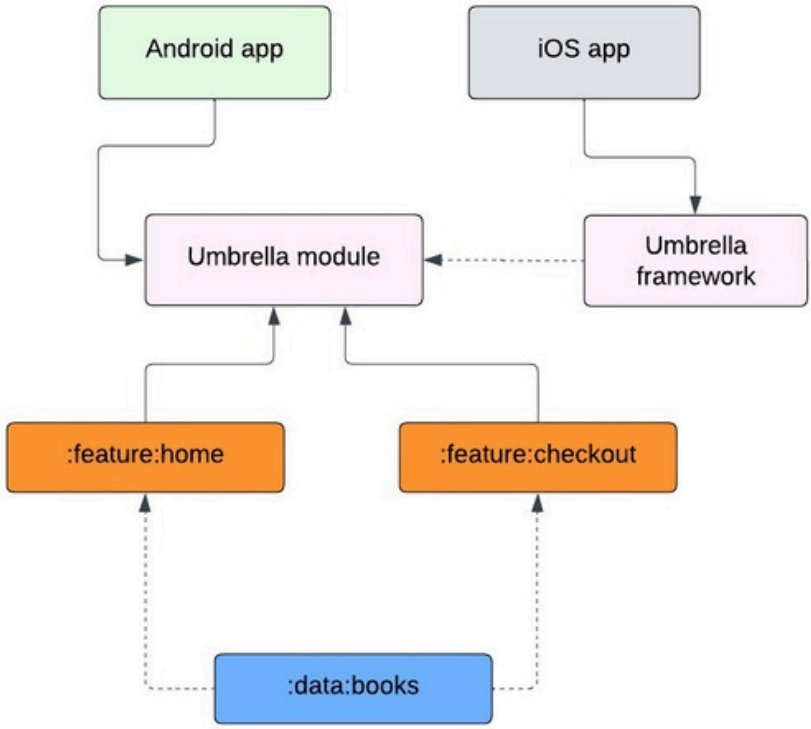


The Android application can depend on the Umbrella module to maintain consistency, or use feature modules separately. The Umbrella module typically contains utilities and dependency injection configurations. The Umbrella framework only exports selected modules, especially when consumed as a remote dependency. This helps minimize the final artifact size and excludes unnecessary auto- generated code. An important limitation of this approach is that the iOS application must consume all feature modules included in the Umbrella framework, without being able to select only some of them.

Why do you need an Umbrella framework?

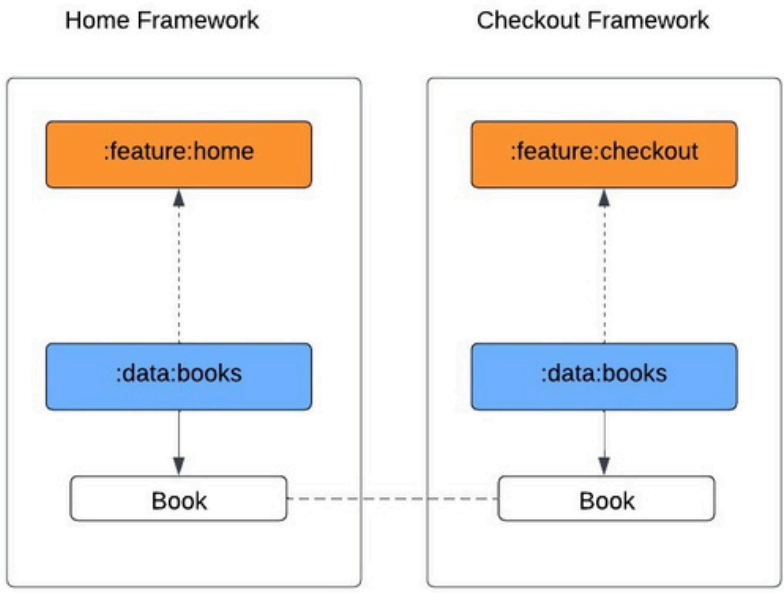
While it's technically possible to use multiple Kotlin Multiplatform module frameworks in iOS, it's not recommended. When a module is converted to a framework, it includes all its dependencies; if these are duplicated, it not only increases the application size but can also generate conflicts and errors. Kotlin avoids generating common framework dependencies to maintain an efficient binary and eliminate redundancies. Sharing these dependencies is not feasible because the Kotlin compiler cannot anticipate the requirements of other compilations. The optimal solution is to implement an Umbrella framework, which prevents dependency duplication, optimizes the final result, and avoids compatibility issues.

For more details about the exact limitations, please check the TouchLab documentation.



Exposing Multiple KMP Frameworks in Detail

Kotlin Multiplatform has a fundamental limitation: the iOS platform cannot access Kotlin modules individually. Instead, it generates a single framework containing all exported Kotlin classes. While it is possible to generate multiple frameworks, this practice is inefficient as it produces a larger binary and creates overhead due to the duplication of Kotlin standard library classes. Additionally, all shared dependencies between Kotlin modules are also duplicated. In our example, we would have something like this:

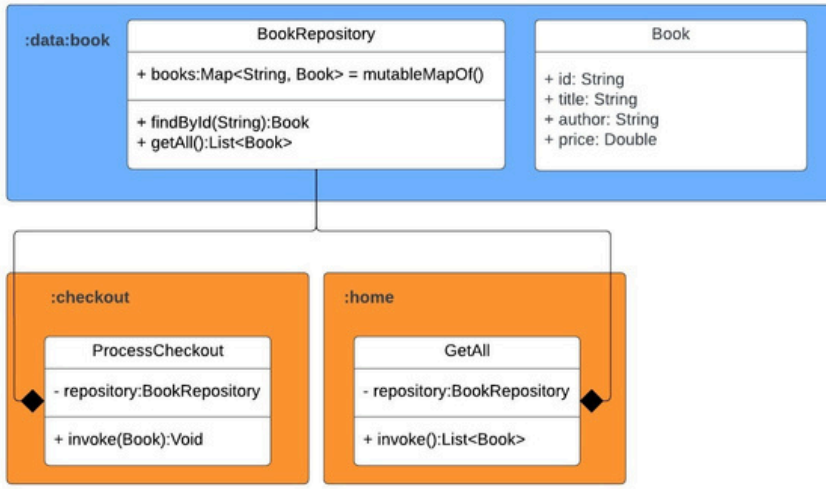


The Book entity from the Home framework and the Book entity from the Checkout framework would represent the same entity defined in the data module. However, in our iOS application, these would be treated as two different entities in different contexts, generating unnecessary duplication. The main limitation is that, in iOS, common classes from each framework are treated as different types. Therefore, a shared data structure cannot be used interchangeably between different frameworks.

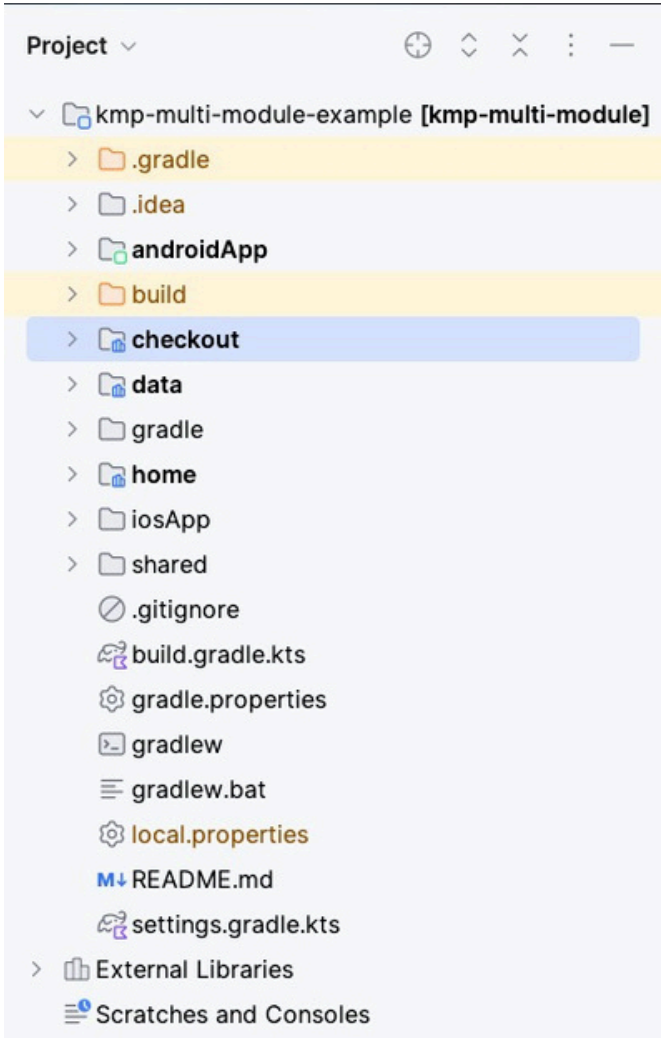
Let's see our example implementation

Let's see how to implement what we discussed above in code. We'll create a KMP project that will include the Home, Checkout, and Data modules, following the structure we've seen in the examples. In this first implementation, we'll separate the code using frameworks in iOS and modules in Android. The interesting part will be observing the behavior of our objects when working with multiple frameworks in iOS.

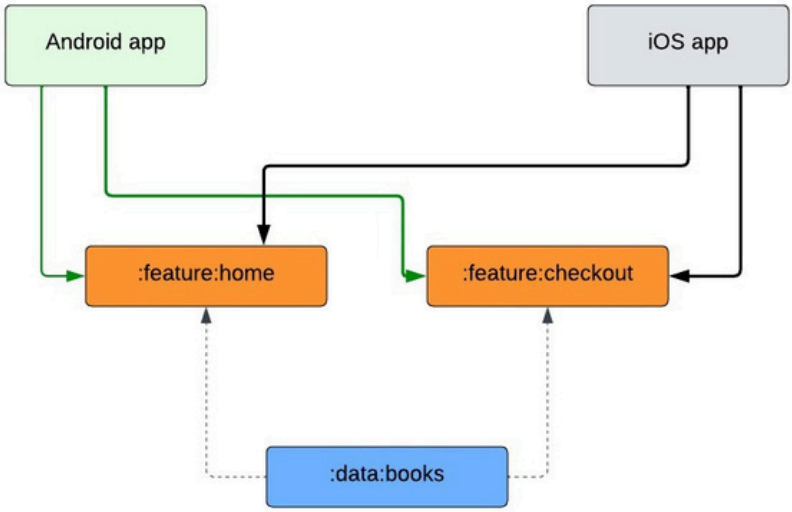
To illustrate this, we'll use the following class diagram:



As we can observe, the ProcessCheckout class from the :checkout module and the GetAll class from the :home module depend on BookRepository, which is located in the :data:book module. The structure of our project would look like this:



As we can observe in the diagram below,



Example of a book selling application.

Android and iOS applications will be responsible for orchestrating the modules (frameworks in iOS) through native code (Kotlin or Swift).

The implementation of multiple modules/frameworks generated from KMP modules can facilitate the gradual adoption of this technology in existing projects. However, it's important to consider the limitations mentioned above if we opt for this approach.

Let's start with Android Our Android application will use the following dependency configuration

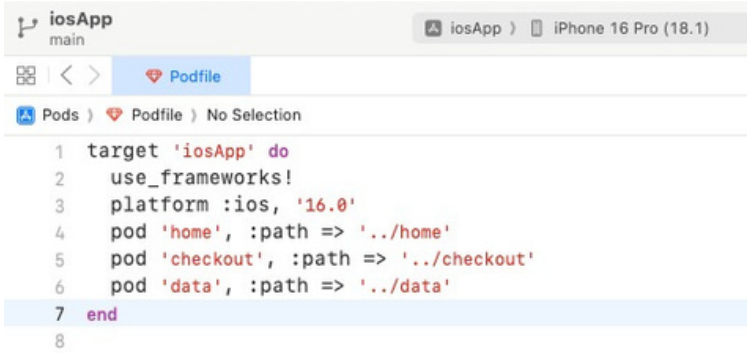
```
dependencies {
  //Checkout
    implementation(projects.checkout)
    //Home
    implementation(projects.home)
    //Data
    implementation(projects.data)

    implementation(libs.compose.ui)
    implementation(libs.compose.ui.tooling.preview)
    implementation(libs.compose.material3)
    implementation(libs.androidx.activity.compose)

    debugImplementation(libs.compose.ui.tooling)
}
```

We will include the modules :checkout, :home and :data in our build.gradle.kts file. Next, we'll see a practical example of using classes between different modules and how the references from the :data module work, which is shared between the feature modules. For this, we'll create a ViewModel that will instantiate the ProcessCheckout and GetAll classes, both sharing a common dependency: BookRepository .

MainViewModel.kt

```
package io.github.santimattius.kmp.android

import io.github.santimattius.kmp.data.Book
import io.github.santimattius.kmp.data.BookRepository
import io.github.santimattius.kmp.checkout.ProcessCheckout
import io.github.santimattius.kmp.home.GetAll

class MainViewModel {
    private val repository = BookRepository()
    private val processCheckout = ProcessCheckout(repository)
    private val getAll = GetAll(repository)

    fun checkout() {
        val books = getAll.invoke()
        val book: Book = books.first()

        processCheckout.invoke(book)
    }
}
```

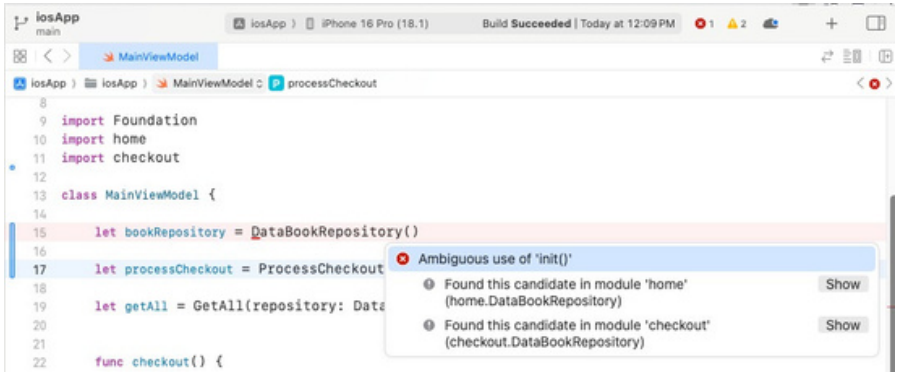
Looking at the imports, we'll notice that BookRepository comes from data and is the same dependency used by both home and checkout modules. The same happens with the book entity, thus avoiding dependency duplication. Now, let's look at the same example in iOS to verify that the behavior is different. Like in Android, let's start by configuring our Podfile with the :home and :checkout modules in our iOS application.



Now let's implement our previous example in Swift.

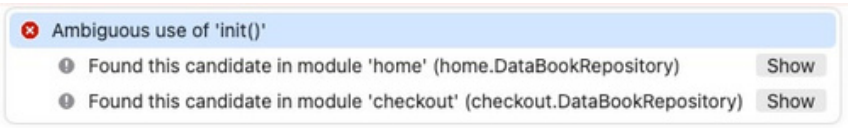
```
import Foundation
import home
import checkout

class MainViewModel {

    let bookRepository = DataBookRepository()
    let processCheckout = ProcessCheckout(repository:
bookRepository)
    let getAll = GetAll(repository: bookRepository)


    func checkout() {
        let books = getAll.invoke()
        let currentBook = books.first

        processCheckout.invoke(book: currentBook)
    }
}
```

Similar to the previous example in Kotlin, we would have a BookRepository instance that we will use in both ProcessCheckout from the checkout framework and GetAll from the home framework. However, let's look at the first issue in the following image.

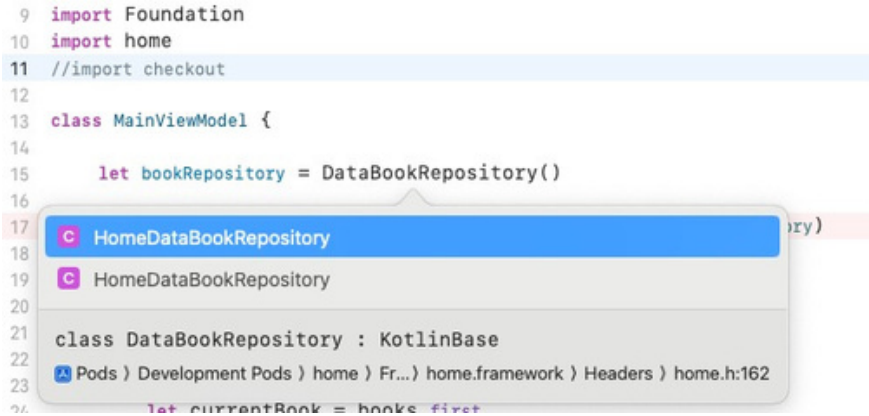


The compiler renames the BookRepository class to DataBookRepository for use in Swift.

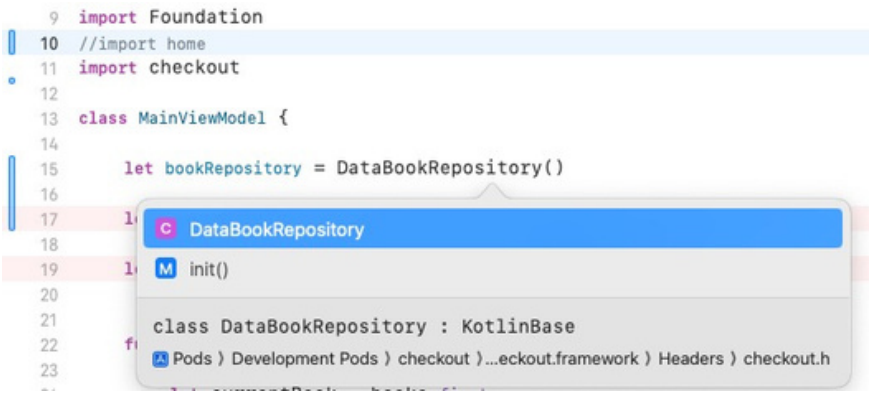
When trying to create an instance of the DataBookRepository class, the compiler shows an "**Ambiguous use of init**" error. This occurs because the compiler cannot determine which constructor to use, as there are two references with the same name, as we'll see in the following images. We'll proceed by removing the line causing the error and continue with the implementation. This will allow us to confirm that the issue arises because the DataBookRepository class is simultaneously defined in both the checkout and home frameworks.



Pods>Development Pods>home>Frameworks>home.framework>Headers>home.h



Pods>Development Pods>checkout>Frameworks>checkout.framework>Headers>checkout.h



thus showing the duplication of classes. The same happens with the Book entity.

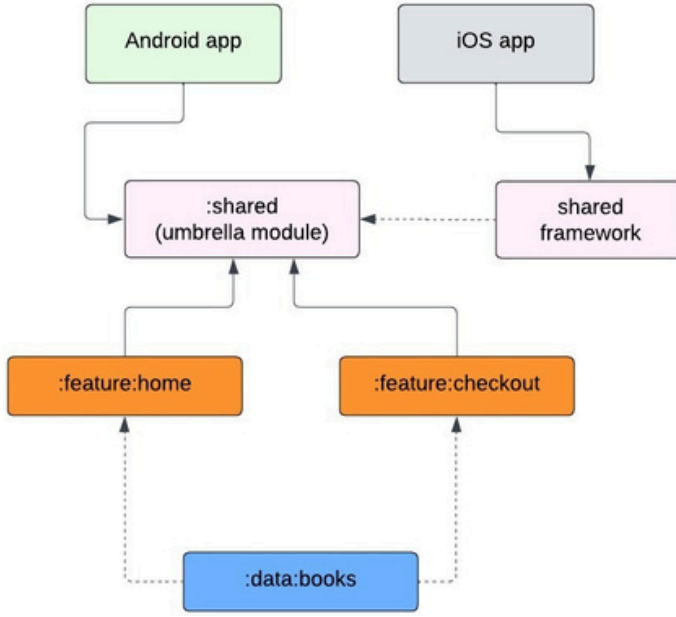
```
9  import Foundation
10 import home
11 import checkout
12
13 class MainViewModel {
14
15     let processCheckout = ProcessCheckout(repository:  DataBookRepository())
16
17     let getAll = GetAll(repository: DataBookRepository())
18
19
20     func checkout() {
21         let books = getAll.invoke()
22         let currentBook = books.first
23
24         processCheckout.invoke(book: currentBook)
25     }
26 }
27
```

Cannot convert value of type 'home.DataBook' to expected argument type 'checkout.DataBook'

Value of optional type 'DataBook?' must be unwrapped to a value of type 'DataBook'

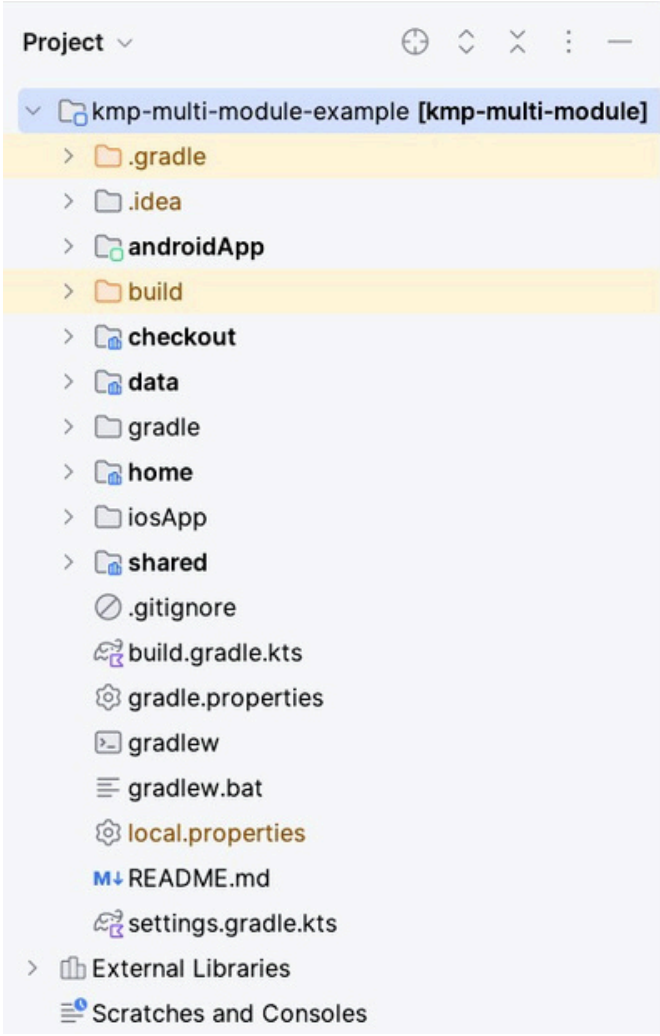
Implementing the Umbrella Module

Let's solve our problem by implementing the Umbrella module. We'll create a :shared module that will encompass our :checkout and :home features, allowing us to include a single framework in iOS.



Definition of the shared module in our project.

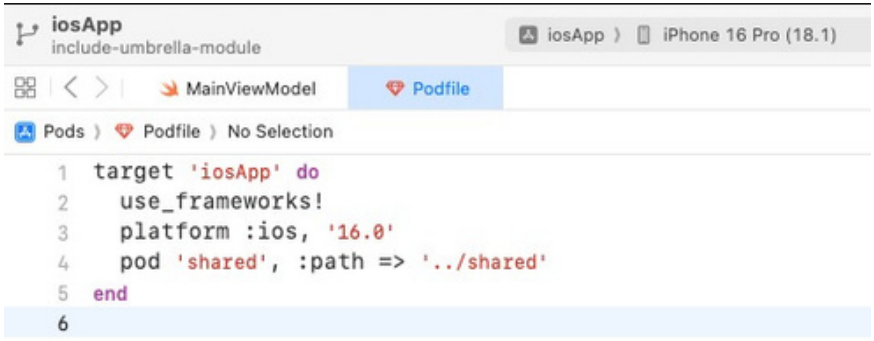
With this modification, the project structure will look like this.



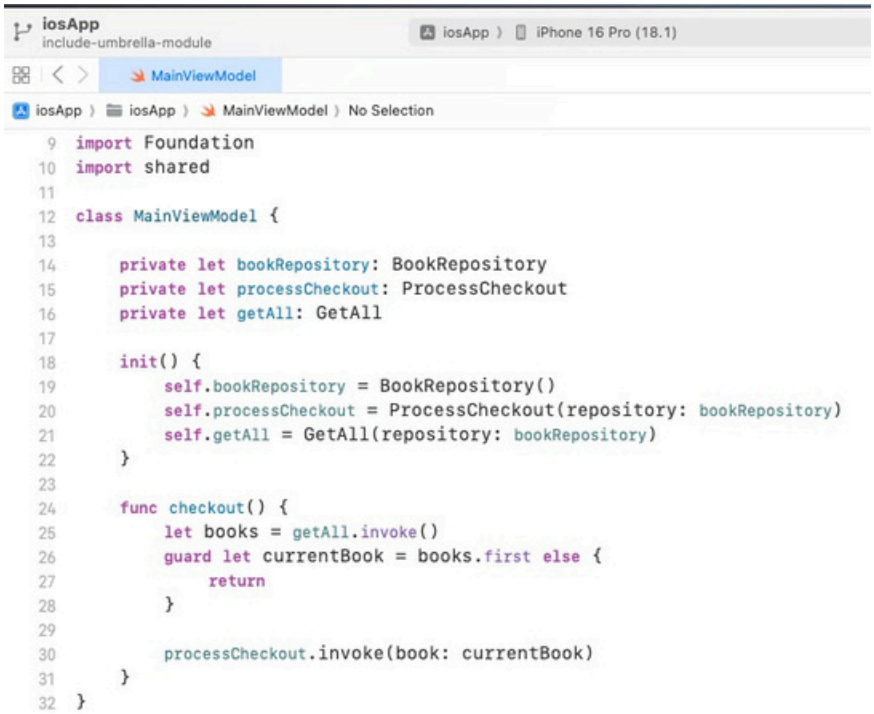
In the 'shared' module, we will define the dependencies mentioned above. For this, we will add the following configuration in our build.gradle.kts file:

```
kotlin {
    //...

    cocoapods {
     summary = "Some description for the Shared Module"
     homepage = "Link to the Shared Module homepage"
     version = "1.0"
     ios.deploymentTarget = "16.0"
     podfile = project.file("../iosApp/Podfile")
     framework {
         baseName = "shared"
         export(project(":home"))
         export(project(":checkout"))
     }
    }

    sourceSets {
        commonMain.dependencies {
            api(project(":home"))
            api(project(":checkout"))
        }
        commonTest.dependencies {
            implementation(libs.kotlin.test)
        }
    }
}
```

In the commonMain dependencies, we include the :home and :checkout modules, along with the definition of our Cocoapods framework. The respective exports allow us to access these dependencies from our iOS code. Next, we'll see how our Podfile is configured with the shared module.



To conclude the example, let's look at the implementation of the code that previously had incompatibilities due to conflicts between the :home and :checkout frameworks.



As we can observe, it is now possible to reuse the BookRepository instance in both ProcessCheckout and GetAll classes, since the type definition is the same regardless of which module it belongs to. This same behavior applies to the Book entity, as evidenced in the checkout function.

Compilation

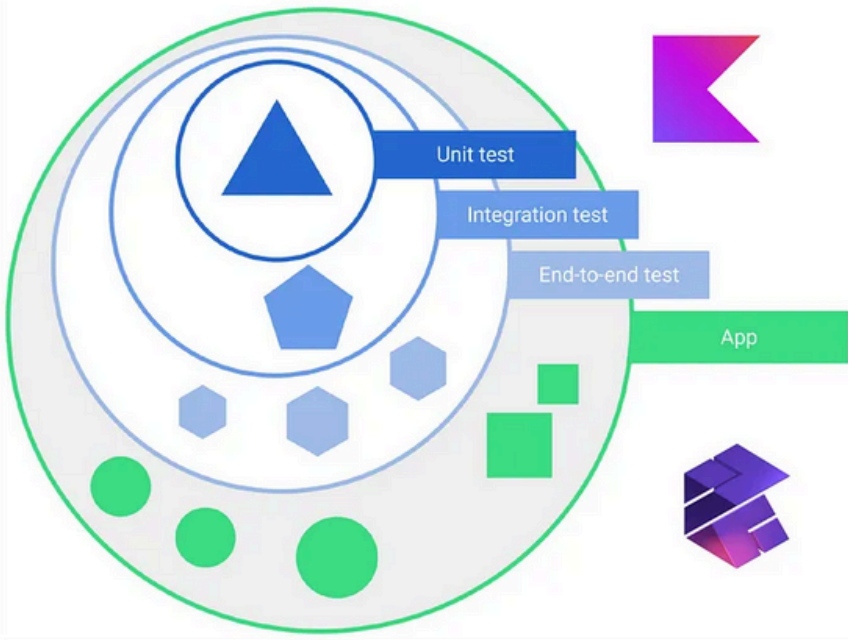
One of the main benefits of modularization is the reduction in compilation times, as unchanged modules can be cached. In theory, this works well, and the Android application effectively builds faster when only some modules have been modified.

However, a challenge arises when building the Kotlin/Native part of KMM, specifically with the gradle tasks linkDebugFrameworkIos and linkReleaseFrameworkIos. These tasks are time-consuming, regardless of the number of modified modules.

Despite this, I've found that it's not always necessary to rebuild the shared module. When making minor changes, it's enough to build the Android application or run iOS tests in the modified module for the changes to be reflected in the iOS application. It's even possible that tests in other modules will also work correctly. While there's no exact formula for this, this approach definitely speeds up the feedback cycle when building the iOS application. However, I believe that implementing automated tests for KMM logic will result in an even faster and more efficient feedback cycle than running Android or iOS applications with each change.

compilarKotlinIosArm64 and **compilarKotlinIosX64** significantly speed up compilation time.

Chapter 5: Testing



Chapter 5: Testing

Kotlin Multiplatform has a fundamental goal: to allow developers to write code once and run it on multiple platforms. However, any error in this shared code can impact all platforms simultaneously.

As Uncle Ben said: "With great power comes great responsibility." For this reason, testing shared code is fundamental.

All software development needs testing to ensure code quality and reliability. In this regard, Kotlin Multiplatform provides various tools and options to perform effective testing across all supported platforms.

Kotlin Multiplatform not only makes it easy to share code between platforms but also allows writing tests that work across all platforms we support.

Benefits of Testing in Kotlin Multiplatform

Testing in Kotlin Multiplatform offers several key benefits:

- **Cross-platform consistency**: Tests run on all supported platforms, ensuring uniform quality across the entire application.
- **Development efficiency**: Writing tests once for all platforms significantly reduces time and effort compared to creating separate tests.
- **Early error detection**: Automated tests identify code issues from the start, allowing corrections before they escalate.
- **Greater confidence in changes**: A robust test suite allows developers to modify code safely, knowing that tests will detect potential issues.

Tools for Testing in Kotlin Multiplatform

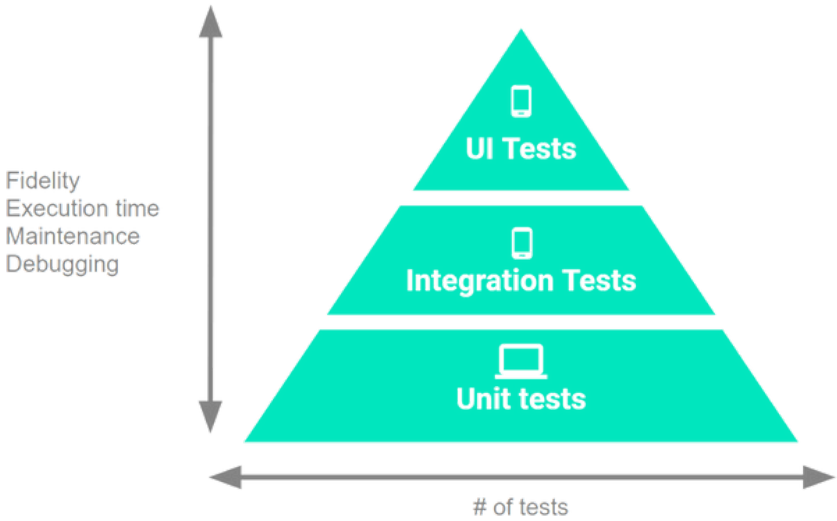
Kotlin Multiplatform provides a range of tools and libraries for testing across all compatible platforms. The following libraries are available for writing tests. You can find references in the kmp-awesome repository.

- **Kotest - test framework**: Powerful, elegant and flexible test framework for Kotlin with additional assertions, property testing and data driven testing
- **Turbine - test library:** A small testing library for kotlinx.coroutines Flow
- **MockingBird - test framework**: A Koltin multiplatform library that provides an easier way to mock and write unit tests for a multiplatform project
- **Mockative - Mocking with KSP**: Mocking for Kotlin/Native and Kotlin Multiplatform using the Kotlin Symbol Processing API (KSP)
- **MocKMP - Mocking with KSP**: A Kotlin/Multiplatform Kotlin Symbol Processor that generates Mocks & Fakes.
- **Mokkery - Mocking library**: Mokkery is a mocking library for Kotlin Multiplatform, easy to use, boilerplate-free and compiler plugin driven. Highly inspired by MockK.
- **KLIP - Snapshot manager for tests**. Kotlin Multiplatform snapshot manager for tests. Automatically generates and asserts against a persistent Any::toString() representation of the object until you explicitly trigger an update. Powered by kotlin compiler plugin to inject relevant keys and paths.
- **Assertk** - Fluent assertions library

While most of these tools are community creations, several have achieved widespread adoption due to their popularity.
In this chapter, we'll explore some of these tools as examples.

Types of Tests

When discussing tests, we generally refer to different types, represented in a pyramid as shown below.



As shown in the pyramid axes, the distribution is based on two fundamental aspects of testing: execution **Speed** and test **Coverage**. The goal of Kotlin Multiplatform is to share business logic across multiple platforms. Since UI tests will depend on platform- specific frameworks, we'll focus on unit and integration tests.

Essential Unit Tests

To follow best practices, implement unit tests for the following components:

- For ViewModels or presenters.
- For the data layer, especially repositories. This layer should be mostly platform-independent, allowing test doubles to replace databases and remote data sources.
- For other platform-independent layers, such as the Domain layer, including use cases and interactions.
- For utility classes, such as string manipulation operations and mathematical calculations.

What do we understand as a unit in our unit tests (Subject Under Test - SUT)?

What do we mean by unit? Let's define this fundamental concept and its principles.

There's a common tendency to establish a one-to-one relationship between tests and classes. However, this practice can result in fragile tests that depend too heavily on specific implementations. The true goal isn't to achieve 100% coverage, but to ensure our tests effectively verify code behavior.

To develop more robust tests, let's consider two essential principles: tests should only change when business specifications change, and code refactoring should not affect the tests.

Integration

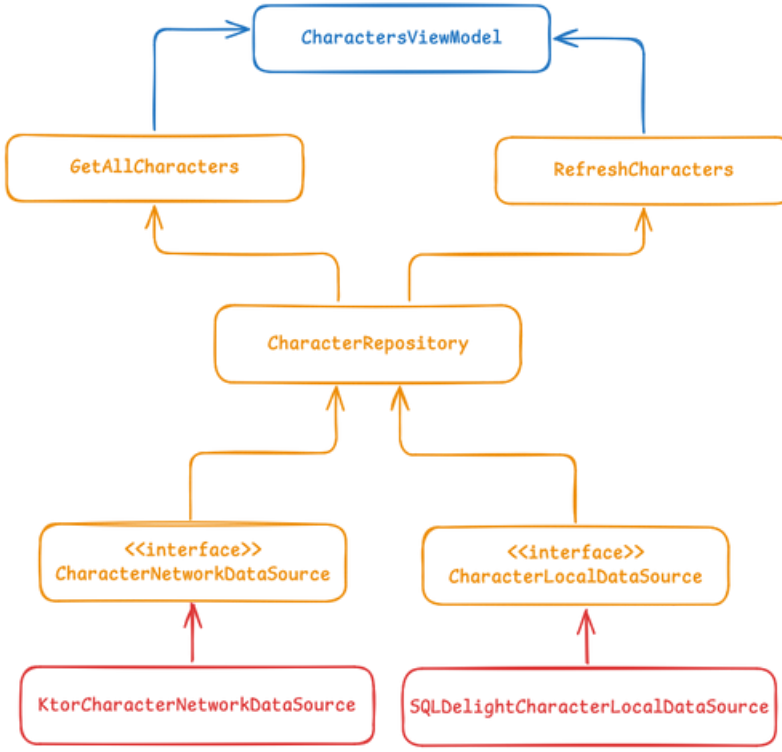
In this type of test, we focus on validating the interaction between our application components in a broader context. This involves more complex scenarios that include interactions with external elements, such as HTTP request libraries and storage systems.

It's important to consider this point when choosing an external library. If it includes testing tools, it will facilitate this type of testing (we'll see this in more detail in the example).

How to identify the scope of our tests

Let's see these concepts in action through a practical example: a simple application that displays a list of Rick and Morty characters.

The application consists of the following components

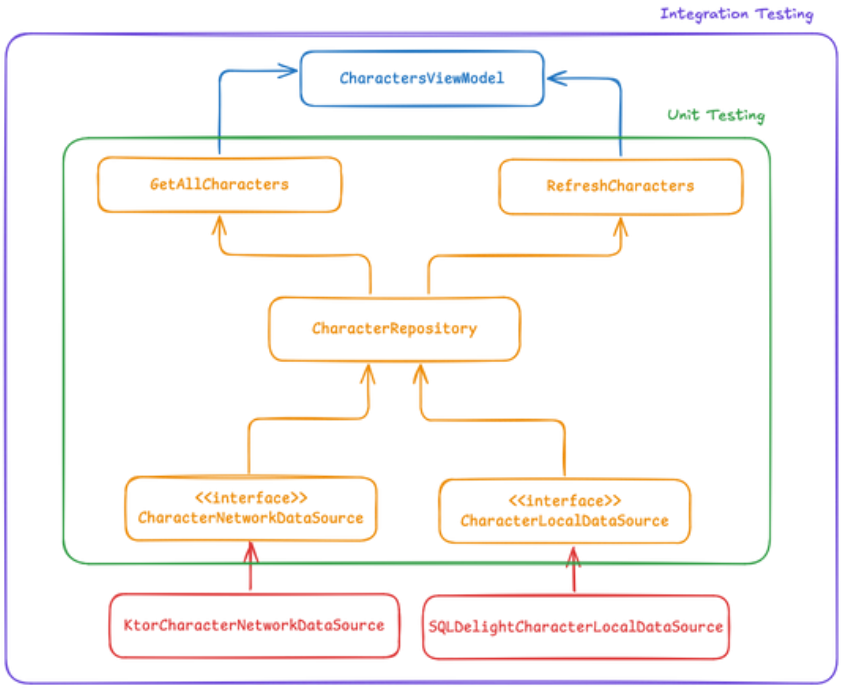


For the implementation, we use a ViewModel that acts as a state container for the views. The application has two main use cases: GetAllCharacters and RefreshCharacters, which manage the information. The CharacterRepository implements the repository pattern and serves as a single source of truth, interacting with two data sources: a local one (CharacterLocalDataSource) and a remote one (CharacterNetworkDataSources). These components are defined as contracts since they depend on concrete infrastructure implementations, such as a database and an HTTP client for server requests.

Now, how do we determine the scope of our tests? Let's define what each type represents:

- **Unit Test**: Evaluates each use case individually, from its logic to the interfaces, without including infrastructure implementations.
- **Integration Test**: Evaluates the interaction (integration) with the infrastructure implementation.

With these concepts clear for our application, let's look at the scope in the following image.



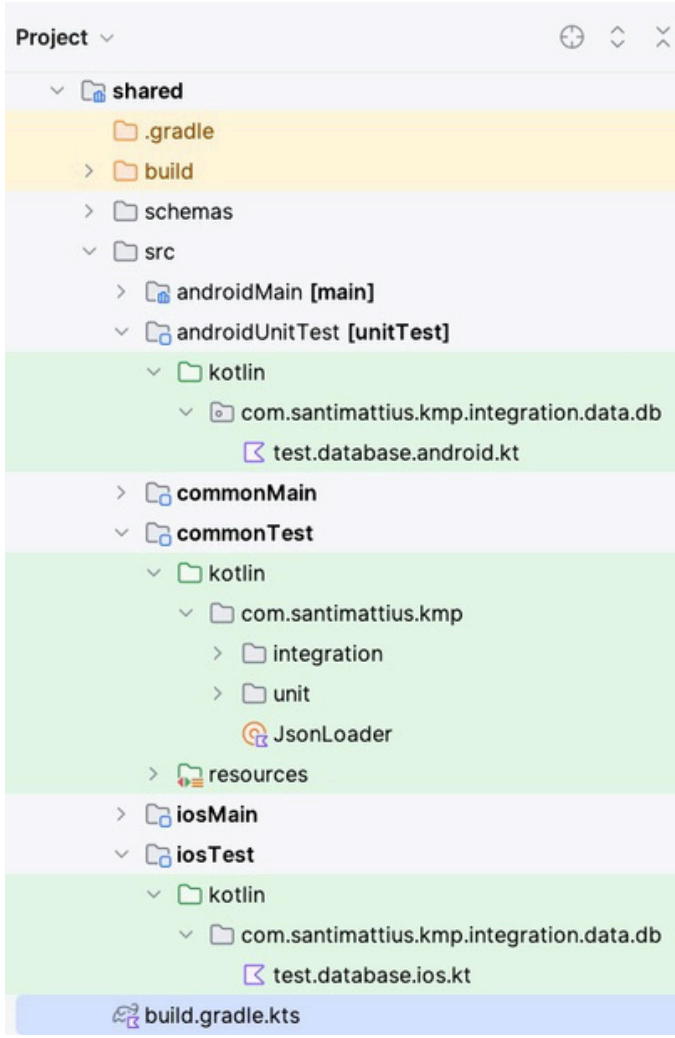
Before implementing the tests in our example, let's see how to configure and run them in our project.

How to Configure and Run our Tests

Just like in shared code, we'll need specific dependencies for testing. For this, we'll configure a dedicated sourceset/target for tests. Next, we'll look at the necessary configuration for Android and iOS.

```
kotlin{
   sourceSets {
        commonTest.dependencies {
            implementation(kotlin("test-common"))
            implementation(kotlin("test-annotations-common"))
            implementation(libs.resource.test)
            implementation(libs.kotlinx.coroutines.test)
            implementation(libs.turbine)
            implementation(libs.kotest.framework.engine)

            implementation(libs.ktor.client.mock)

            implementation(libs.koin.test)
        }

        val androidTest = sourceSets.getByName("androidUnitTest") {
            dependencies {
                implementation(kotlin("test-junit"))
                implementation(libs.junit)
                implementation(libs.sqldelight.jvm)
            }
        }


        iosTest.dependencies {
            //ios testing dependencies
        }

   }

}
```

In most cases, it's not necessary to configure platform-specific dependencies since our main goal is to validate shared code. Once the dependencies are configured in their respective sourcesets, we'll define the directory for tests, as shown in the following image.

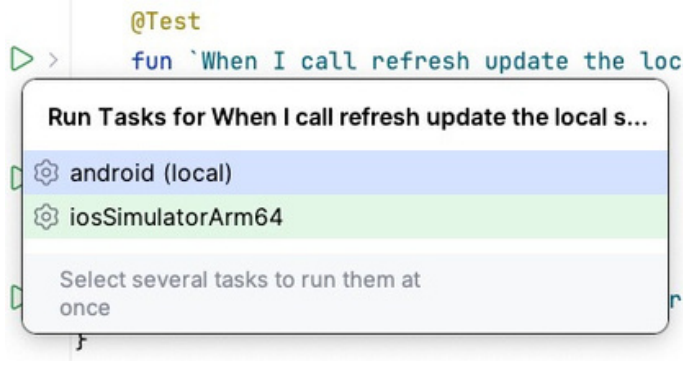


How to Run our Tests

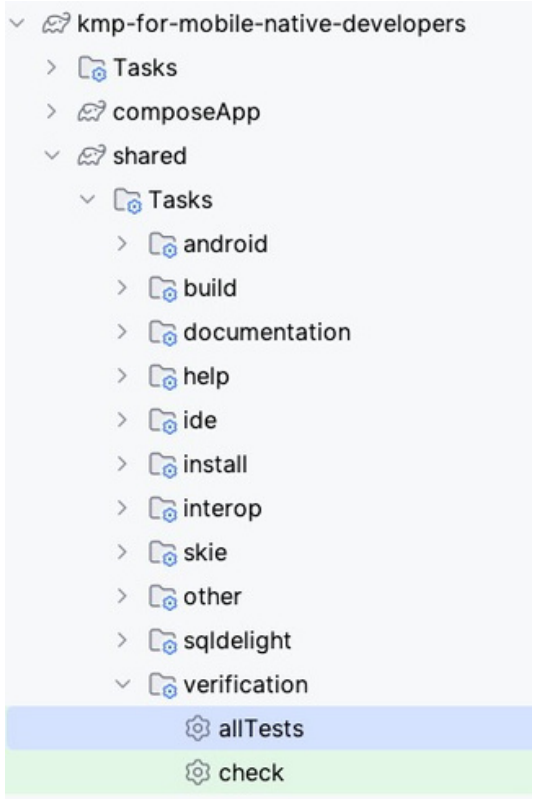
Tests can be executed in two ways: individually or all at once. To run a specific test, the IDE displays a play button next to each function marked with @Test

```kotlin
RefreshCharactersTest.kt
package com.santimattius.kmp.unit.domain.mokkery

import ...

class RefreshCharactersTest {

    private val characters = CharactersResponseMother.characters()
    private val networkDataSource = mock<CharacterNetworkDataSource>()
    private val localDataSource = InMemoryCharacterLocalDataSource()
    private val repository = CharacterRepository(localDataSource, networkDataSource)

    private val refreshCharacters = RefreshCharacters(repository)

    @Test
    fun `When I call refresh update the local storage`() = runTest {...}

    @Test
    fun `When the service returns an empty response`() = runTest {...}

    @Test
    fun `When the service returns an failure response`() = runTest {...}
}
```

when pressing this button, a dropdown menu will appear that allows you to select the platform on which you want to run your tests



Alternatively, we can run all tests using the Gradle verification task, either from the IDE or from the command line.




```
./gradlew :shared:allTests
```

This task will run all our tests across the different platforms configured in the project. Now that we know how to configure and run our tests, let's see how to implement them and what approaches Kotlin Multiplatform offers.

Let's Code

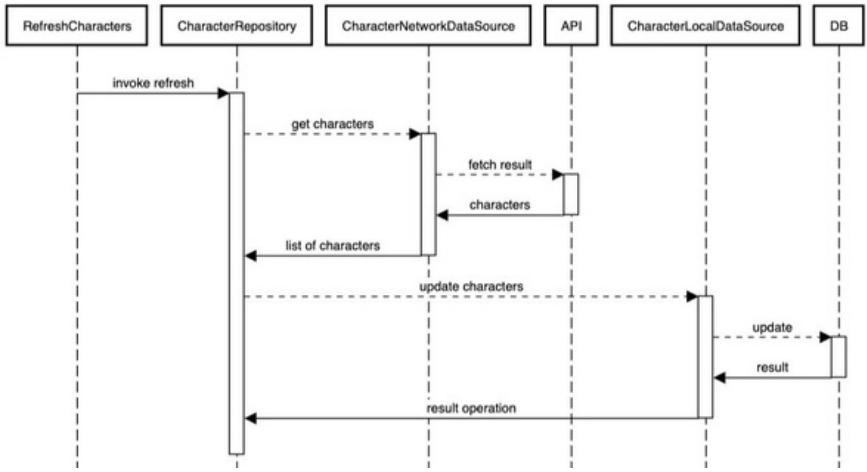
Let's start with unit tests for our use cases. Before we begin, it's important to familiarize ourselves with some key concepts that will help us create efficient unit tests that provide quick feedback.

How to Avoid Slow and Coupled Tests

- **Fakes**: Fakes are alternative implementations that we develop in parallel with the real implementation.
- **Stubs**: Stubs are similar to Fakes, but with a key difference: they contain predefined values that don't need to be passed as arguments when instantiating them.
- **Mocks**: Mocks have an important conceptual difference: besides modeling the response of the element, they allow us to validate its interaction. This means we can verify both behavior and collaboration between classes.

First Unit Test

Let's start with the use case for updating characters in the local data source. The following sequence diagram will help us visualize the update flow in our code.



In our example, we'll need to create test doubles for our data sources (DB and API). First, we'll implement them manually, and then we'll use a mocking library. Let's start with the implementation of CharacterLocalDataSource:

```
class InMemoryCharacterLocalDataSource : CharacterLocalDataSource {
   //....
}
```

An in-memory implementation will store the information only during the test context. And for our CharacterNetworkDataSource we will create a Fake implementation.

```
class FakeCharacterNetworkDataSource : CharacterNetworkDataSource {
    private val jsonLoader = JsonLoader()

    override suspend fun find(id: Long): Result<NetworkCharacter> {
        return all().fold(onSuccess = { characters ->
            runCatching { characters.first { it.id == id } }
        }, onFailure = { Result.failure(it) })
    }

    override suspend fun all(): Result<List<NetworkCharacter>> {
        return runCatching {

jsonLoader.load<CharactersResponse>("characters.json").results
        }
    }
}
```

In this implementation, we use a JsonLoader class that loads a JSON file (which contains a copy of the real API response). This allows us to work with realistic data during testing. To implement this JsonLoader, we use **kotlinx-resources,** a library that will be very useful in our upcoming tests. This tool makes it easy to load files from the local project directory.

```
class JsonLoader {

    private val json = Json {
        ignoreUnknownKeys = true
    }

    fun load(file: String): String {
        val loader = Resource("src/commonTest/resources/${file}")
        return loader.readText()
    }

    internal inline fun <reified R : Any> load(file: String) =
        this.load(file).convertToDataClass<R>()

    internal inline fun <reified R : Any>
String.convertToDataClass(): R {
        return json.decodeFromString<R>(this)
    }

}
```

This avoids having to create complex instances and manually generate data for tests.

> 💡 Tip: To improve the above, we could apply the Object Mother Pattern, which will allow us to have more readable, maintainable, and quickly generated tests.
>
> There are several strategies for managing test instances:
>
> - Traditional
> - Builder Pattern
> - Object Mother
> - Named Arguments

Now that we have our test doubles ready, let's implement the first test to validate the successful case.

```
class RefreshCharactersTest {

    private val networkDataSource = FakeCharacterNetworkDataSource()
    private val localDataSource = InMemoryCharacterLocalDataSource()
    private val repository = CharacterRepository(localDataSource,
networkDataSource)

    private val refreshCharacters = RefreshCharacters(repository)

    @AfterTest
    fun tearDown() {
        localDataSource.clear()
    }

    @Test
    fun `When I call refresh update the local storage`() = runTest {
        // test code
    }
}
```

First, we create the necessary instances for our tests: the object under test and the repository. The **CharacterRepository** and **RefreshCharacters** classes are real implementations, not mocked.

> 💡 Another alternative would have been to create a test double for our repository, but as we saw earlier, we only create test doubles for those components that have external dependencies, such as input/output operations.

Once the instances are generated, we define the test. Here we can use the Given-When-Then pattern to structure the test, invoke the use case, and perform the assertion on the datasource. In this case, since we use Flow, we are using Turbine to interact with flows during testing.

```
class RefreshCharactersTest {

    private val networkDataSource =
FakeCharacterNetworkDataSource()
    private val localDataSource =
InMemoryCharacterLocalDataSource()
    private val repository =
CharacterRepository(localDataSource, networkDataSource)

    private val refreshCharacters =
RefreshCharacters(repository)

    @AfterTest
    fun tearDown() {
        localDataSource.clear()
    }

    @Test
    fun `When I call refresh update the local storage`() =
runTest {
        //Given
        //When
        refreshCharacters.invoke()
        //Then
        localDataSource.all.test {
            assertEquals(true, awaitItem().isNotEmpty())
        }
    }
}
```

But how do we validate the case when the NetworkDataSource returns an empty response? For this, we need to modify our data source. The solution is to implement a stub.

```
class StubCharacterNetworkDataSource(
    private val characters: MutableList<NetworkCharacter> =
mutableListOf()
) : CharacterNetworkDataSource {

    fun setCharacters(characters: List<NetworkCharacter>) {
        this.characters.clear()
        this.characters.addAll(characters)
    } override suspend fun find(id: Long): Result<NetworkCharacter>
    {
        return all().fold(onSuccess = { characters ->
            runCatching { characters.first { it.id == id } }
        }, onFailure = { Result.failure(it) })
    }

    override suspend fun all(): Result<List<NetworkCharacter>> {
        return runCatching { characters }
    }
}
```

And the test would look like this

```
class RefreshCharactersTest {

    private val characters =
JsonLoader.load<CharactersResponse>("characters.json").results
    private val networkDataSource =
StubCharacterNetworkDataSource(characters.toMutableList())
    private val localDataSource = InMemoryCharacterLocalDataSource()
    private val repository = CharacterRepository(localDataSource,
networkDataSource)

    private val refreshCharacters = RefreshCharacters(repository)

    @Test
    fun `When I call refresh update the local storage`() = runTest {
        refreshCharacters.invoke()
        localDataSource.all.test {
            assertEquals(true, awaitItem().isNotEmpty())
        }
    }

    @Test
    fun `When the service returns an empty response`() = runTest {
        networkDataSource.setCharacters(emptyList())
        refreshCharacters.invoke()
        localDataSource.all.test {
            assertEquals(true, awaitItem().isEmpty())
        }
    }
}
```

As we need to generate different test scenarios, it's essential to have more flexible mocks. However, we must remember that this code also requires maintenance. For this reason, it's convenient to use mocking libraries like Mockk or Mockito, which will be familiar to Android developers. While Kotlin Multiplatform doesn't yet have solutions as established as these, they have served as inspiration for the community. In our example, we'll use Mokkery, a library inspired by Mockk according to its documentation.

Mockk is a mocking library implemented entirely in Kotlin that, according to its documentation, offers multiplatform support. However, it still has some issues with native platforms like iOS and macOS.

Let's see how to replace our mocks using **Mokkery**

```
import dev.mokkery.answering.returns
import dev.mokkery.everySuspend
import dev.mokkery.mock

class RefreshCharactersTest {

    private val characters = CharactersResponseMother.characters()
    private val networkDataSource =
mock<CharacterNetworkDataSource>()
    private val localDataSource = InMemoryCharacterLocalDataSource()
    private val repository = CharacterRepository(localDataSource,
networkDataSource)

    private val refreshCharacters = RefreshCharacters(repository)

    @Test
    fun `When I call refresh update the local storage`() = runTest {
      //Given
        everySuspend {
            networkDataSource.all()
        } returns Result.success(characters)
        //When
        refreshCharacters.invoke()
        //Then
        localDataSource.all.test {
            assertEquals(true, awaitItem().isNotEmpty())
        }
    }

    @Test
    fun `When the service returns an empty response`() = runTest {
        //Given
        everySuspend {
            networkDataSource.all()
        } returns Result.success(emptyList())
        //When
        refreshCharacters.invoke()
        //Then
        localDataSource.all.test {
            assertEquals(true, awaitItem().isEmpty())
        }
    }
}
```

With Mokkery, we can create a mock of the CharacterNetworkDataSource interface in the following way:

```
  private val networkDataSource = mock<CharacterNetworkDataSource>()
```

and configure the behavior of its methods

```
everySuspend {
    networkDataSource.all()
} returns Result.success(characters)
```

With a mocking library, we can also easily perform other types of verifications, such as checking that the all method of networkDataSource was called exactly once.

```
    @Test
    fun `When I call refresh update the local storage`() =
runTest {
        //Given
        everySuspend {
            networkDataSource.all()
        } returns Result.success(characters)
        //When
        refreshCharacters.invoke()
        //Then
        verifySuspend(mode = exactly(1)) {
            networkDataSource.all()
        }
        localDataSource.all.test {
            assertEquals(true, awaitItem().isNotEmpty())
        }
    }
```

Thanks to Mokkery, we can now test the remaining cases more efficiently and clearly.

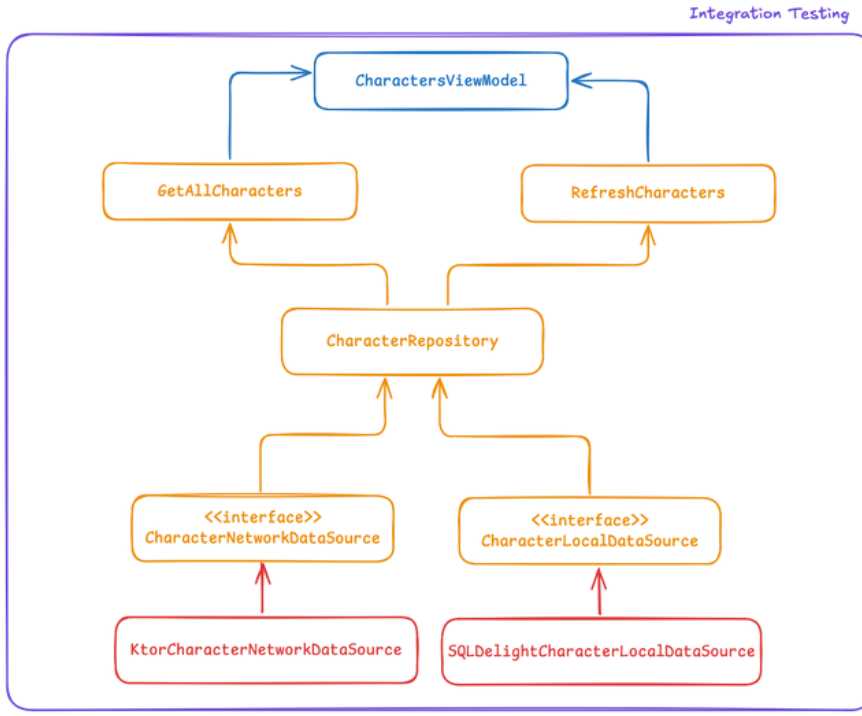
Setup | Mokkery
How to add Mokkery to your Gradle project rapidly!

https://mokkery.dev/docs/Setup

It is recommended to generate mocks only for interfaces that are coupled to external data sources, as we saw in the previous example.

Integration Tests

As shown in the initial diagram, we will validate the integration of components and their behavior.



Integration tests allow us to validate the infrastructure, that is, the external libraries we use to manage our data. In our case, we use Ktor for HTTP requests and SQLDelight for local storage. Both libraries provide specific tools for testing.

Testing with Ktor

Ktor provides an Engine that allows us to create simulations of our services. The implementation is simple: we just need to define an Engine of type MockEngine and incorporate it into our client configuration.

```
val mockEngine = MockEngine { request ->
    respond(
        content = ByteReadChannel("""{"ip":"127.0.0.1"}"""),
        status = HttpStatusCode.OK,
        headers = headersOf(HttpHeaders.ContentType,
"application/json")
    )
}
```

To facilitate testing with Ktor, we will create some simple abstractions that will allow us to reuse configurations across all our tests. We will implement the following code to configure the Mock engine for testing.

```
fun testKtorClient(mockClient: MockClient = MockClient()):
HttpClient {
    val engine = testKtorEngine(mockClient)
    return HttpClient(engine) {
        install(ContentNegotiation) {
            json(Json {
                prettyPrint = true
                isLenient = true
                ignoreUnknownKeys = true
            })
        }
    }
}

private fun testKtorEngine(interceptor: ResponseInterceptor)
= MockEngine { request ->
    val response = interceptor(request)
    respond(
        content = ByteReadChannel(response.content),
        status = response.status,
        headers = headersOf(HttpHeaders.ContentType,
"application/json")
    )
}
```

You can find the complete code in the file MockClient.kt.
Now let's see how to implement our integration tests

```
class RefreshCharactersIntegrationTest {

    private val jsonResponse = JsonLoader.load("characters.json")
    //KtorClient setup
    private val mockClient = MockClient()
    private val ktorClient = testKtorClient(mockClient)
    private val networkDataSource =
KtorCharacterNetworkDataSource(ktorClient)

    private val localDataSource = InMemoryCharacterLocalDataSource()
    private val repository = CharacterRepository(localDataSource,
networkDataSource)

    private val refreshCharacters = RefreshCharacters(repository)

    @AfterTest
    fun tearDown() {
        localDataSource.clear()
    }

    @Test
    fun `When I call refresh update the local storage`() = runTest {
        //Given
        val response = DefaultMockResponse(jsonResponse,
HttpStatusCode.OK)
        mockClient.setResponse(response)
        //When
        refreshCharacters.invoke()
        //Then
        localDataSource.all.test {
            assertEquals(true, awaitItem().isNotEmpty())
        }
    }
```

In the test, we will create an instance of KtorCharacterNetworkDataSource, which concretely implements our CharacterNetworkDataSource interface. This time we will initialize it with a special HttpClient for testing that uses MockEngine.

Let's apply the same approach to our CharacterLocalDataSource.

Testing | Ktor
Ktor provides a MockEngine that simulates HTTP calls without connecting to the endpoint.
https://ktor.io/docs/client-testing.html

Testing with SQLDelight

SQLDelight can be used for testing, but it requires platform-specific configuration. In its implementation, we need to define an appropriate driver for each platform.

```
// database.common.kt expect class DriverFactory {

    fun createDriver(): SqlDriver
}

fun createDatabase(driver: SqlDriver): CharactersDatabase {
    return CharactersDatabase(driver)
}

// database.android.kt
actual class DriverFactory(private val context: Context) {
    actual fun createDriver(): SqlDriver {
        return AndroidSqliteDriver(CharactersDatabase.Schema,
context, "app_database.db")
    }
}

//database.ios.kt
actual class DriverFactory {
    actual fun createDriver(): SqlDriver {
        return NativeSqliteDriver(CharactersDatabase.Schema,
"app_database.db")
    }
}
```

In our case, we have a DriverFactory class implemented in both Android and iOS, each with its specific drivers. For testing, we follow the same principle, but apply it to the source code sets intended for tests.

```
//test.database.common.kt
expect fun testDbDriver(): SqlDriver

//test.database.android.kt
actual fun testDbDriver(): SqlDriver {
    return JdbcSqliteDriver(JdbcSqliteDriver.IN_MEMORY)
        .also {
            CharactersDatabase.Schema.create(it)
        }
}
//test.database.ios.kt
actual fun testDbDriver(): SqlDriver {
    return inMemoryDriver(CharactersDatabase.Schema)
}
```

As we can observe, this library uses a similar concept to what we implemented in our InMemoryCharacterLocalDataSource, which is an in-memory implementation.

Let's implement this change in our test.

```
class RefreshCharactersIntegrationTest {

    private val jsonResponse = JsonLoader.load("characters.json")

    //KtorClient setup
    private val mockClient = MockClient()
    private val ktorClient = testKtorClient(mockClient)
    private val networkDataSource =
KtorCharacterNetworkDataSource(ktorClient)

    //SQLDelight setup
    private val db = createDatabase(driver = testDbDriver())
    private val localDataSource =
SQLDelightCharacterLocalDataSource(db)
    private val repository = CharacterRepository(localDataSource,
networkDataSource)

    private val refreshCharacters = RefreshCharacters(repository)

    @Test
    fun `When I call refresh update the local storage`() = runTest {
        //Given
        val response = DefaultMockResponse(jsonResponse,
HttpStatusCode.OK)
        mockClient.setResponse(response)
        //When
        refreshCharacters.invoke()
        //Then
        localDataSource.all.test {
            assertEquals(true, awaitItem().isNotEmpty())
        }
    }

    @Test
    fun `When the service returns an empty response`() = runTest {
        //Given
        val response = DefaultMockResponse("{}", HttpStatusCode.OK)
        mockClient.setResponse(response)
        //When
        refreshCharacters.invoke()

        localDataSource.all.test {
            assertEquals(true, awaitItem().isEmpty())
        }
    }
}
```

Similar to Ktor, we use instances of our local storage implementation, SQLDelightCharacterLocalDataSource.

> Testing: SQLDelight
> SQLDelight - Generates typesafe Kotlin APIs from SQL
> https://cashapp.github.io/sqldelight/2.0.0/android_sqlite/testing/

So far, we have validated all components of the use case through our tests. Now, let's see how we can improve the code in our project and measure its coverage.

Validating our Dependency Injection

In integration tests, we make minimal adjustments to external library configurations to adapt them to testing needs. However, this process can become repetitive for each use case. This is where dependency injection and Koin help us optimize these configurations. The first step is to configure our test dependencies with Koin.

```
val testPlatformModule: Module = module {
    single<SqlDriver> { testDbDriver() }
    single<MockClient> { MockClient() }
    single<HttpClient> { testKtorClient(get()) }
}
```

After defining our dependencies, we configure the test to use Koin.

```
class RefreshCharactersIntegrationTest : KoinTest {

    private val jsonResponse =
JsonLoader.load("characters.json")

    //KtorClient setup
    private val mockClient: MockClient by inject()

    @BeforeTest
    fun setUp() {
        startKoin {
            modules(
                testPlatformModule,
                sharedModule
            )
        }
    }

    @AfterTest
    fun tearDown() {
        stopKoin()
    }

    @Test
    fun `When I call refresh update the local
storage`() = runTest {
        //.....
    }
}
```

and in our test we request an instance of the object under test, in this case RefreshCharacters

```
  @Test
    fun `When I call refresh update the local
storage`() = runTest {
        //Given
        val useCase = get<RefreshCharacters>() //from
koin
        val localDataSource =
get<CharacterLocalDataSource>()
        val response = MockResponse.ok(jsonResponse)
        mockClient.setResponse(response)
        //When
        useCase.invoke()
        //Then
        localDataSource.all.test {
        assertEquals(true,
awaitItem().isNotEmpty())
        }
    }
```

Coverage Metrics

Test coverage metrics indicate what percentage of code has been tested, being fundamental to evaluate test quality and identify areas without coverage. While these metrics don't guarantee the complete absence of errors, tools like Jacoco and Slather allow us to calculate them and integrate them into the development cycle. In Kotlin Multiplatform, we'll use Kover, a Gradle plugin similar to Jacoco.

Kover

Kover is a toolset designed to measure test coverage of Kotlin code compiled for JVM and Android platforms. Its main component is a Gradle plugin that we'll explore next.

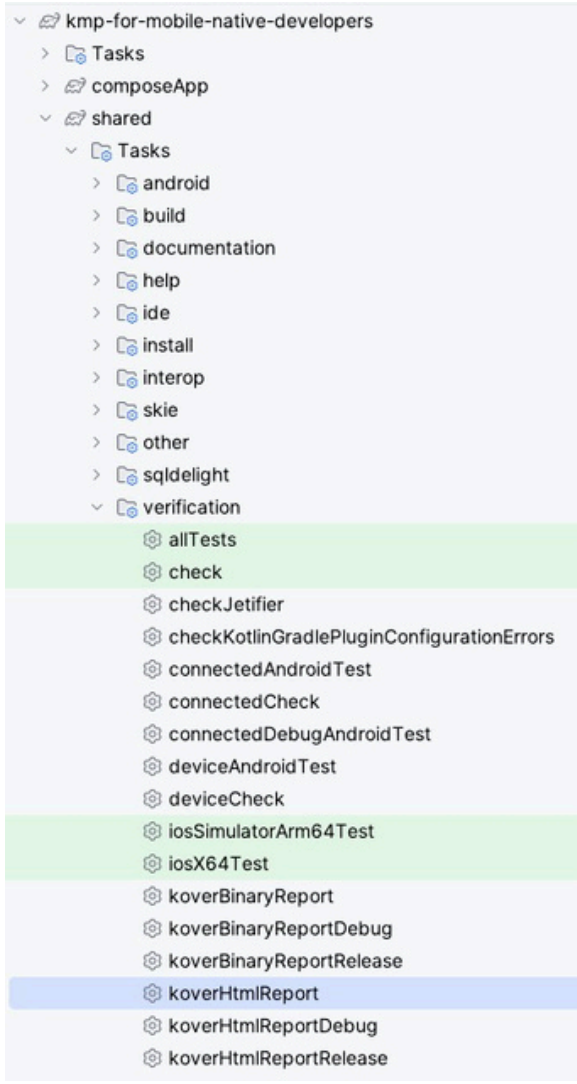
Kover Features

- Code coverage measurement through JVM tests (important: no support yet for JS and native targets).
- Report generation in HTML and XML formats.
- Compatibility with Kotlin JVM and Kotlin Multiplatform projects.
- Support for Kotlin Android projects with build variants (note: instrumentation tests on Android devices are not yet supported).
- Compatibility with mixed Kotlin and Java code.
- Configuration of verification rules with coverage thresholds in the Gradle plugin.
- Integration with JaCoCo library as an alternative for measuring coverage and generating reports.

To implement Kover in our project, we only need to add its Gradle plugin:

```
plugins {
     id("org.jetbrains.kotlinx.kover") version "0.7.6"
}
```

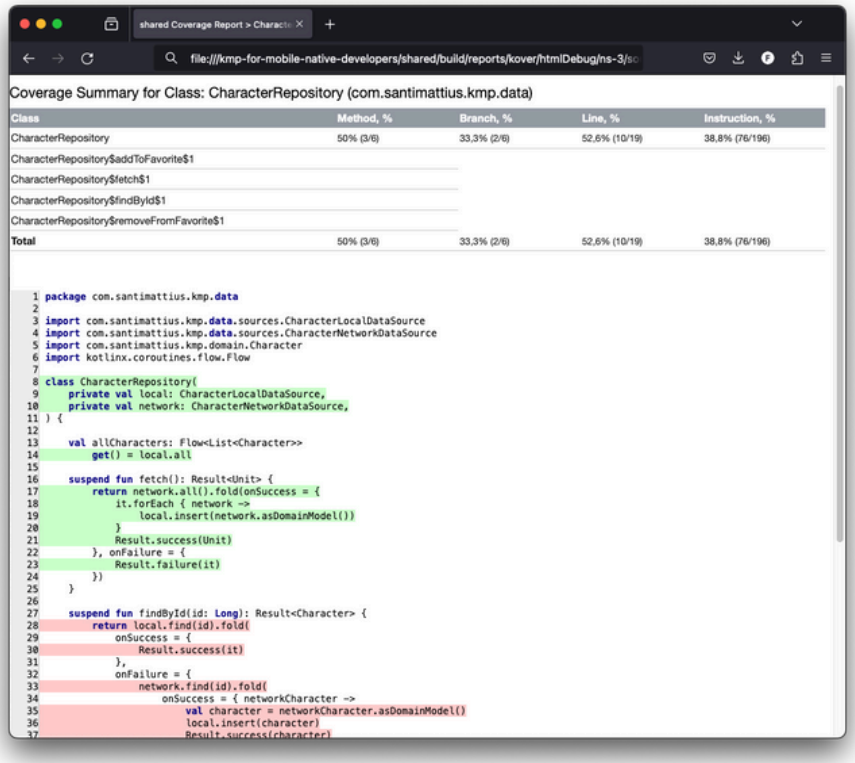
Once the plugin is added, we'll be able to run the Kover tasks available in Gradle.



In this case we will execute

```
./gradlew :shared:koverHtmlReport
```

To generate the HTML report shown below



You can configure coverage limits in your projects using Kover by defining custom rules. For example:

```
koverReport {
    verify {
        rule("Basic Line Coverage") {
            isEnabled = true
            bound {
                minValue = 80 // Minimum coverage percentage
                maxValue = 100 // Maximum coverage
percentage (optional)
                metric = MetricType.LINE
                aggregation =
AggregationType.COVERED_PERCENTAGE
            }
        }

        rule("Branch Coverage") {
            isEnabled = true
            bound {
                minValue = 70 // Minimum coverage percentage
for branches
                metric = MetricType.BRANCH
            }
        }
    }
}
```

Although Kover is in Alpha version and does not yet support Kotlin Native, it is useful for validating our shared code.

Best Practices for Testing in Kotlin Multiplatform

To ensure effective testing in Kotlin Multiplatform, we can follow these software development best practices:

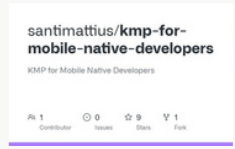
- **Write tests from the start**: Beginning with tests early in development helps detect problems early and builds a solid testing foundation.
- **Automate testing**: Automation ensures consistent execution and minimizes human error.
- **Use parameterized tests**: These tests allow evaluation of multiple data sets with a single test case, improving maintainability. For this we can use Kotest.
- **Separate tests from implementations**: Keeping test code separate from production code improves organization and facilitates future changes.

Rules for Using Tests in Multiplatform Projects

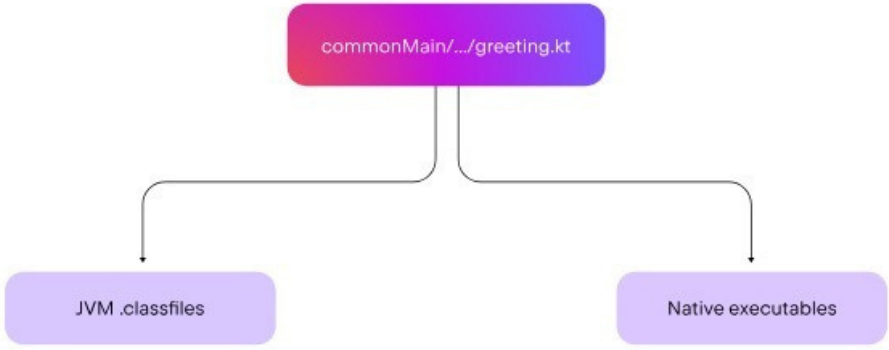
When implementing tests in Kotlin Multiplatform applications, consider these important guidelines:

- For common code, use only multiplatform libraries like kotlin.test. Add dependencies to the commonTest set.
- The Asserter type from kotlin.test should be used indirectly. Although the instance is visible, avoid using it directly in tests.
- Stick to the testing library API. The compiler and IDE will help you avoid framework-specific features.
- When using commonTest, run your tests with each planned framework to verify correct environment setup.
- For platform-specific code, take advantage of the framework's native features, including annotations and extensions.
- Tests can be run from the IDE or through Gradle tasks.
- Test execution automatically generates HTML reports.

GitHub - santimattius/kmp-for-mobile-native-developers at unit_and_integration_testing
KMP for Mobile Native Developers



Chapter 6: Using Native Libraries in Kotlin Multiplatform



Chapter 6: Using Native Libraries in Kotlin Multiplatform

The adoption of Kotlin Multiplatform represents a strategic step towards more coherent cross-platform development. This technology allows sharing code and business logic, reducing work duplication and improving consistency between applications. When starting projects with KMP for Android and iOS, a practical approach is to adapt and integrate existing native solutions into the KMP module, rather than rewriting everything from scratch. This strategy allows us to leverage both KMP functionalities and platform-specific code. In this section, we'll explore how to extend the Bugsnag SDK for use from KMP modules, both in Android and iOS. We'll start with the integration of existing native SDKs, focusing on avoiding unnecessary rewrites.

How do we include Android or iOS specific code in a KMP module?

Using Android Dependencies in KMP

To add Android-specific dependencies to a Kotlin Multiplatform module, the process is identical to traditional Android projects. We just need to add the dependency in the Android source set within the build.gradle(.kts) file in the shared directory. In this example, we'll implement Bugsnag and Android Startup, two platform-exclusive dependencies for Android.

```
kotlin {
  sourcesets{
      commonMain.dependencies{
        // common dependencies
      }
       androidMain.dependencies {
            api(libs.bugsnag.android)

implementation(libs.androidx.startup.runtime)
        }
  }
}
```

Once these dependencies are configured, we can use them within the Android sourceset.
Next, we'll explore the configuration of iOS-specific dependencies. Then we'll return to the implementation after having the dependencies configured on both platforms.

How to Use iOS Dependencies in KMP

Apple SDK dependencies like Foundation or Core Bluetooth are precompiled in Kotlin Multiplatform projects and require no additional configuration. You can reuse libraries and frameworks from the iOS ecosystem in your iOS sourcesets. Kotlin is compatible with Objective-C and Swift dependencies, as long as they expose their APIs to Objective-C using the @objc attribute. However, Swift-only dependencies are not yet supported. CocoaPods integration has the same limitation: it doesn't support Swift-only pods. To manage iOS dependencies in Kotlin Multiplatform projects, we recommend using CocoaPods. You should only manage dependencies manually if you need to customize the interoperability process or have a specific reason to do so. In our case, we'll use CocoaPods. To begin, we need to configure the CocoaPods plugin in our KMP project:

```
[versions]
kotlin = "your-kotlin-version"

[plugins]
cocoaPods = { id =
"org.jetbrains.kotlin.native.cocoapods", version.ref =
"kotlin" }
```

Next, we'll apply the CocoaPods plugin in both the root project and the shared module.

```
//root build.gradle.kts
plugins {
    alias(libs.plugins.androidApplication) apply false
    alias(libs.plugins.androidLibrary) apply false
    alias(libs.plugins.kotlinMultiplatform) apply false
    alias(libs.plugins.jetbrainsKotlinAndroid) apply
false
    alias(libs.plugins.cocoaPods) apply false
}

//shared module build.gradle.kts
plugins {
    alias(libs.plugins.kotlinMultiplatform)
    alias(libs.plugins.cocoaPods)
    alias(libs.plugins.androidLibrary)
}
```

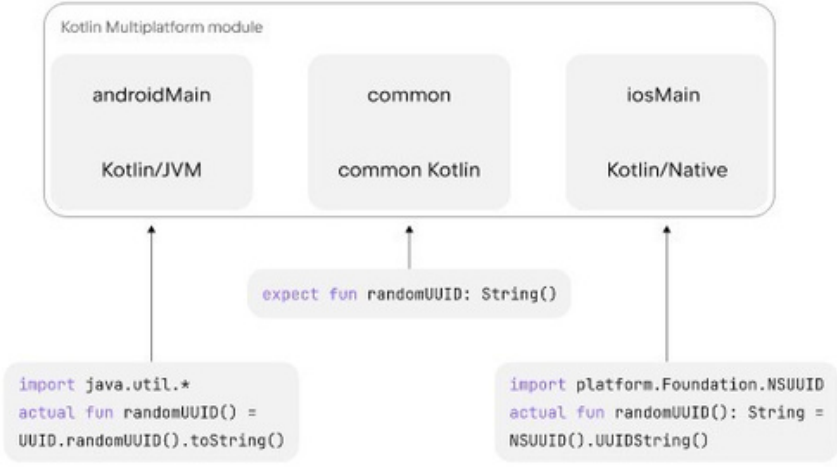
With the CocoaPods plugin installed, we can configure our shared module as a pod and define the necessary dependencies. For this example, we'll use Bugsnag as a native library.

```
kotlin{
cocoapods {
        version = "1.0"
        summary = "Some description for a Kotlin/Native
module"
        homepage = "Link to a Kotlin/Native module homepage"
        name = "Shared"
        ios.deploymentTarget = "14.0"

        framework {
            baseName = "Shared"
            isStatic = false
        }

        pod("Bugsnag"){
            version = "6.28.0"
        }
    }
}
```

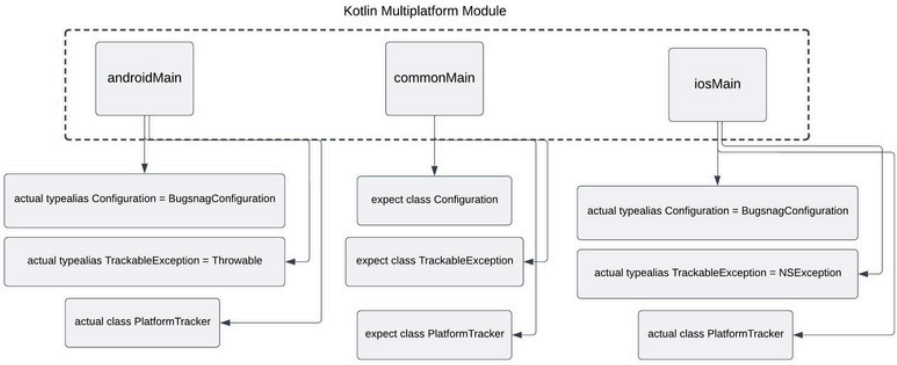
Now that we have configured the dependencies for both platforms, let's see how to reuse these native APIs through Kotlin Multiplatform.

Expect/Actual

Kotlin provides an elegant mechanism to access platform-specific APIs while developing common logic: expect and actual declarations. The mechanism is simple: the common source set of a multiplatform module defines an expect declaration, and each platform source set provides its corresponding actual declaration. The compiler verifies that each declaration marked with the expect keyword in the common sources has its corresponding declaration marked with actual in all target platforms. This system works with most Kotlin declarations: functions, classes, interfaces, enums, properties, and annotations. In this section, we'll focus on using expect/actual functions and properties.



Now let's look at the practical application of this concept using the Bugsnag API as an example.



The diagram above shows the general concept of our implementation. Let's now look at the code in detail. In commonMain, we'll define the expect declarations for our API functions, ensuring consistency across platforms. Bugsnag represents an ideal case, as it maintains consistent nomenclature in its APIs for both Android and iOS, from method signatures to entity structures.

```
// bugsnag.common.kt
package com.santimattius.kmp.playground

//SDK configurations
expect class Configuration
// Information track
expect class TrackableException

object Bugsnag {

    private val provider: PlatformTracker = PlatformTracker()

    fun initialize(config: Configuration) {
        provider.initialize(config)
    }

    fun track(exception: TrackableException) {
        provider.track(exception)
    }
}

internal expect class PlatformTracker(){
    fun initialize(config: Configuration)
    fun track(exception: TrackableException)
}
```

Let's look at the Android implementation.

In the androidMain sourceset, we'll implement concrete versions of the classes defined in the shared code. For the entities Configuration and TrackableException, we'll use typealias as a specific solution. Let's look at this implementation in detail.

```
package com.santimattius.kmp.playground

import com.bugsnag.android.Bugsnag
import com.bugsnag.android.Configuration as
BugsnagConfiguration

actual typealias Configuration = BugsnagConfiguration
actual typealias TrackableException = Throwable

internal actual class PlatformTracker {
    actual fun initialize(config: Configuration) {
        val context = applicationContext ?: run {
            // TODO: add logging later
            return
        }
        Bugsnag.start(context, config)
    }

    actual fun track(exception: TrackableException) {
        Bugsnag.notify(exception)
    }
}
```

As we can observe in the imports, the typealias acts as direct references to the native API entities.

```
import com.bugsnag.android.Bugsnag
import com.bugsnag.android.Configuration as BugsnagConfiguration
```

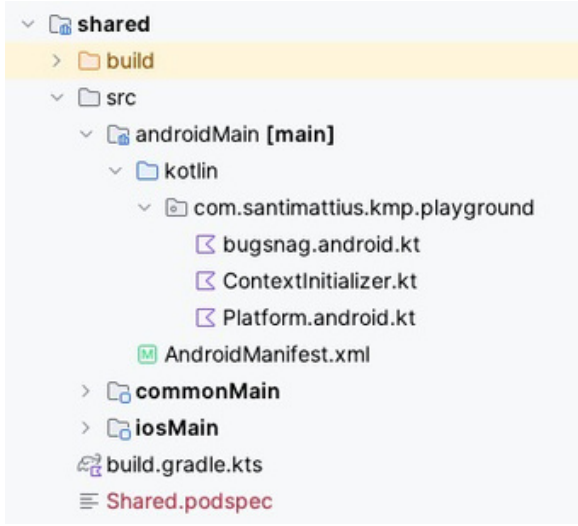
This is where we directly use Android dependencies.

How can we get the Android Context in KMP?

Among the Android-specific dependencies we defined earlier, we find Android Startup. Kotlin Multiplatform's flexibility to implement platform-specific code allows us to apply an elegant solution: Bugsnag needs Android's applicationContext to work during application startup. Through Android Startup, we can obtain this context and, if desired, initialize our library. The solution involves implementing an Android Startup Initializer to capture the applicationContext.

```
int ernal var applicationContext: Context? = null
    private set

class ContextInitializer: Initializer<Unit> {
    override fun create(context: Context) {
        applicationContext = context.applicationContext
    }

    override fun dependencies(): List<Class<out
Initializer<*>>> {
        return emptyList()
    }
}
```

After defining the initializer, we need to register it in the AndroidManifest. To do this, we'll first create this file in our androidMain directory.



In the AndroidManifest.xml file, we need to add the following configuration:

```
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools">

    <application>
        <provider
            android:name="androidx.startup.InitializationProvider"
            android:authorities="${applicationId}.androidx-startup"
            android:exported="false"
            tools:node="merge">
            <meta-data

android:name="com.santimattius.kmp.playground.ContextInitializer"
                android:value="androidx.startup" />
        </provider>
    </application>

</manifest>
```

To use Bugsnag in Android, we just need to reference our Bugsnag object from the application. Let's see an example:

```
import android.app.Application
import com.santimattius.kmp.playground.Bugsnag
import com.santimattius.kmp.playground.Configuration

class MainApplication : Application() {

    override fun onCreate() {
        super.onCreate()
        // Initialization
        Bugsnag.initialize(config = Configuration.load(this))
        // Send test exception
        Bugsnag.track(exception = Throwable(message = "This is
a test!!"))
    }
}
```

Thanks to the alias definitions, we can use native APIs directly from our Kotlin Multiplatform module, as shown in our import declarations.

Next, we'll look at the iOS implementation.

```
package com.santimattius.kmp.playground

import cocoapods.Bugsnag.Bugsnag
import cocoapods.Bugsnag.BugsnagConfiguration
import kotlinx.cinterop.ExperimentalForeignApi
import platform.Foundation.NSException

@OptIn(ExperimentalForeignApi::class)
actual typealias Configuration = BugsnagConfiguration
actual typealias TrackableException = NSException

@OptIn(ExperimentalForeignApi::class)
internal actual class PlatformTracker {
     actual fun initialize(config: Configuration) {
         Bugsnag.startWithConfiguration(config)
    }

    actual fun track(exception: TrackableException) {
        Bugsnag.notify(exception)
    }
}
```

Looking at the imports, we can see that our definitions use native APIs provided by both the iOS platform and Bugsnag.

```
import cocoapods.Bugsnag.Bugsnag
import cocoapods.Bugsnag.BugsnagConfiguration
import kotlinx.cinterop.ExperimentalForeignApi
import platform.Foundation.NSException
```

```
import SwiftUI
import Shared
import Bugsnag

@main
struct iOSApp: App {

    init() {
        // Initialization
         let config = BugsnagConfiguration.loadConfig()
         config.appVersion = "1.0.0-alpha"

         Bugsnag.shared.initialize(config: config)
         // Send test exception
         let exception =
NSException(name:NSExceptionName(rawValue: "NamedException"),
                                reason:"Something happened",
                                userInfo:nil)
         Bugsnag.shared.track(exception: exception)
    }

 var body: some Scene {
  WindowGroup {
   ContentView()
  }
 }
}
```

Just like in Android, we're using the platform's native types, but this time accessing them through our Shared module.
Our Kotlin Multiplatform solution is ready! 🎉

Can we use it from a Kotlin Multiplatform module?

Can we use our Bugsnag SDK adaptation from a Kotlin Multiplatform module? The answer is yes. In fact, making existing functionality accessible from our KMP modules is one of the main benefits driving the adoption of this technology. To better understand how it works, let's analyze a practical example with a repository.

```
class CrashRepository {

    private val coroutineScope =
CoroutineScope(Dispatchers.Default)

    suspend fun crash() {
        val handler = CoroutineExceptionHandler { _, exception -
>
            println("CoroutineExceptionHandler got $exception")
            // send log to bugsnag
            Bugsnag.track(exception.asTrackableException())
        }

        val job = coroutineScope.launch(handler) { // root
coroutine, running in GlobalScope
            throw AssertionError()
        }

        val deferred = coroutineScope.async(handler) { // also
root, but async instead of launch
            throw ArithmeticException() // Nothing will be
printed, relying on user to call deferred.await()
        }

        joinAll(job, deferred)
    }
}
```

This is a deliberate example of how to generate an exception 😀 . Our goal is to capture this exception using CoroutineExceptionHandler and report it to Bugsnag through our SDK adaptation. For this, we need to transform the exception to the TrackableException type.

> 💡 While we could have directly used the Throwable type in the track function signature since it's native to Kotlin, we chose to define the TrackableException type to achieve a clearer and more expressive API.

To implement this conversion, we create a Throwable extension function with platform-specific implementations.

```
//bugsnag.common.kt
expect fun Throwable.asTrackableException():
TrackableException

//bugsnag.android.kt
actual fun Throwable.asTrackableException() = this

//bugsnag.ios.kt
actual fun Throwable.asTrackableException() =
NSException.exceptionWithName(
    name = this::class.simpleName,
    reason = message ?: toString(),
    userInfo = null
)
```

In Android, since TrackableException is an alias for Throwable, the extension simply returns the original exception. In iOS, we create a new NSException that encapsulates the error information. With this complete implementation, let's analyze its advantages and disadvantages.

Pros and Cons

Pros

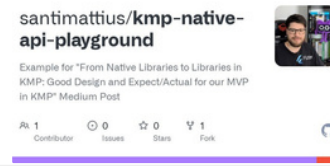
- Allows reusing existing solutions for MVP in KMP Maintains developer experience, especially for those coming from Android, by preserving familiar API definitions Facilitates early adoption of KMP Promotes
- robust and synchronized design between native solutions
-

Cons

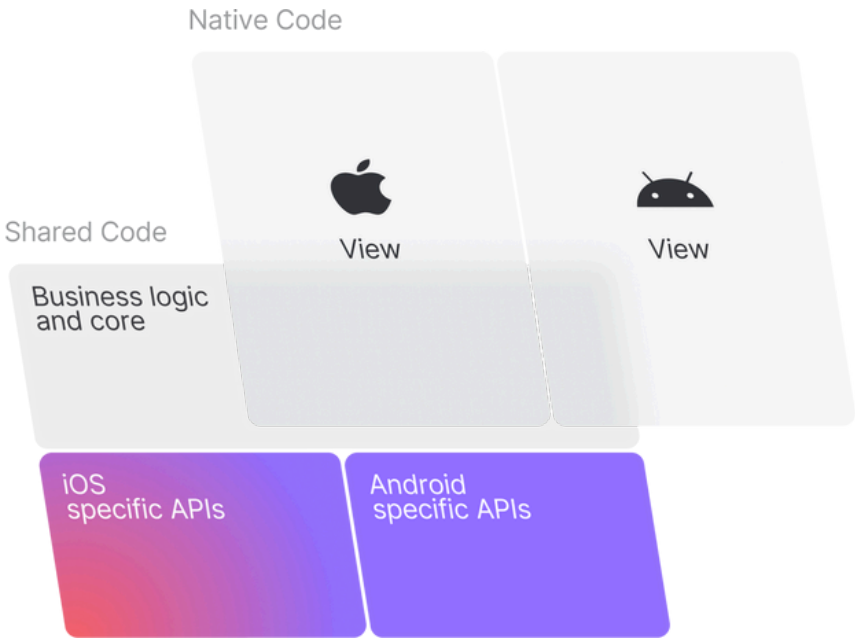
- APIs may have inconsistent designs across platforms
- Increases maintenance cost by adding a new technology stack
- Changes in native APIs require adjustments in KMP adaptation
- Cross-platform adaptation presents specific challenges: in Android, Java-
- Kotlin interoperability, and in iOS, converting Swift code not compatible with
- Objective-C

In my opinion, these solutions should be limited to specific cases, such as external integrations without native KMP support. It's important to evaluate the cost of rewriting these modules. Although interoperability works in both directions (we can use native dependencies in KMP and vice versa), it's better to avoid duplicate solutions. Software development experience has taught us the problems caused by code duplication.

GitHub - santimattius/kmp-native-api-playground
https://github.com/santimattius/kmp-native-api-playground



Chapter 7: Libraries



Chapter 7: Libraries

In Kotlin Multiplatform application development, it's essential to have a robust set of libraries that facilitate common tasks such as networking, data storage, and state management. These libraries are specifically designed to work consistently across all platforms supported by KMP, allowing developers to maintain a single codebase while leveraging native capabilities of each platform. Below, we'll explore some of the most popular and proven libraries in the KMP community, organized by categories according to their main functionality. These tools have been selected based on their maturity, active support, and adoption in real projects.

Networking

Networking libraries are fundamental for creating applications that work on both Android and iOS using Kotlin Multiplatform. These libraries allow both platforms to communicate with the Internet using the same code. Let's look at the best available libraries for making network connections in Kotlin Multiplatform:

Ktor

Ktor is an open-source framework for creating web and server applications in Kotlin. Ktor Client is a component of Ktor used to make HTTP requests from a Kotlin multiplatform application. With Ktor Client, you can make HTTP requests from your shared code across compatible platforms. Ktor Client provides a declarative and fluid API for making HTTP requests simply and efficiently, making it suitable for developing multiplatform applications that need to interact with web services. You can use Ktor Client to make GET, POST, PUT, DELETE, and other HTTP operations, as well as easily handle headers, parameters, and request and response data.

```
internal fun apiClient(baseUrl: String) = HttpClient {

    install(ContentNegotiation) {
        json(Json {
            prettyPrint = true
            isLenient = true
            ignoreUnknownKeys = true
        })
    }
    install(Logging) {
        logger = Logger.DEFAULT
        level = LogLevel.ALL
    }

    defaultRequest {
        url(baseUrl)
        contentType(ContentType.Application.Json)
    }
}
```

```
//using ktor client
class KtorRemoteMoviesDataSource(
    private val client: HttpClient,
) : RemoteMoviesDataSource {

    override suspend fun getMovies():
Result<List<MovieDto>> = runCatching {
        // invoke service
        val response = client.get("movie/popular")
        val result = response.body<MovieResponse>()
        result.results
    }
}
```

For more information on how to use Ktor in your Kotlin Multiplatform projects, you can check the official Ktor documentation.

Creating a cross-platform mobile application | Ktor

The Ktor HTTP client can be used in multiplatform projects. In this tutorial, we'll create a simple Kotlin Multiplatform Mobile application, which sends a request and receives a response body as plain HTML text.

https://ktor.io/docs/getting-started-ktor-client-multiplatform-mobile.html#code

Ktorfit

Ktorfit is a HTTP client/Kotlin Symbol Processor for Kotlin Multiplatform i Android, iOS, Js, Jvm, Linux) using KSP and Ktor clients inspired by Retrofit.

```
class ServiceCreator(baseUrl: String) {

    private val client = HttpClient {
        install(ContentNegotiation) {
            json(Json { isLenient = true; ignoreUnknownKeys = true
})
        }
    }
    private val ktorfit = Ktorfit.Builder()
        .baseUrl(baseUrl)
        .httpClient(client)
        .build()

    fun createPictureService() = ktorfit.create<PictureService>()
}

interface PictureService {

    @GET("random")
    suspend fun random(): Picture
}
```

For more information about Ktorfit and how to use it in your projects, you can check the official Ktorfit documentation https://foso.github.io/Ktorfit/

For a complete example, you can check out this Github repository: https://github.com/santimattius/kmp-networking

Storage

Mobile applications need to store information on the device. Kotlin Multiplatform makes this easier by providing tools that work the same way on both Android and iOS. Let's look at the best available tools for storing data in Kotlin Multiplatform, starting with how to handle user preferences.

Datastore

Jetpack Datastore is a data storage solution that allows you to store key-value pairs or objects written with protocol buffers. Datastore uses Kotlin coroutines and Flow to store data asynchronously, consistently, and transactionally. If you currently use SharedPreferences to store data, consider migrating to Datastore.

```
import androidx.datastore.core.DataStore
import
androidx.datastore.preferences.core.PreferenceDataStoreFactory
import androidx.datastore.preferences.core.Preferences
import kotlinx.atomicfu.locks.SynchronizedObject
import kotlinx.atomicfu.locks.synchronized
import okio.Path.Companion.toPath

private lateinit var dataStore: DataStore<Preferences>

private val lock = SynchronizedObject()

fun getDataStore(producePath: () -> String): DataStore<Preferences>
=
    synchronized(lock) {
        if (::dataStore.isInitialized) {
            dataStore
        } else {
            PreferenceDataStoreFactory.createWithPath(produceFile =
{ producePath().toPath() })
                .also { dataStore = it }
        }
    }

internal const val dataStoreFileName = "counter.preferences_pb"
```

For a complete example of using DataStore in KMP, you can check out the following link:

Multiplatform Settings

This is a Kotlin library for Multiplatform apps that enables common code to persist key-value data.

```
import com.russhwolf.settings.Settings
import
com.santimattius.kmp.skeleton.core.preferences.IntSettingConf
ig
import kotlinx.coroutines.flow.Flow

//commonMain
expect fun provideSettings(): Settings

class SettingsRepository(
    settings: Settings = provideSettings(),
) {


    private val _counter = IntSettingConfig(settings,
"counter", 0)
    val counter: Flow<Int> = _counter.value

    fun increment() {
        val value = _counter.get().toInt() + 1
        _counter.set("$value")
    }

    fun decrease() {
        val value = _counter.get().toInt() - 1
        if (value < 0) {
            _counter.set("0")
        } else {
            _counter.set("$value")
        }
    }
}
```

Initialization in Android/iOS

```
//androidMain
import androidx.preference.PreferenceManager
import android.content.SharedPreferences
import com.russhwolf.settings.Settings
import com.russhwolf.settings.SharedPreferencesSettings

actual fun provideSettings(context:Context): Settings {
    val preferences =
PreferenceManager.getDefaultSharedPreferences(context)
    return SharedPreferencesSettings(sharedPref)
}

//iosMain
import com.russhwolf.settings.NSUserDefaultsSettings
import com.russhwolf.settings.Settings
import platform.Foundation.NSUserDefaults

actual fun provideSettings(): Settings{
    return
NSUserDefaultsSettings(NSUserDefaults.standardUserDefaults)
}
```

As we can see, Multiplatform Settings uses the preferences implementations of each platform.

For more detailed information and documentation about Multiplatform Settings, you can check the official documentation on GitHub.

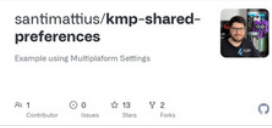
GitHub - russhwolf/multiplatform-settings: A Kotlin Multiplatform library for saving simple key-value data

https://github.com/russhwolf/multiplatform-settings

For a complete example, you can check out the following Github repository:

GitHub - santimattius/kmp-shared-preferences: Example using Multiplaform Settings

https://github.com/santimattius/kmp-shared-preferences



KStore

A tiny Kotlin multiplatform library that assists in saving and restoring objects to and from disk using kotlinx.coroutines, kotlinx.serialisation and okio. Inspired by RxStore.

Features

- 🔒 Read-write locks; with a mutex FIFO lock
- 💾 In-memory caching; read once from disk and reuse
- 📪 Default values; no file? no problem!
- 🚚 Migration support; moving shop? take your data with you
- 🚂 Multiplatform!

GitHub - xxfast/KStore: A tiny Kotlin multiplatform library that assists in saving and restoring objects to and from disk using kotlinx.coroutine

https://github.com/xxfast/KStore

Database

SQLDelight

SQLDelight generates typesafe Kotlin APIs from your SQL statements. It verifies your schema, statements, and migrations at compile-time and provides IDE features like autocomplete and refactoring which make writing and maintaining SQL simple.

```
CREATE TABLE Favorite (
    resourceId INTEGER PRIMARY KEY NOT NULL,
    title TEXT NOT NULL, overview TEXT NOT
    NULL, imageUrl TEXT NOT NULL, type TEXT
    NOT NULL

);

selectAllFavorite:
SELECT * FROM Favorite;
```

The SQLDelight plugin generates the necessary classes to interact with the database. In this example, AppDatabase and databaseQueries are generated by SQLDelight.

```
class SQLDelightFavoriteLocalDataSource(
    db: AppDatabase,
    private val dispatcher: CoroutineDispatcher =
Dispatchers.IO,
) : FavoriteLocalDataSource {

    private val databaseQueries = db.appDatabaseQueries

    override val all: Flow<List<Favorite>>
        get() = databaseQueries
            .selectAllFavorite()
            .asFlow()
            .mapToList(dispatcher)

}
```

For more information about SQLDelight and how to use it in your projects, you can check the official SQLDelight documentation https://cashapp.github.io/sqldelight/2.0.1/.

You can also find practical examples in the accompanying example.

Room

Room is Android's official database library, and now it can also be used in Kotlin Multiplatform projects. From now on, the same database you create for the Android target will be available across all targets.

```
// shared/src/commonMain/kotlin/Database.kt

@Database(entities = [TodoEntity::class], version = 1)
@ConstructedBy(AppDatabaseConstructor::class)  abstract
class AppDatabase : RoomDatabase() {
  abstract fun getDao(): TodoDao
}

// The Room compiler generates the `actual` implementations.
@Suppress("NO_ACTUAL_FOR_EXPECT")
expect object AppDatabaseConstructor :
RoomDatabaseConstructor<AppDatabase> {
    override fun initialize(): AppDatabase
}

@Dao
interface TodoDao {
  @Insert
  suspend fun insert(item: TodoEntity)

  @Query("SELECT count(*) FROM TodoEntity")
  suspend fun count(): Int

  @Query("SELECT * FROM TodoEntity")
  fun getAllAsFlow(): Flow<List<TodoEntity>>
}

@Entity
data class TodoEntity(
  @PrimaryKey(autoGenerate = true) val id: Long = 0,
  val title: String,
  val content: String
)
```

For more detailed information about Room in Kotlin Multiplatform, you can check the official documentation at the following link

Room (Kotlin Multiplatform) | Android Developers
The Room persistence library provides an abstraction layer over SQLite to allow for more robust database access while harnessing the full power of SQLite.
https://developer.android.com/kotlin/multiplatform/room

Multiplatform Jetpack Libraries

Google officially supports Kotlin Multiplatform for sharing business logic between iOS and Android. Many Jetpack libraries have already been adapted to take advantage of KMP.

The following Jetpack libraries are compatible with KMP:

- Annotations
  - https://developer.android.com/jetpack/androidx/releases/annotation
- Collection
  - https://developer.android.com/jetpack/androidx/releases/collection
- DataStore
  - https://developer.android.com/jetpack/androidx/releases/datastore
- Lifecycle
  - https://developer.android.com/jetpack/androidx/releases/lifecycle
- Paging
  - https://developer.android.com/jetpack/androidx/releases/paging
- Room
  - https://developer.android.com/jetpack/androidx/releases/room
- SQLite
  - https://developer.android.com/jetpack/androidx/releases/sqlite

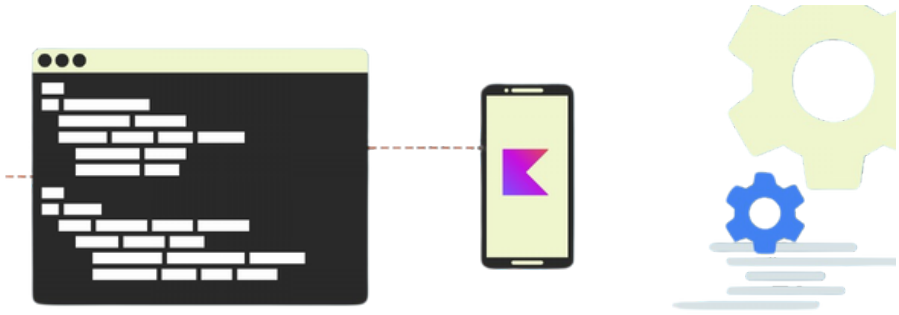
Android Support for Kotlin Multiplatform to Share Business Logic Across Mobile, Web, Server, and Desktop
https://android-developers.googleblog.com/2024/05/android-support-for-kotlin-multiplatform-to-share-business-logic-across-mobile-web-server-desktop.html

Summary

These are some of the libraries that help us address the challenge of delegating platform-specific implementations. They allow our code to be fully multiplatform, giving us the flexibility to choose which business logic to share while keeping critical aspects like the user interface separate. If you want to discover other libraries with Kotlin Multiplatform support, you can visit the following Github repository or the Jetbrains website where you can find official and community libraries.

- Klibs.io: https://klibs.io/
- Awesome Kotlin Multiplatform: https://github.com/terrakok/kmp-awesome

Chapter 8: Essential Tools and Plugins for Kotlin Multiplatform Development



Chapter 8: Essential Tools and Plugins for Kotlin Multiplatform Development

This chapter provides a concise overview of tools and plugins that enhance Kotlin Multiplatform application development. We'll explore essential tools like SKIE, KMMBridge, Xcode Kotlin, KMM Plugin, Dokka, and DeteKt. Each section examines their key features, development tool integrations, and how they improve the developer experience. This guide helps developers understand and navigate the Kotlin Multiplatform tooling ecosystem.

KDoctor

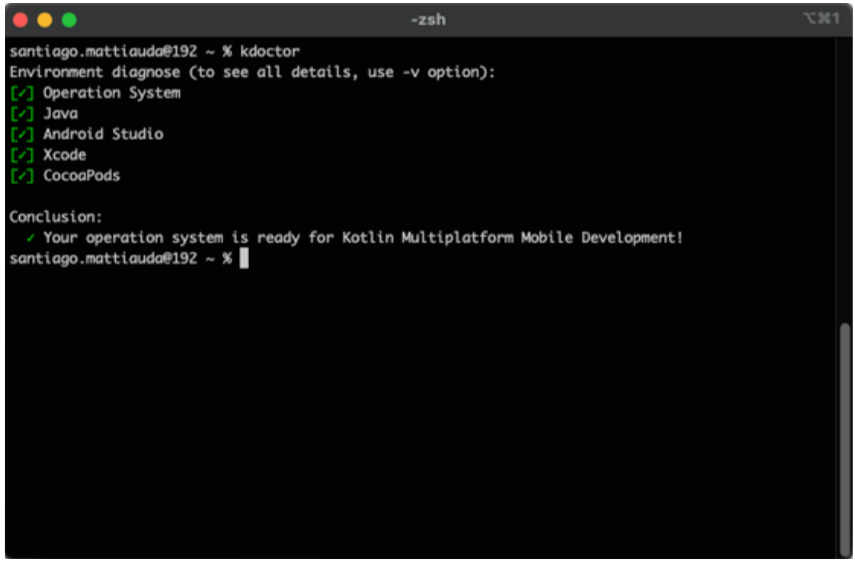
KDoctor ensures that all required components are properly installed and ready for use. If something is missed or not configured, KDoctor highlights the problem and suggests how to fix the problem.

KDoctor runs the following diagnostics:

- System - checks an operating system version
- JDK: checks that JDK installation and JAVA_HOME setting
- Android Studio - checks Android Studio installation, Kotlin and Kotlin Multiplatform Mobile plugins
- Xcode - checks Xcode installation and setup
- CocoaPods - checks ruby environment and cocoapods gem installation

Extra diagnostics:

- Synthetic generated project - downloads and builds project from https://github.com/Kotlin/kdoctor.
- Local Gradle Project - checks a user's project in the current directory



GitHub - Kotlin/kdoctor: Environment analysis tool Environment analysis tool. Contribute to Kotlin/kdoctor development by creating an account on GitHub.

https://github.com/Kotlin/kdoctor

SKIE

One of the main disadvantages of Kotlin Multiplatform is the lack of direct interaction with Swift. Without this, Swift can only communicate with Kotlin indirectly, through Objective-C. This approach, although functional, has many limitations and causes Kotlin to lose support for many of its language features. SKIE is a specialized plugin of the Kotlin native compiler that recovers support for some of these features by modifying the Xcode framework produced by the Kotlin compiler. Thanks to this, it is not necessary to change the way you distribute and consume your Kotlin Multiplatform frameworks. Developed by TouchLab, SKIE is designed to facilitate the development of projects with Kotlin Multiplatform, focusing on improving interoperability and the safe export of interfaces between different platforms. Below, its features oriented to Kotlin Multiplatform are described.

SKIE Features

- **Safe Interface Exportation**: Verifies type compatibility between platforms and ensures multi-platform interoperability.
- **Export Process Automation**: Automatically generates the necessary code and simplifies configuration with a Gradle DSL.
- **Multi-platform Compatibility**: Compatible with all Kotlin Multiplatform targets and integrates well with existing tools.
- **Support and Extensibility**: Provides comprehensive documentation and allows extensions to adapt to project needs.
- **Active Maintenance**: TouchLab regularly updates SKIE and has the backing of an active community.

SKIE is a powerful and essential tool for Kotlin Multiplatform developers. It allows them to safely and efficiently export interfaces between different platforms. With features that automate the export process and ensure type compatibility, SKIE significantly improves the development experience. In addition, it facilitates the creation of robust and easy-to-maintain multi-platform applications.

Next, we see an example of the use of Sealed Classes in Swift:

```
sealed class Status {
    object Loading : Status()
    data class Error(val message: String) : Status()
    data class Success(val result: SomeData) : Status()
}
```

Swift without SKIE:

```
func updateStatus(status: Status) {
    switch status {
    case _ as Status.Loading:
        showLoading()
    case let error as Status.Error:
        showError(message: error.message)
    case let success as Status.Success:
        showResult(data: success.result)
    default:
        fatalError("Unknown status")
    }
}
```

Swift with SKIE

```
func updateStatus(status: Status) {
    switch status {
    case _ as Status.Loading:
        showLoading()
    case let error as Status.Error:
        showError(message: error.message)
    case let success as Status.Success:
        showResult(data: success.result)
    default:
        fatalError("Unknown status")
    }
}
```

To see more features of SKIE, I leave its documentation below.

SKIE Intro | SKIE Welcome to the SKIE documentation. Let's give your iOS developer experience a boost, shall we?

https://skie.touchlab.co/intro

KMMBridge

KMMBridge is a Gradle tool that simplifies the publication of Kotlin Multiplatform framework binaries for Xcode. It allows publication on different backends and its use through CocoaPods or Swift Package Manager.

- Creates and publishes XCFramework zip files of your Kotlin modules
- Publishes on various online storage platforms
- Configures and publishes versions for SPM and CocoaPods for other developers

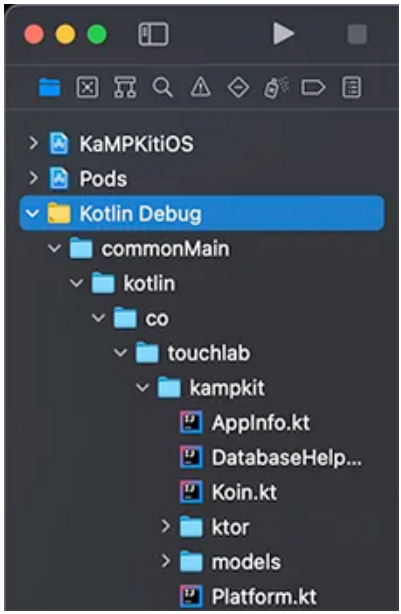
It offers a local development flow for SPM in addition to its publication functionality.
It is aimed at those who need to publish Xcode frameworks from Kotlin for iOS developers, useful for teams testing KMP, needing modularization, or publishing SDKs.

KMMBridge Intro | KMMBridge
KMMBridge is a set of Gradle tooling that facilitates publishing and consuming pre-built Kotlin Multiplatform Xcode Framework binaries.

https://kmmbridge.touchlab.co/docs/

Xcode Kotlin

The xcode-kotlin plugin allows you to debug Kotlin code running in an iOS application, directly from Xcode. This provides a smoother development and integration experience for iOS developers using Kotlin shared code. In addition, it allows a more accessible experience for large teams, where not all members can directly edit the shared code.



TouchLab's Xcode Kotlin integrates Kotlin into Xcode projects for iOS development, enhancing the experience of developers using Kotlin Multiplatform. It allows native integration of Kotlin code into Swift and Objective-C projects. Its main features are:

Features

Integration with Xcode

- **Direct support for Xcode:** It enables the inclusion of Kotlin code in Xcode projects, improving interoperability with Swift and Objective-C.
- **Compatibility with Xcode tools:** Ensures that Xcode tools, such as the debugger and compiler, work correctly with Kotlin.

Ease of configuration

- **Simplified configuration:** It offers predefined scripts and configurations to integrate Kotlin into Xcode, minimizing complexity and potential errors.
- **Detailed documentation:** Provides comprehensive guides for setting up and using Kotlin in iOS projects.

Automatic generation of bindings

- **Automatic bindings:** Automatically creates the necessary bindings to access Kotlin code from Swift and Objective-C, avoiding the manual writing of bridge code.

Improvement of the developer experience

- **Simplified workflow:** By automating the integration of Kotlin into Xcode.

Xcode Kotlin: Xcode support for Kotlin browsing and debugging
The xcode-kotlin plugin allows debugging of Kotlin code running in an iOS application, directly from Xcode.

https://touchlab.co/xcodekotlin

KMM Plugin

The KMM Plugin is an essential tool for developers using Kotlin Multiplatform in Android Studio. This plugin offers complete integration with Android Studio, thus making the creation, compilation, and debugging of KMP projects easier. With the KMM Plugin, developers can easily create Kotlin Multiplatform projects and access all the features of Kotlin Multiplatform directly from Android Studio. This includes the ability to define shared modules, manage dependencies, and perform unit tests on shared code, all within the familiar Android Studio development environment.

Plugin Features

Integration with Android Studio

- Integrates with Android Studio providing tools for cross-platform development.
- Facilitates the creation of KMP projects with automatic wizards.

Advanced Development Tools

- Offers autocomplete and refactoring tools.
- Provides advanced navigation and code search.

Debugging and Testing

- Supports Kotlin code debugging on Android and iOS.
- Facilitates unit and integration testing for shared code.

Developer Experience

- Automates repetitive tasks of cross-platform development.
- Provides clear and detailed error messages.

The KMM Plugin from JetBrains is an essential tool for Kotlin Multiplatform developers seeking seamless integration between Android and iOS. With features that simplify project setup and significantly improve developer experience, this plugin allows creating cross-platform applications, taking full advantage of Android Studio and Kotlin capabilities.

Kotlin Multiplatform Mobile: IntelliJ IDEs Plugin | Marketplace

The Kotlin Multiplatform Mobile plugin helps you develop applications that work on both Android and iOS. With the Kotlin Multiplatform Mobile plugin for Android...

https://plugins.jetbrains.com/plugin/14936-kotlin-multiplatform-mobile

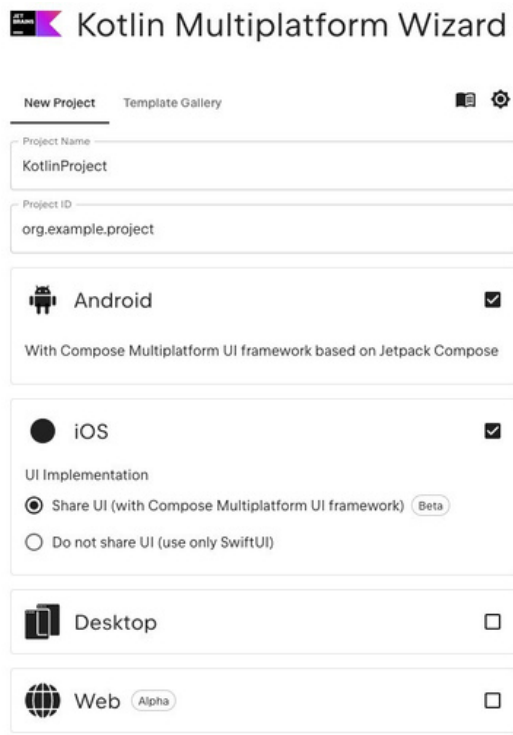
Kotlin Multiplatform Wizard

Kotlin Multiplatform Wizard is a tool from JetBrains designed to simplify the creation of new Kotlin Multiplatform projects. It has an intuitive user interface that guides developers through the initial project setup.

Features

- Helps to set up the structure and dependencies of new Kotlin Multiplatform projects.
- Allows selecting the platforms (JVM, JS, Android, iOS, etc.) for the project.
- Kotlin Multiplatform Wizard generates the base code, including configuration and sample code, saving time.

Kotlin Multiplatform Wizard is an excellent tool for any developer looking for a quick and easy way to start with Kotlin Multiplatform. With its focus on ease of use and customization, it facilitates the creation of new projects, allowing developers to focus on what really matters: writing high-quality code.

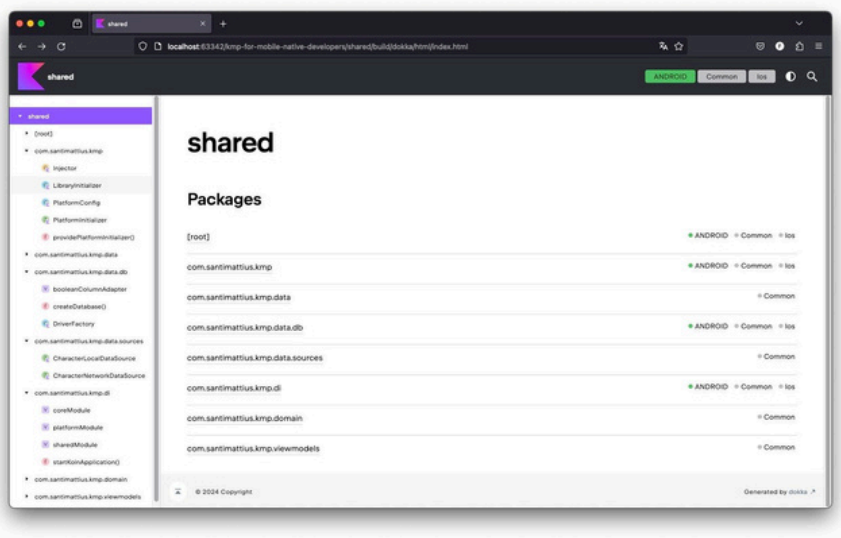


In addition, on the website you will find a gallery of other templates with different configurations. For example, you will find one based on Amper that we will see later.

Kotlin Multiplatform Wizard | JetBrains
Create your first multiplatform project using the Kotlin Multiplatform wizard for Android, iOS, and Desktop, or use one of the pre-made templates.
https://kmp.jetbrains.com/

Dokka: Generating Clear and Concise Documentation

Documentation is essential in any software development project. Dokka is a documentation generation tool specifically designed for Kotlin projects, including Kotlin Multiplatform. Dokka analyzes the source code and generates clear and concise documentation in HTML, Markdown, or Javadoc formats. This documentation describes the public API of a Kotlin Multiplatform project, making it easier for developers to understand how to use the different parts of the shared code. In addition, it promotes good development practices by making the documentation easily accessible for the entire team.



Dokka Features

Automatic Documentation Generation

- Automatically generates documentation from KDoc comments in Kotlin source code.
- Supports Kotlin Multiplatform projects, including modules for JVM, JS and Native.

Flexible Configuration

- Integrates with Gradle, allowing documentation generation to be configured from build.gradle.
- Offers advanced configurations to customize the format and content of the documentation.

Output in Multiple Formats

- Generates documentation in several formats, including HTML and Markdown.
- Can generate documentation in a format similar to Javadoc.

Integration with Build Tools

- Compatible with integration and continuous delivery (CI/CD) systems.
- Integrates with IntelliJ IDEA and other JetBrains-based IDEs.

Documentation Enrichment

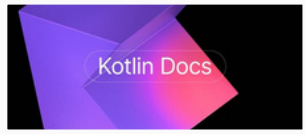
- Allows the inclusion of images, links and other external resources in the documentation.
- Can include code examples and snippets in the documentation.

Optimization of the Developer Experience

- Generates documentation with a clear and easy-to-navigate structure.
- Supports the inclusion of annotations and comments in the documentation.

Dokka, essential for Kotlin developers, supports clear and up-to-date documentation of projects, including Kotlin Multiplatform. It facilitates the generation of documentation, supports various formats and integrates with build tools and CI/CD, improving the quality and accessibility of code documentation. This benefits the understanding and maintenance of projects, for individual developers and teams.

Get started with Dokka | Kotlin
Below you can find simple instructions to help you get started with Dokka.



https://kotlinlang.org/docs/dokka-get-started.html

DeteKt: Code Quality Improvement

Code quality is essential for the maintainability and scalability of any software project. DeteKt is a static code analysis tool designed to identify and correct common quality problems in Kotlin projects. With DeteKt, developers can automatically analyze their Kotlin Multiplatform code for errors, redundancies, incorrect naming conventions, and other potential problems. This helps ensure that shared code is clean, consistent, and easy to maintain over time.

DeteKt Features

Static Code Analysis

- **Code Problem Detection:** Analyzes source code to identify common issues such as style errors, potential failures, and complexity problems. This helps improve code quality. **Code**
- **Quality Report:** Generates detailed reports highlighting areas of the code that require attention, thus facilitating the identification and correction of problems.

Support for Kotlin Multiplatform

- Multiple Target Analysis: It is compatible with Kotlin Multiplatform projects and allows analysis of shared and platform-specific code (JVM, JS, Native).
- Code Consistency: Ensures that code quality rules are consistently applied across different platforms, maintaining a uniform standard.

Flexible and Customizable Configuration

- Gradle Plugin: Easily integrates with Gradle, allowing code analysis to be configured and run as part of the build process.
- Customizable Rules: Provides a wide range of predefined rules and allows developers to define custom rules to suit the specific standards of their project.

Integration with Development Tools

- CI/CD Integration: It is compatible with continuous integration and delivery (CI/CD) systems, which facilitates the incorporation of static analysis into the development pipeline.
- IDE Compatibility: Works well with IntelliJ IDEA and Android Studio, allowing developers to view and correct code quality problems directly in their development environment.

Detailed and Actionable Reports

- Various Report Formats: Generates reports in various formats, such as HTML, XML, and plain text, which facilitates their integration with other tools and systems.
- Link to Source Code: Provides direct links to the lines of code that have problems, making it easier to review and correct them.

Continuous Code Improvement

- Best Practice Rules: Includes rules based on Kotlin development best practices, helping developers write cleaner and more maintainable code.
- Code Smell Detection: Identifies code smells like long classes, complex methods, and code duplication, promoting healthier software design.

Extensibility

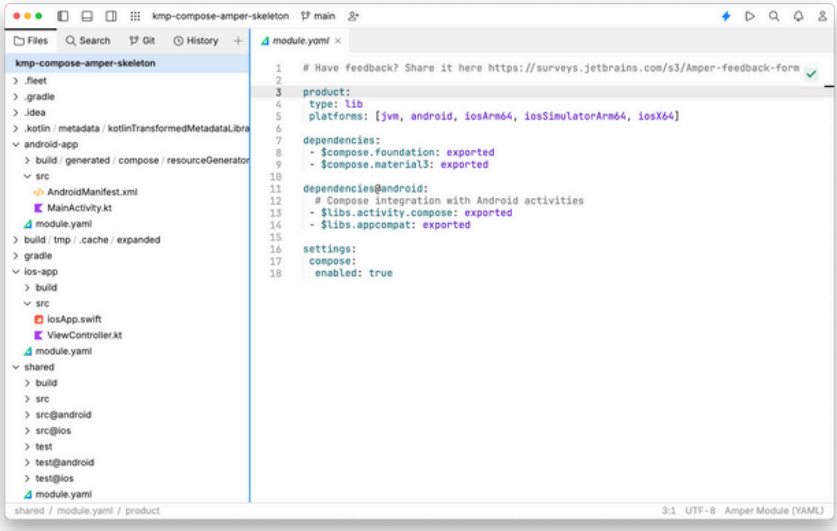
- Plugins and Extensions: Allows the creation of plugins and extensions to add additional functionality or adapt DeteKt to the specific needs of the project.
- Exception Configuration: Offers options for configuring exceptions and exclusions, allowing the analysis to be tailored to the specific contexts of the project.

DeteKt is a powerful and essential tool for Kotlin developers looking to maintain high-quality, clean, and error-free code. With features that facilitate the static analysis of Kotlin Multiplatform projects, ensure code consistency, and allow smooth integration with development tools and CI/CD, DeteKt significantly improves code quality and team productivity. This tool allows developers to proactively detect and correct problems, promoting best practices and healthier software design.

Hello from detekt | detekt
The official website of detekt: A static analyzer for Kotlin
https://detekt.dev/

Amper

Amper is a build system developed by JetBrains, specifically designed for the Kotlin ecosystem, including Kotlin Multiplatform projects. Below, its main features focused on Kotlin Multiplatform are described.



Amper Features

Process Optimization

- **Build Efficiency:** Optimizes the construction process by reducing compilation times and improving the overall workflow efficiency.
- **Incremental Compilation:** Supports incremental compilation, where only recent changes in the source code are recompiled, speeding up the build process.

Support for Kotlin Multiplatform

- **Multiplatform Compatibility:** It is designed to handle Kotlin Multiplatform projects, allowing the construction of common and platform-specific modules (JVM, JS, Native) efficiently.
- **Simplified Configuration:** Provides simplified configuration for multiplatform projects, making it easier to manage and maintain build configurations.

Dependency Management

- **Efficient Dependency Handling:** Provides an advanced system for dependency management, ensuring that all necessary libraries and frameworks are correctly integrated and updated.
- **Conflict Resolution:** Automatically handles dependency conflicts, facilitating the integration of multiple libraries and components into a project.

Task Automation

- **Customizable Build Tasks´** Allows the creation and configuration of customizable build tasks, adapting to the specific requirements of each project.
- **Support for Common Tasks´** Includes support for common tasks such as compilation, packaging, test execution, and documentation generation, facilitating the development workflow.

Amper is an advanced and efficient build system developed by JetBrains, ideal for Kotlin projects, including Kotlin Multiplatform. Its features optimize the construction process, facilitate dependency management, and provide seamless integration with JetBrains tools and ecosystems. In this way, Amper significantly improves the efficiency and productivity of the development workflow. This tool allows developers to manage complex projects more effectively, ensuring fast, reliable, and well-integrated builds in the development environment.

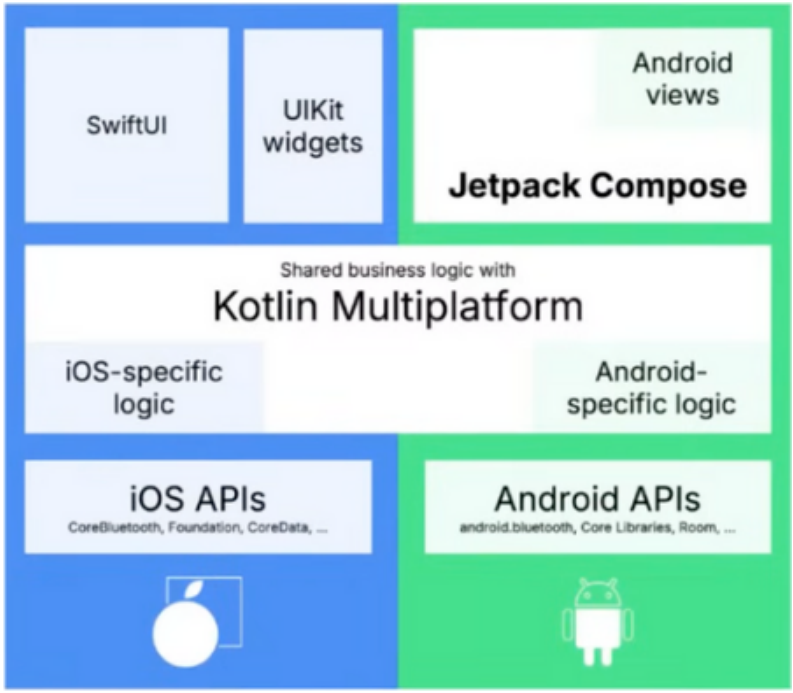
GitHub: JetBrains/amper: Amper - a project configuration and build tool with a focus on the user experience and the IDE support Amper - a project configuration and build tool with a focus on the user experience and the IDE support: JetBrains/amper

https://github.com/JetBrains/amper

Summary

In summary, this chapter provides a comprehensive overview of the various tools and technologies available for efficient application development using Kotlin Multiplatform. From SKIE and KMMBridge to facilitate integration with Xcode, to Dokka and DeteKt for enhancing code quality and documentation. The importance of analysis tools and code coverage tools like DeteKt and IDEs and build systems like Fleet and Amper are also highlighted. Each tool has a specific purpose, but all work together to provide a smooth and efficient development experience on Kotlin Multiplatform. Through this article, developers can gain a deeper understanding of these tools and how they can enhance their application development process.

Hello Kotlin Multiplatform!



References

This section compiles all bibliographic references used as sources for the development of the content of this book on Kotlin Multiplatform. The references are organized by chapters to facilitate consultation and follow-up, providing the necessary information to access the original cited materials. Each reference includes the author or organization, the title of the resource, the platform or website where it is available, and the corresponding link to directly access the content. These sources have been carefully selected to ensure updated, accurate, and relevant information about multiplatform development with Kotlin. Readers can use these references to delve deeper into specific topics, verify information, or explore additional concepts that complement the content presented in each chapter.

Chapter 1: Introduction to Kotlin Multiplatform

- Jetbrains. **“Compose Multiplatform, Develop stunning shared UIs for Android, iOS, desktop, and web”**. *JetBrains.com*. https://www.jetbrains.com/lp/compose-multiplatform/
- Jetbrains. **“Choosing a configuration for your Kotlin Multiplatform project”**. *JetBrains.com*. https://www.jetbrains.com/help/kotlin-multiplatform-dev/multiplatform-project-configuration.html
- Jetbrains. **”Kotlin Multiplatform, Simplify the development of cross-platform projects and reduce the time spent writing and maintaining the same code for different platforms”**. *Kotlinlang.org*. https://kotlinlang.org/docs/multiplatform.html
- Jetbrains. **”Create a multiplatform app using Ktor and SQLDelight”**. *JetBrains.com*. https://www.jetbrains.com/help/kotlin-multiplatform-dev/multiplatform-ktor-sqldelight.html
- Jetbrains. **“Kotlin Multiplatform, Share code on your terms”**. *JetBrains.com*. https://www.jetbrains.com/kotlin-multiplatform/

Chapter 2: Understanding the Basic Project Structure

- Jetbrains. **“CocoaPods overview and setup ”**. *Kotlinlang.org*. https://kotlinlang.org/docs/native-cocoapods.html
- JetBrains. **"Kotlin Gradle Plugin"**. *Kotlinlang.org*. https://kotlinlang.org/docs/gradle.html.
- JetBrains. **"Getting Started with Kotlin Multiplatform"**. *Kotlinlang.org*. https://kotlinlang.org/docs/multiplatform-get-started.html.
- JetBrains. **"Discover Kotlin Multiplatform Project"**. *Kotlinlang.org*. https://kotlinlang.org/docs/multiplatform-discover-project.html.
- JetBrains. **"Advanced Project Structure in Kotlin Multiplatform"**. *Kotlinlang.org*. https://kotlinlang.org/docs/multiplatform-advanced-project-structure.html.
- JetBrains. **"Sharing Code Across Platforms"**. *Kotlinlang.org*. https://kotlinlang.org/docs/multiplatform-share-on-platforms.html.
- JetBrains. **"Expect and Actual Declarations"**. *Kotlinlang.org*. https://kotlinlang.org/docs/multiplatform-expect-actual.html.
- JetBrains. **"Kotlin Multiplatform Hierarchy"**. *Kotlinlang.org*. https://kotlinlang.org/docs/multiplatform-hierarchy.html.
- JetBrains. **"Default Hierarchy Template"**. *Kotlinlang.org*. https://kotlinlang.org/docs/multiplatform-hierarchy.html#default-hierarchy-template.
- JetBrains. **"Manual Configuration of Kotlin Multiplatform Hierarchy"**. *Kotlinlang.org*. https://kotlinlang.org/docs/multiplatform-hierarchy.html#manual-configuration.

Chapter 3: Dependency Injection

- Kotlin. **"Kotlin Multiplatform update"**. *Twitter*. https://twitter.com/kotlin/status/1710269674016125100.
- Kosi-Libs. **"Getting Started with Kodein 7.22"**. *Kosi-Libs.org*. https://kosi-libs.org/kodein/7.22/getting-started.html.
- Insert Koin. **"Koin: Pragmatic Kotlin Dependency Injection"**. *Insert-Koin.io*. https://insert-koin.io/.
- P-Y. **"DIY: Your Own Dependency Injection Library"**. *P-Y Blog*. https://blog.p-y.wtf/diy-your-own-dependency-injection-library.
- Evan Tatarka. **"Kotlin Inject"**. *GitHub*. https://github.com/evant/kotlin-inject.

Chapter 4: Modularization

- JetBrains. **"Multiplatform Project Configuration: Module Configurations".** *JetBrains Help*. https://www.jetbrains.com/help/kotlin-multiplatform-dev/multiplatform-project-configuration.html#module-configurations.
- Touchlab. **"Optimizing Gradle Builds in Multi-Module Projects"**. *Touchlab*. https://touchlab.co/optimizing-gradle-builds-in-Multi-module-projects.
- Touchlab. **"Multiple Kotlin Frameworks in an Application"**. *Touchlab*. https://touchlab.co/multiple-kotlin-frameworks-in-application.
- Google. **"Modularization Patterns"**. *Android Developers*. https://developer.android.com/topic/modularization/patterns.

Chapter 5: Testing

- Fowler, Martin. **"The Practical Test Pyramid"**. *MartinFowler.com*. https://martinfowler.com/articles/practical-test-pyramid.html.
- Fowler, Martin. **"Integration Test"**. *MartinFowler.com*. https://martinfowler.com/bliki/IntegrationTest.html.
- Fowler, Martin. **"Object Mother"**. *MartinFowler.com*. https://martinfowler.com/bliki/ObjectMother.html.
- Refactoring Guru. **"Builder Design Pattern"**. *Refactoring.Guru*. https://refactoring.guru/design-patterns/builder.
- Fowler, Martin. **"Test Double"**. *MartinFowler.com*. https://martinfowler.com/bliki/TestDouble.html.
- Google. **"Fundamentals of Testing"**. *Android Developers*. https://developer.android.com/training/testing/fundamentals.
- Kotlin. **"Kotlinx-Kover"**. *GitHub*. https://github.com/Kotlin/kotlinx-kover.

Chapter 6: Using Native Libraries in Kotlin Multiplatform

- JetBrains. **"Managing iOS Dependencies in Kotlin Multiplatform: With CocoaPods"**. *Kotlinlang.org*. https://kotlinlang.org/docs/multiplatform-ios-dependencies.html#with-cocoapods.
- JetBrains. **"Managing iOS Dependencies in Kotlin Multiplatform: Without CocoaPods"**. *Kotlinlang.org*. https://kotlinlang.org/docs/multiplatform-ios-dependencies.html#without-cocoapods.
- JetBrains. **"Expect and Actual Declarations"**. *Kotlinlang.org*. https://kotlinlang.org/docs/multiplatform-expect-actual.html.
- JetBrains. **"Type Aliases in Kotlin"**. *Kotlinlang.org*. https://kotlinlang.org/docs/type-aliases.html.

Practical Example Repositories for the Book

The following repositories provide practical example code that complements the theoretical content presented in this book. Each repository is designed to illustrate specific concepts of Kotlin Multiplatform development and serve as a reference for implementing patterns and techniques discussed in the different chapters. These examples range from the basic structure of a KMP project to more advanced implementations such as modularization, integration with native APIs, and preference management. Readers can clone, explore, and modify these repositories to directly experiment with the code as they progress in their le ar ning. It is recommended to review these examples in parallel with reading the corresponding chapters to gain a deeper and more practical understanding of the presented concepts.

- **KMP for Mobile Native Developers**
  - https://github.com/santimattius/kmp-for-mobile-native-developers
- **KMP Multi Module Example**
  - https://github.com/santimattius/kmp-multi-module-example
- **KMP Preferences Example**
  - https://github.com/santimattius/kmp-preferences-example
- **KMP Native API Playground**
  - https://github.com/santimattius/kmp-native-api-playground
- **KMP Networking**
  - https://github.com/santimattius/kmp-networking