KMM Advanced. Делаем общий DI для iOS и Android

Анна Жаркова
Lead Mobile developer

О себе

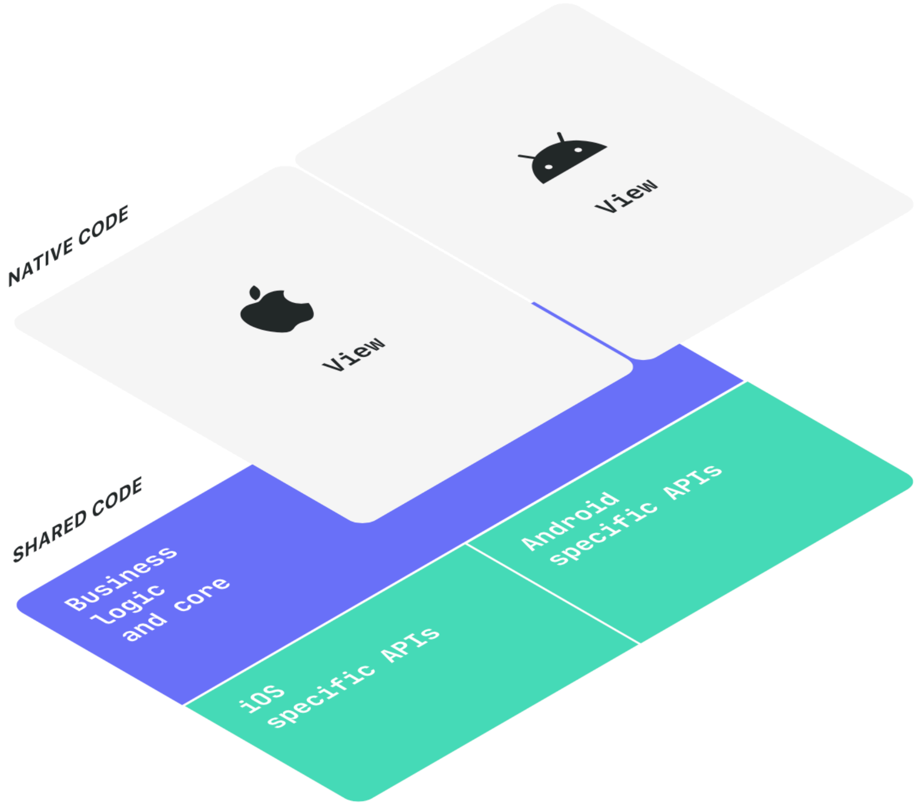
- Опыт в коммерческой разработке ПО более 7 лет
- Ведущий мобильный разработчик компании «Usetech».
- Разработка iOS и Android приложений нативных (Swift/Objective-C, Kotlin/Java) и кроссплатформенных (Xamarin, Kotlin multiplatform)
- Разработка архитектуры мобильных приложений для обеих платформ
- Управление командой направления.
- Ведет младших разработчиков (менторство).
- Пишет статьи, выступает на митапах (Mobius Moscow 2020, AppsLive 2020)

Goals. Сегодня рассмотрим:

- Kotlin Multiplatform. Особенности связи общего и нативного кода
- Существующие готовые решения (нативные и не только), и почему они не подходят
- Что должно быть в хорошем DI решении
- Нюансы и подводные камни реализации под Android и iOS
- Нативные (и не только) декорации и оптимизация решения

Kotlin Multiplatform

- SDK, не фреймворк
- Kotlin модуль компилируется JVM в байткод для Android и LLVM в байткод для iOS
- Шарить можно практически все (кроме UI)



USETECH

2021 г.

Kotlin Multiplatform

Kotlin Common

Kotlin JVM

Kotlin Native

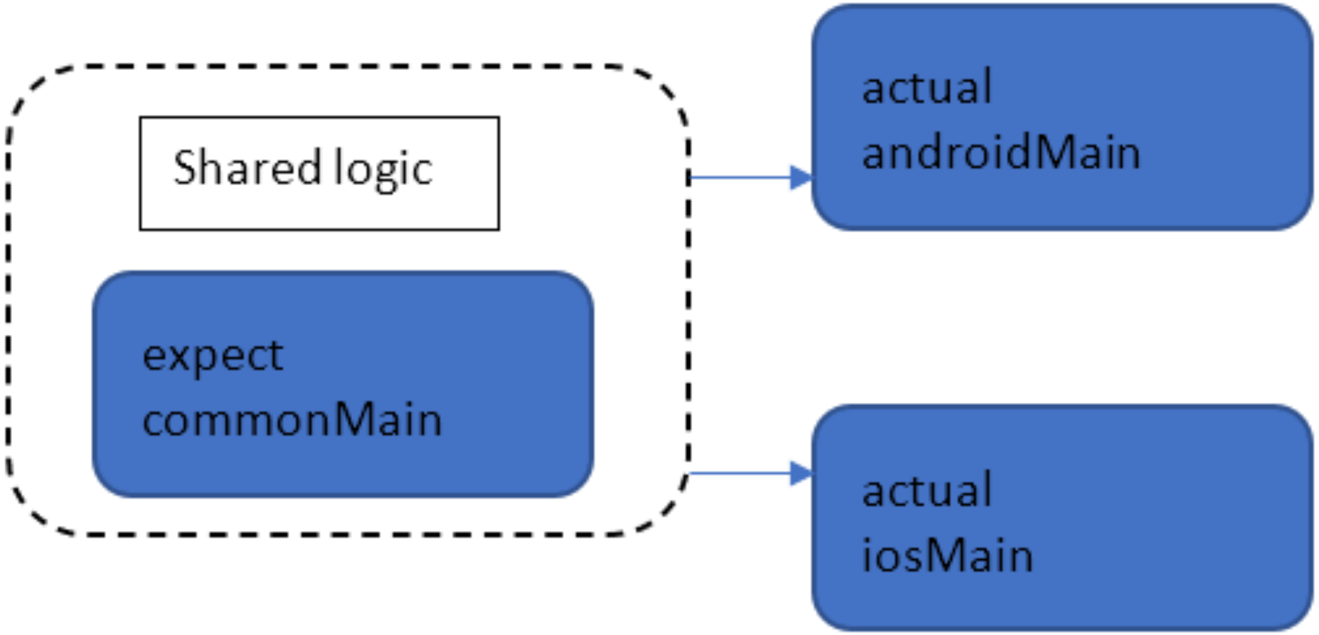
Kotlin JS

Shared vs нативное

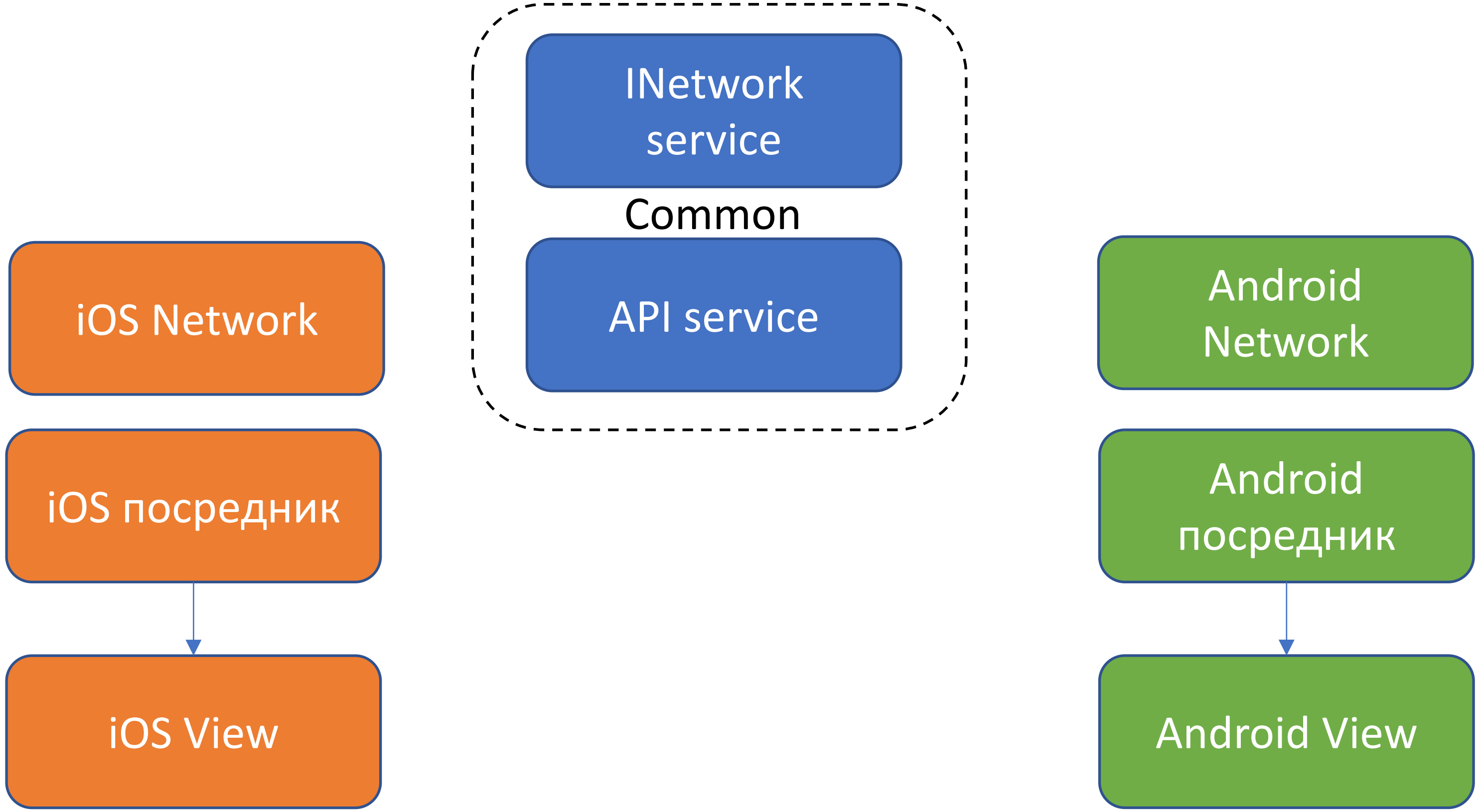
- сервисы для работы с сетью
- сервисы для работы с БД
- модели данных
- архитектура (extra)

- Экраны, View (UI)
- Работа с датчиками
- Работа с платформой

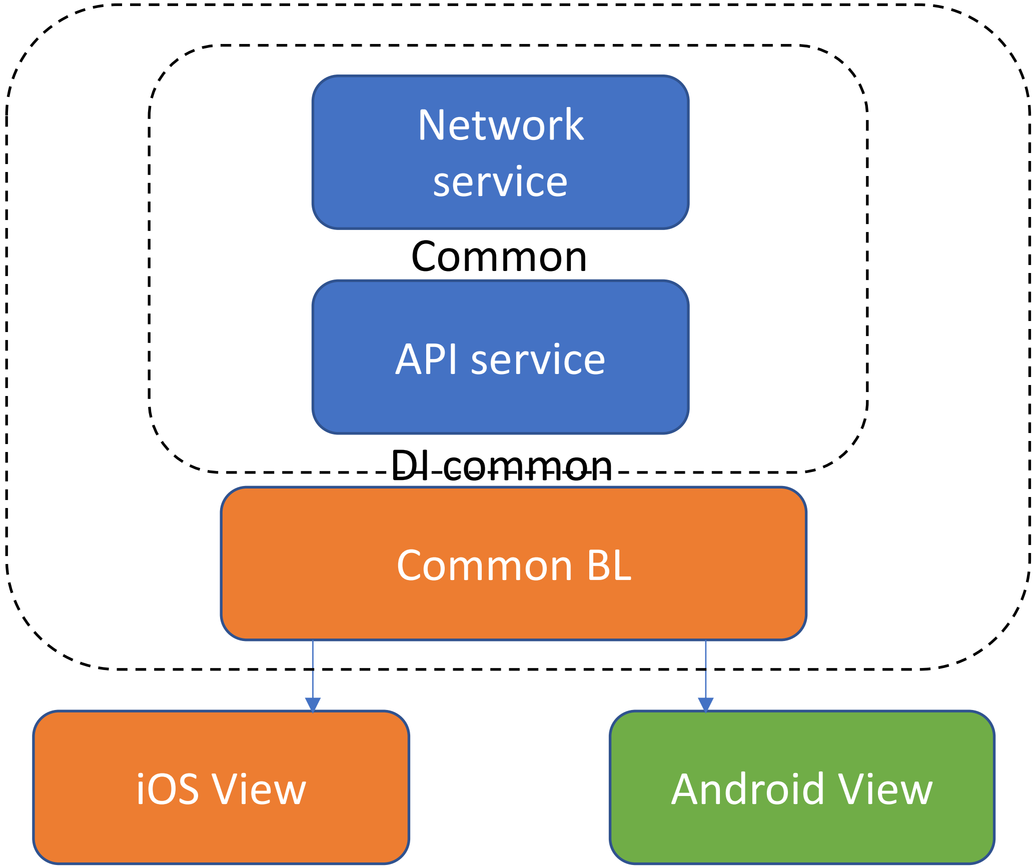
Shared (общая логика)



Связь между слоями



Связь между слоями



Рекомендуемое решение (JetBrains)

Dependency injection

Refactor your application into independent modules that can work on their own. The Dependency Injection design pattern ↗ is very useful to create such an architecture.

Android Developers > Docs > Guides

Dependency injection in Android

Dependency injection (DI) is a technique widely used in programming and well suited to Android development. By following the principles of DI, you lay the groundwork for good app architecture.

Implementing dependency injection provides you with the following advantages:

- Reusability of code
- Ease of refactoring
- Ease of testing

Способы реализации DI

1. Manual dependency injection

2. Взять готовое решение

Manual dependency injection

Service Locator

```
public static class ServiceLocator
{
    public static object GetService(Type type) {}
    public static T GetService<T>() {}
}


public interface IServiceLocator
{
    T GetService<T>();
}
```

Manual dependency injection

Service Locator

```
object ServiceLocator {
    fun <reified T> register(factory: () -> T)
    fun <reified T> resolve(): T
}

class MoviesService {
    private val network: NetworkService = ServiceLocator.resolve()
}

fun main() {
    ServiceLocator.register<NetworkService> { NetworkService() }
    val moviesService = MoviesService()
}
```

Что должно быть в хорошем DI решении

1. Потокобезопасность (thread-safety)

2. Корректное управление памятью (поддержка weak)

3. Поддержка разных скоупов

4. Поддержка синглтонов

*Сахар

Что должно быть в хорошем DI решении для KMM

1. Потокобезопасность (thread-safety)
2. Корректное управление памятью (поддержка weak)
3. Поддержка разных скоупов
4. Поддержка синглтонов
5. <u>Работает на iOS и Android (учитывая нюансы)</u>

Что еще должно быть

1. Работоспособность

2. Устойчивость к ошибкам

3. Возможность дебага…

* Управление зависимостями, circular dependencies

Стандартно для работающего решения

Сконцентрируемся на реализации под KMM

USETECH

Платформенные нюансы на KMM

Kotlin common

iOS:
Kotlin native

1. Управление потоками
2. Управление памятью (weak)
3. Инициализация (Kotlin Objective-C interop)
4. Соотношение типов (Kotlin Objective-C interop)
5. Синглтоны

Android:
Kotlin JVM

Готовые решения для Android

1. Dagger

2. Hilt

Готовые решения для Android. Почему нет

Dagger, Hilt не подходят для Common!!!

- Используют механизм JVM
- Annotations -> KAPT поддерживается только в JVM
- iOS - > Kotlin Native

Возможные решения для iOS

- Swinject https://github.com/Swinject/Swinject
- PropertyWrappers (Swift 5.1) + ServiceLocator

Возможные решения для iOS. Минус

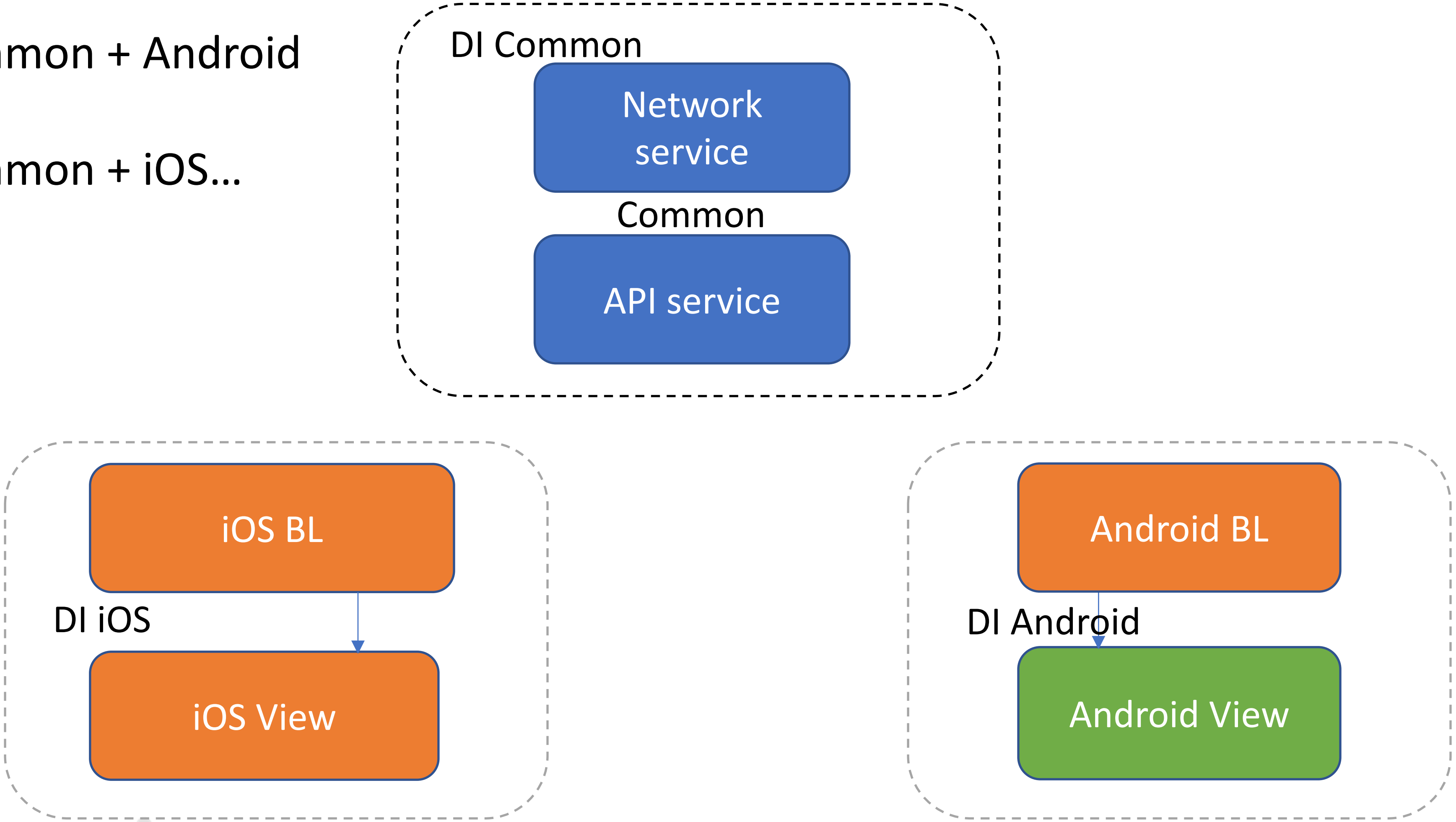
- Есть только на стороне iOS
- Нужно решение и в Common

А если использовать
разное решение на каждой платформе?

USE TECH

Комбинирование DI на платформах

- DI Common + Android
- DI Common + iOS…



Как соединить DI в общей части и нативе?

Expect – actual ?

Прокидывать целую библиотеку в Common ?
Подключать через gradle/pods и использовать в iOSMain/AndroidMain?

Не best practices

```
expect class DIManager


//iOS
actual typealias DIManager = SwinjectManager

//Android
actual typealias DIManager = HiltManager
```

USETECH

Как соединить DI в общей части и нативе?

В данном случае можно обратиться к common за нужным экземпляром сервиса из DI Common.

Подойдет для разной архитектуры нативных частей (условно)

```
//iOS
class MoviesVC : UIViewController {
    lazy var moviesService: MoviesService? = {
        return DICommon.shared.moviesService
    }()
    //...
}

//Android
class MoviesFragment: BaseFragment() {
    val moviesService: MoviesService? = DICommon.shared.moviesService
    //...
}
```

Есть готовое решение и для KMM

USETECH

Готовые решения для КММ

- Koin
- Kodein

<u>В чем же проблема???</u>

Готовые решения для КММ. Koin

- Заточен в основном под Android и JVM
  (но нет WeakReference)

- Kotlin Native:
  - нет weak reference
  - потокобезопасность синглтонов

- Есть отдельная версия под iOS

Готовые решения для KMM. Kodein

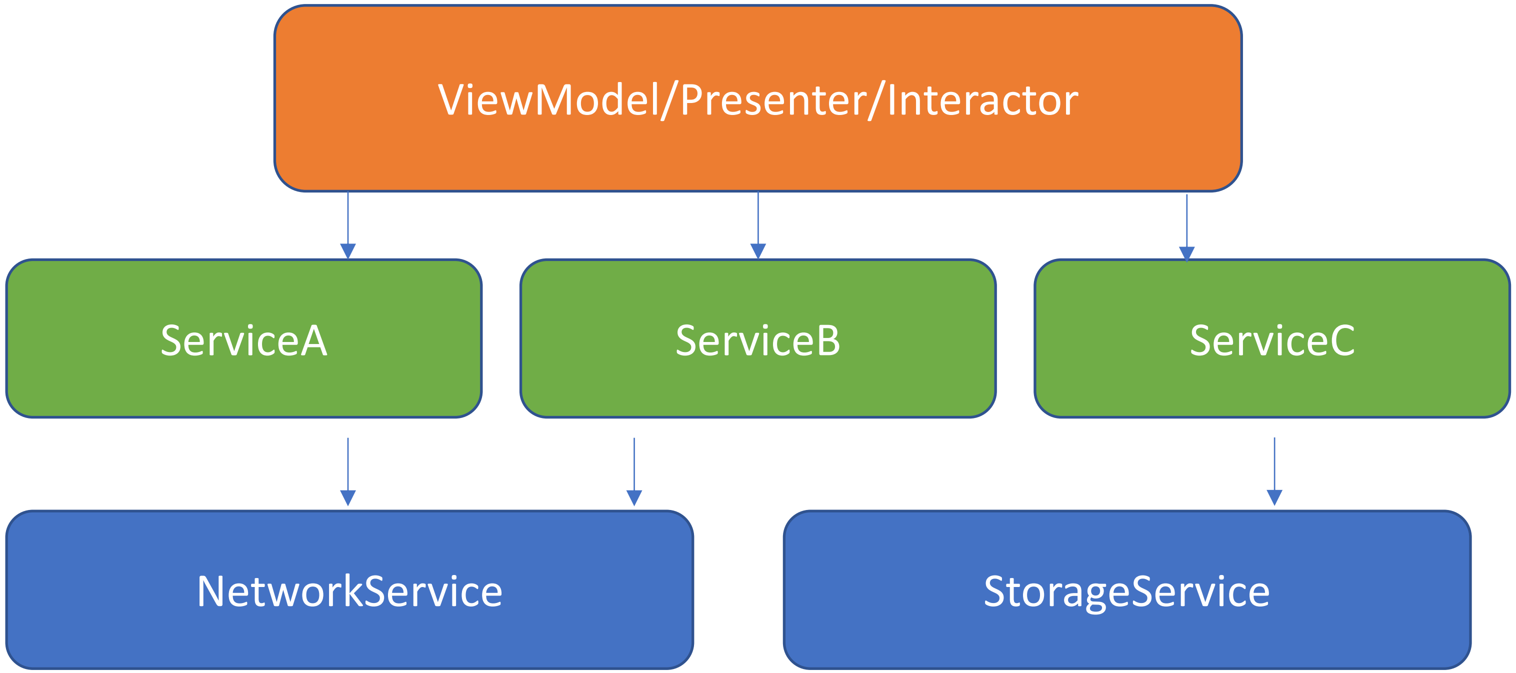
- Заточен в основном под Android и JVM
- Kotlin Native:
  - нет weak reference
  - потокобезопасность синглтонов

Можно поковыряться и доделать в чужом коде,

Но попробуем сделать что-то свое

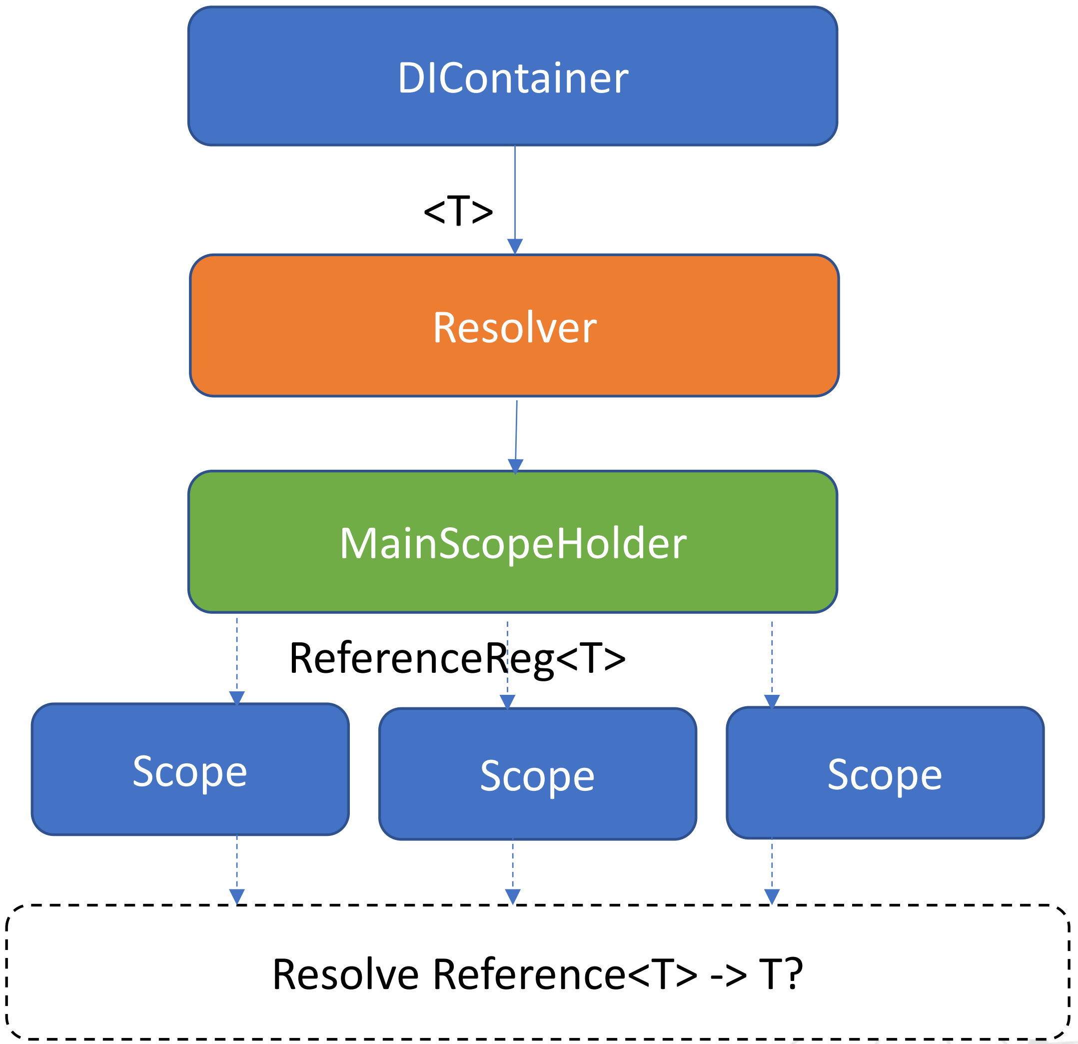
Практическое применение

DI для связей между слоями BL



Описание решения

DIContainer -> Resolver



Описание решения

DIContainer -> Resolver

<u>Можно создавать свои контейнеры на модуль</u>

```
public class DIContainer : ContainerProtocol {
   val resolver = Resolver()

   public fun <T : Any> register(clazz: KClass<T>,
   scope: ScopeType = ScopeType.Graph,
   fabric: () -> T?) {
       resolver.register(clazz,scope,fabric)
   }

   public fun <T : Any> resolve(clazz: KClass<T>): T? {
       return resolver.resolve(clazz)
   }
 /*...*/
}
```

Описание решения

Resolver -> Scope

-> RegistrationRef map

```
public class Resolver : ResolverProtocol {
    private var defaultScope: ScopeType = ScopeType.Graph
    private var registrations: HashMap<Int, RegistrationRef<*>> = hashMapOf()
    private  val scopeHolder = MainScopeHolder()

   /*...Магия регистрации*/

   override fun <T : Any> resolve(clazz: KClass<T>): T? {
        val reference = this.searchReference(clazz)
        if (reference != null) {
            val scopeType = reference.scope
            return selectScope(scopeType).resolve(reference,  clazz)
        }
        return null
    }
```

Описание решения

Поиск регистрационной инфо по ключу qualifiedName

```
fun <T : Any> searchReference(clazz: KClass<T>): RegistrationRef<T>? {
    val name = clazz.qualifiedName
    trySynchronized(registrations) {
        return (registrations.values.filter { it.name == name })
        .firstOrNull() as RegistrationRef<T>?
    }
}
```

Описание решения

Скоуп выбирается по инфо из регистрации

```
val scopeHolder = MainScopeHolder()

fun selectScope(scope: ScopeType): ScopeProtocol {
    when (scope) {
        ScopeType.Cached -> return scopeHolder.cachedScope
        ScopeType.Container -> return scopeHolder
            .containerScope
        ScopeType.Entity -> return scopeHolder.entityScope
        ScopeType.Graph -> return scopeHolder.graphScope
    }
}
```

USETECH

Описание решения

Скоуп хранится регистрационной инфо и задается при регистрации в контейнере

```
class Resolver {
    //Скоупы разные
    private fun <T : Any> add(clazz: KClass<T>,scope: ScopeType, /*...*/) {
        val reference = RegReference<T>(/*...*/)
        reference.scope = scope
        val key = clazz.hashCode()

        if (registrations.containsKey(key)) {
            registrations[key] = reference
        } else {
            registrations.put(key, reference)
        }
    }

    fun <T : Any> register(clazz: KClass<T>, scope: ScopeType,/*...*/) {
        this.add(clazz = clazz, scope = scope, fabric = fabric)
    }
```

Типы используемых ссылок

RegistrationReg<T> - инфо для регистрации компонента

```
class RegistrationRef<T:Any>(val name: String,
val initor: ()->T?) {
    var scope: ScopeType = ScopeType.Graph
    var resolveProperties: ((T?)->Unit)? = null

    fun resolve():T? {
        return initor.invoke()
    }

    fun resolveDependencies(resolvedInstance: T?){
        if (resolvedInstance != null) {
            resolveProperties?.invoke(resolvedInstance)
        }
    }
}
```

Типы используемых ссылок

LazyReference<T> - lazy-экземпляры

WeakReference<T> - слабые ссылки

SingletonReference<T> - синглтон

Скоупы

Определяет время и условие жизни компонента/ ссылки

```
interface ScopeProtocol {
    fun<T:Any> resolve(regReference: RegistrationRef<T>, clazz: KClass<T>):T?

    fun<T:Any> save(instance: T, key: String)

    fun<T:Any> save(reference: RegistrationRef<T>, key: String):T?

    fun<T:Any> previous(key: String):T?
}
```

USETECH

Скоупы (все названия условны)

CachedScope – отдает уже существующий экземпляр компонента

```
class CachedScope : ScopeProtocol {
private var cachedServices: HashMap<String, Any?> = hashMapOf()
override fun <T : Any> resolve(
    regReference: RegistrationRef<T>,
    clazz: KClass<T>
): T? {
    val key = regReference.name
    var service: T? = previous(key)
    if (service != null) {
        return service
    } else {
        return save(regReference, key)
    }
}
```

USETECH

Скоупы (все названия условны)

ShareScope – отдает weak-ссылки

```
class ShareScope :ScopeProtocol {
var cachedServices:HashMap<String, BoxWeak>  = hashMapOf()

override fun <T : Any> resolve(
    regReference: RegistrationRef<T>,
    clazz: KClass<T>
): T? {
    val key = regReference.name
    var service: T? = previous(key)
    if (service != null) {
        return service
    }
    return save(regReference, key)
}
```

Скоупы (все названия условны)

ContainerScope – синглтоны
Скоуп обращается к основному контейнеру приложения

```
class ContainerScope : ScopeProtocol{
    private var cachedServices: HashMap<String, Any?> = hashMapOf()

    override fun <T : Any> resolve(
        regReference: RegistrationRef<T>,
        clazz: KClass<T>
    ): T? {
        val key = regReference.name
        var serviceReference = cachedServices[key] as? Reference<T>
        if (serviceReference != null) {
            return serviceReference.current
        } else {
            return saveSingleton(regReference, key)
        }
    }
```

Скоупы (все названия условны)

Reference<T> - холдит ссылку на экземпляр и фабрику

```kotlin
data class Reference<out T: Any>(
    val current: T?,
    val initor: () -> T?
)
```

Скоупы (все названия условны)

EntityScope – отдает каждый раз новый экземпляр

```
class EntityScope : ScopeProtocol {
    override fun <T : Any> resolve(
        regReference: RegistrationRef<T>,
        clazz: KClass<T>
    ): T? {
        return regReference.resolve()
    }
}
```

Скоупы (все названия условны)

GraphScope – сохраняет/извлекает из графа

```
class GraphScope :  ScopeProtocol {
    var resolutionDepth: Int = 0 //скоуп на 1 контейнер
    var graph:HashMap<String,Any>  = hashMapOf()

   override fun<T:Any> resolve(regReference: RegistrationRef<T>,
    clazz: KClass<T>):T? {
        val key = regReference.name
        resolutionDepth = resolutionDepth + 1
        defer {
            resolutionDepth = resolutionDepth - 1
            if (resolutionDepth == 0) {
                graph.clear()
            }
        }
        var service: T? = previous(key)
        if (service != null) {
            return service
        } else {
            val resolvedService = save(regReference, key)
            /*...магия зависимостей...*/
            return resolvedService
        }
    }
```

USETECH

Выглядит несложно,
поэтому займемся нюансами

Платформенные нюансы на KMM

Kotlin common

iOS:
Kotlin native

1. Управление потоками
2. Управление памятью (weak)
3. Инициализация (Kotlin Objective-C interop)
4. Соотношение типов (Kotlin Objective-C interop)
5. Синглтоны

Android:
Kotlin JVM

План действий

1. Инициализация (Kotlin Objective-C interop)
2. Соотношение типов (Kotlin Objective-C interop)
3. Управление памятью (weak)
4. Thread-safety (+ синглтоны +синхронизация)

Инициализация объектов

Factory -> generics

Swift generics vs Kotlin generics

```
//Swift
class Factory {
    func create<T>()->T {
        return T()
    }
}

//Kotlin
class Factory {
    fun <reified T : Any> create: T? {
    val primaryConstructor = T::class.constructors.find {
                                it.parameters.isEmpty() }
    return primaryConstructor?.call()
}
```

Инициализация объектов

Создание инстансов в Kotlin
generics есть только в Kotlin JVM

Kotlin Native???

JVM

constructors

All constructors declared in this class.

```
abstract val constructors: Collection<KFunction<T>>
```

Инициализация объектов

Создание инстансов в Kotlin
generics есть только в Kotlin JVM

Kotlin Native???

Common JVM JS Native Version 1.4

kotlin-stdlib / kotlin.reflect / KClass

KClass

```
interface KClass<T : Any> :
    KDeclarationContainer,
    KAnnotatedElement,
    KClassifier
```

Represents a class and provides introspection capabilities. Instances of this class are obtainable by the `::class` syntax. See the Kotlin language documentation for more information.

Parameters

`T` - the type of the class.

Properties

constructors

All constructors declared in this class.

```
abstract val constructors: Collection<KFunction<T>>
```

USETECH

Инициализация объектов

А в Kotlin Native – нет.

Ни constructors

Common JVM JS Native Version 1.4

kotlin-stdlib / kotlin.reflect / KClass

KClass

```
interface KClass<T : Any> :
    KDeclarationContainer,
    KAnnotatedElement,
    KClassifier
```

Represents a class and provides introspection capabilities. Instances of this class are obtainable by the `::class` syntax. See the Kotlin language documentation for more information.

Parameters

`T` - the type of the class.

Properties

qualifiedName

The fully qualified dot-separated name of the class, or `null` if the class is local or a class of an anonymous object.

```
abstract val qualifiedName: String?
```

Инициализация объектов

Ни createInstance,
ни primaryConstructor

Common JVM JS Native Version 1.4

kotlin-stdlib / kotlin.reflect.full / createInstance

createInstance

1.1

```
fun <T : Any> KClass<T>.createInstance(): T
```

Creates a new instance of the class, calling a constructor which either has no parameters or all parameters of which are optional (see KParameter.isOptional). If there are no or many such constructors, an exception is thrown.

Common JVM JS Native Version 1.4

kotlin-stdlib / kotlin.reflect.full

Package kotlin.reflect.full

Extensions for Kotlin reflection provided by `kotlin-reflect` library.

Exceptions

Properties

Functions

USETECH

Инициализация объектов

Kotlin/Objective-c interop

```
fun getOriginalKotlinClass(objCClass: ObjCClass): KClass<*>? {
    val typeInfo = getTypeInfoForClass(objCClass.objcPtr())
    if (typeInfo.isNull()) return null

    return KClassImpl<Any>(typeInfo)
}

fun getOriginalKotlinClass(objCProtocol: ObjCProtocol): KClass<*>? {
    val typeInfo = getTypeInfoForProtocol(objCProtocol.objcPtr())
    if (typeInfo.isNull()) return null

    return KClassImpl<Any>(typeInfo)
```


2021 г.

Инициализация объектов

Kotlin Native позволяет получить инфо о класс KClass, но не создать экземпляр

```
fun createKotlin(className: String){
    val name = className// MoviesListViewModel::class.qualifiedName
    val clazz =  NSClassFromString(name)
    val kclass = clazz?.let { getOriginalKotlinClass(it) }

    kclass?.constructors //Не доступен в Kotlin Native!!!!
}
```

USETECH

Инициализация объектов

Поэтому потребуется хранить ссылку на инициализатор типа

```
//Kotlin
class Reference<T> {
   var name: String
   var initor: ()->T
}

class Factory {
//... Какой-то код

  fun create<T>():T? {
    val refByType = searchForRef(T::class.qualifiedName)
    return refByType?.initor()?.invoke()
  }
//... Какой-то код
}
```

И это не все про типы

Interop типов

Kotlin Native со стороны iOSapp оперирует ObjCClass

Kotlin JVM -> KClass<T>

```kotlin
package kotlinx.cinterop
import kotlin.native.*

interface ObjCObject
interface ObjCClass : ObjCObject
interface ObjCClassOf<T : ObjCObject> : ObjCClass
// TODO: T should be added to ObjCClass and all meta-classes instead.
typealias ObjCObjectMeta = ObjCClass

interface ObjCProtocol : ObjCObject

fun<T: Any> DIContainer.createType(type: ObjCClass): KClass<T>? {
    return getOriginalKotlinClass(type) as? KClass<T>
}
```

Interop типов

Kotlin Native: доступны только типы Foundation, UIKit, из commonMain

Про классы со стороны iOSApp Kotlin не знает

```kotlin
/**
 * If [objCClass] is a class generated to Objective-C header for Kotlin class,
 * returns [KClass] for that original Kotlin class.
 *
 * Otherwise returns `null`.
 */
fun getOriginalKotlinClass(objCClass: ObjCClass): KClass<*>? {
    val typeInfo = getTypeInfoForClass(objCClass.objcPtr())
    if (typeInfo.isNull()) return null

    return KClassImpl<Any>(typeInfo)
}
```

Interop типов

Class name со стороны iOSApp отличается от class name со стороны shared или androidApp

```
__attribute__((objc_subclassing_restricted))
__attribute__((swift_name("MoviesService")))
@interface SharedMoviesService : SharedBase
- (instancetype)initWithNetworkClient:(SharedNetworkClient * _Nullable)networkClient
__attribute__((swift_name("init(networkClient:)")))
__attribute__((objc_designated_initializer));


String(describing: MoviesService.self) //iOS "SharedMoviesService"
MoviesService::class.qualifiedName    //Android "MoviesService"
```

Interop типов

Восстановление через имя класса провалено

```
fun<T: Any> createType(type: String): KClass<T>? {
    val objcClass = NSClassFromString(type) //SharedMoviesService
    if (objcClass != null) {
        return getOriginalKotlinClass(objcClass) as? KClass<T>
    }
    return null //<--- такого класса в common main нет
}
```

Interop типов

Для работы со стороны iOSApp
все запросы через ObjCClass

```
fun<T: Any> DIContainer.createType(type: ObjCClass): KClass<T>? {
    return getOriginalKotlinClass(type) as? KClass<T>
}

fun<T: Any> DIContainer.register(type: ObjCClass,
scope: ScopeType, fabric: () -> T?) {
    val clazz = createType<T>(type)
    if (clazz != null) {
        resolver.register(clazz,scope,fabric)
    }
}

fun <T : Any> DIContainer.resolve(type: ObjCClass): T? {
    val clazz = createType<T>(type)
    if (clazz != null) {
        return resolver.resolve(clazz)
    }
    return null
}
```

Checked. Идем дальше

- ✓ Инициализация (Kotlin Objective-C interop)
- ✓ Соотношение типов (Kotlin Objective-C interop)
- Управление памятью (weak)
- Thread-safety (+ синглтоны +синхнронизация)

Настройка управления памяти

Strong/weak reference

Retain cycle

```
expect class WeakReference<T : Any>(referred: T) {
fun clear()
fun get(): T?
}


//Android
actual typealias WeakReference<T> = java.lang.ref.WeakReference<T>

//iOS
actual typealias WeakReference<T> = kotlin.native.ref.WeakReference<T>
```

USETECH

2021 г.

Настройка управления памяти

В скоупе храним в обертке

```
expect class WeakReference<T : Any>(referred: T) {
    fun clear()
    fun get(): T?
}

data class BoxWeak(var value: WeakReference<Any>?)
```

Checked. Идем дальше

- ✓ Инициализация (Kotlin Objective-C interop)
- ✓ Соотношение типов (Kotlin Objective-C interop)
- ✓ Управление памятью (weak)
- Thread-safety (+ синглтоны +синхнронизация)

Memory management

Можно и красоты добавить

```kotlin
class WeakValue<T:Any>(obj: T? = null): ReadWriteProperty<Any?, T?> {
    private var wref : WeakReference<T>?

    init {
        this.wref = obj?.let { WeakReference(it) }
    }

    override fun getValue(thisRef:Any? , property: KProperty<*>): T? {
        return wref?.get()
    }

    override fun setValue(thisRef: Any?, property: KProperty<*>, value: T?) {
        wref = value?.let { WeakReference(it) }
    }
}

//simply weakify
public fun <T:Any> weakify(obj: T? = null) = WeakValue(obj)
```

Теперь займемся thread-safety

Потокобезопасность Kotlin Native

Kotlin JVM concurrency != Kotlin Native concurrency

expect Dispatchers

Kotlin Native

iOS:
actual Dispatchers

Kotlin JVM

Android:
actual Dispatchers

Потокобезопасность Kotlin Native

Kotlin Native требует особого решения работы с объектами, используемыми разными потоками:

- Mutable == только в одном потоке
- Immutable == несколько потоков

В Kotlin JVM проблем с потоками и объектами нет

Потокобезопасность Kotlin Native

Изменяемый объект надо заморозить для передачи между потоками.

Замороженный объект не изменяем

1.3

```
fun <T> T.freeze(): T
```

Freezes object subgraph reachable from this object. Frozen objects can be freely shared between threads/workers.

Exceptions

`FreezingException` - if freezing is not possible

https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.native.concurrent/freeze.html

Потокобезопасность Kotlin Native

Заморозить блок можно так…

```
actual val defaultDispatcher: CoroutineContext = MainDispatcher

actual  val uiDispatcher: CoroutineContext = MainDispatcher

private object MainDispatcher: CoroutineDispatcher() {
    override fun dispatch(context: CoroutineContext, block: Runnable) {
        try{
            block.run().freeze()
        }catch (err: Throwable){
            throw err
        }
    }
}
```

USETECH

Потокобезопасность Kotlin Native

… или так

```
actual fun callFreeze(callback: (Response) -> Unit) {
        val block = {
            callback(Response("from ios").freeze())
        }
        block.freeze()
        dispatch_async(
            queue = dispatch_get_global_queue(
                DISPATCH_QUEUE_PRIORITY_DEFAULT.toLong(),
                0.toULong()
            ),
            block = block
        )
    }
```

… но это еще не все

Дорабатываем синглтоны

Синглтоны Kotlin Native frozen по умолчанию

```
Uncaught Kotlin exception: kotlin.native.concurrent.FreezingException:
freezing of com.example.first_kmm_app.shared.network.NetworkService.Companion
@2ff1028 has failed,
first blocker is HttpClient[io.ktor.client.engine.ios.IosClientEngine

....
```

Дорабатываем синглтоны

Синглтоны Kotlin Native
frozen по умолчанию

Надо использовать @ThreadLocal

```
* Marks a top level property with a backing field or an object as thread local.
* The object remains mutable and it is possible to change its state,
* but every thread will have a distinct copy of this object,
* so changes in one thread are not reflected in another.
*
* The annotation has effect only in Kotlin/Native platform.
*
* PLEASE NOTE THAT THIS ANNOTATION MAY GO AWAY IN UPCOMING RELEASES.

public expect annotation class ThreadLocal()
```

Дорабатываем синглтоны

Пробуем написать сами обработку ThreadLocal без аннотации

Вдохновляемся https://github.com/ktorio/ktor

```
public actual fun <T : Any> threadLocal(value: T): ReadOnlyProperty<Any, T?> {
   //...
    return object : ReadOnlyProperty<Any, T?> {
        init {
            freeze()
        }
//...
    }
}
```

Дорабатываем синглтоны

Freeze exception вернулся

Без freeze тоже не работает

```
Uncaught Kotlin exception: kotlin.native.concurrent.FreezingException:
freezing of com.example.first_kmm_app.shared.network.NetworkService.Companion
@2ff1028 has failed,
first blocker is HttpClient[io.ktor.client.engine.ios.IosClientEngine

....
```

Дорабатываем синглтоны

@ThreadLocal -> companion object / object

Обработка спрятана где-то в глубинах LLVM Kotlin Native (не будем трогать)

```
//LLVMDeclarations.kt
private fun createSingletonDeclarations(irClass: IrClass): SingletonLlvmDeclarations? {
        if (irClass.isUnit()) {
            return null
        }

        val storageKind = irClass.storageKind(context)
        val threadLocal = storageKind == ObjectStorageKind.THREAD_LOCAL
    ///...
        return SingletonLlvmDeclarations(instanceAddress)
}
```

https://github.com/JetBrains/kotlin-native/

USETECH

Дорабатываем синглтоны

@ThreadLocal -> companion object / object

Синглтоном может быть любой класс -> прячем в Wrapper

```
@ThreadLocal
public object SingletonWrapper : Wrapper{
    override fun <T : Any> make(initor: () -> T): Reference<T> {
        val value = initor.invoke()
        return Reference(value)
    }
}
```

Дорабатываем синглтоны

Подключаем к скоупу

```
class ContainerScope : ScopeProtocol{

    override fun <T : Any> resolve(/*...*/): T? {
        //...
        val singleton = singleton(regReference)
        //...
        return singleton?.current
    }

    fun <T: Any> singleton(reference: RegistrationRef<T>):Reference<T>? {
        return Singleton.make(reference)
    }
}
```

Не проще ли было бы решить на уровне контейнера?

Решение на уровне контейнера

@ThreadLocal на уровне контейнера подойдет для случая 1 Container без конфигурации

```
//Единый контейнер
class Container {
    @ThreadLocal
    companion object {
        val Factory = Container()
    }
}
```

Решение на уровне контейнера

Если же будет использоваться создание static на уровне native, то нет

```
class ConfiguratorApp {
    val container = Container()
}

//iOS SceneDelegate -- тут не поможет
static var configurator: ConfiguratorApp {
    return  ConfiguratorApp()
}
```

USETECH

Забыли синхронизацию

Синхронизация

Разное понимание у Kotlin Native/Kotlin JVM
Реализуем через expect/actual

```
expect inline fun <R> trySynchronized(lock: Any?, block: () -> R): R

//iOS
actual inline fun <R> trySynchronized(lock: Any?, block: () -> R): R {
    if (lock != null) {
        objc_sync_enter(lock)
        val result = block()
        objc_sync_exit(lock)
        return result
    } else {
        return block()
    }
}

//Android
actual inline fun <R> trySynchronized(lock: Any?, block: () -> R): R =
    if (lock == null)
        block()
    else
        kotlin.synchronized(lock, block)
```

Синхронизация

Подключаем в скоупы для записи/чтения

```
trySynchronized(_lock) {
    var service: T? = previous(key)
    if (service != null) {
        return service
    } else {
        return save(regReference, key)
    }
}
```

Done!

- ✓ Инициализация (Kotlin Objective-C interop)
- ✓ Соотношение типов (Kotlin Objective-C interop)
- ✓ Управление памятью (weak)
- ✓ Thread-safety (+ синглтоны +синхронизация)

… немного сахара для удобства

USETECH

Kotlin delegate properties

Container.resolve ->
ReadOnlyProperty<Any?,T?>

```
class CachedValue<T:Any>(private val clazz: KClass<T>,
private var container : Container  = Container.Factory):
    ReadOnlyProperty<Any?, T?> {

    override fun getValue(thisRef:Any? , property: KProperty<*>): T? {
        return container.resolve(clazz)
    }

}

inline fun <reified T:Any> cached(container: Container = Container.Factory)
= CachedValue(T::class,container)

//call
val service: MoviesService? by cached()
```

Kotlin delegate properties

Container.register -> ReadWriterProperty<Any?,T?>

```
class CachedValue<T:Any>(private val clazz: KClass<T>,
                         private val container : Container  = Container.Factory,
                         private val fabric: () -> T?):
    ReadWriteProperty<Any?, T?> {
  /*..*/

    override fun setValue(thisRef: Any?, property: KProperty<*>, value: T?) {
        container.register(clazz, ScopeType.Cached,fabric)
    }
}

//call
var service: MoviesService? by cached { MoviesService(NetworkClient()) }
```

Аннотации для подключения зависимостей

Annotation processing -> JVM

KSP -> Common
https://github.com/google/ksp

```
class CommonAnnotationProcessor : SymbolProcessor() {
    //...
    val functions = mutableListOf<String>()
    val visitor = FindFunctionsVisitor()

    override fun process(resolver: Resolver) {
        //какая-то обработка
    }

    inner class FindFunctionsVisitor : KSVisitorVoid() {
     //... код какой-то
    }

}
```

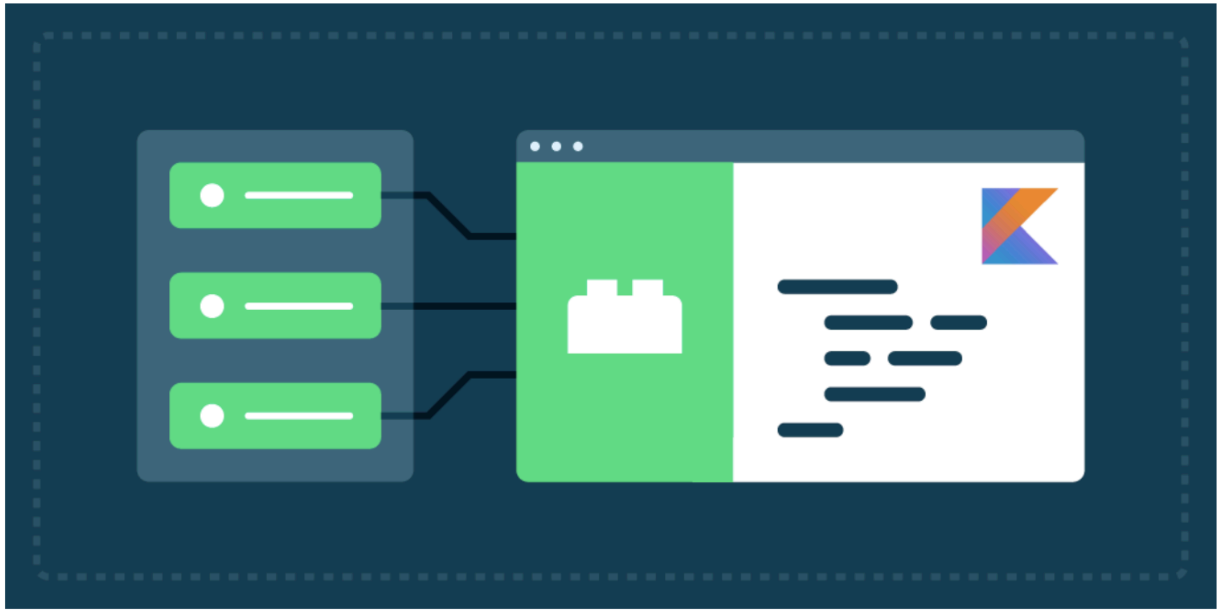
Аннотации для подключения зависимостей

KSP заявлен как замена KAPT

Announcing Kotlin Symbol Processing (KSP) Alpha

10 February 2021

Posted by Ting-Yuan Huang, Software Engineer and David Winer, Product Manager



Today we are excited to announce the alpha of **Kotlin Symbol Processing** (KSP), an all-new tool for building lightweight compiler plugins in Kotlin. KSP offers similar functionality to KAPT, however it's up to 2x faster, offers direct access to Kotlin compiler features, and is being developed with multiplatform compatibility in mind.

USETECH

2021 г.

Аннотации на стороне Android

На стороне Android можно использовать
@Inject (javax.inject.Inject)

Для подключения Container – свой
AnnotationProcessor

```
@AutoService(Processor::class)
class DIProcess : BaseProcessor() {
   override fun getSupportedAnnotationTypes(): MutableSet<String> {
        return mutableSetOf(
            Injected::class.java.name
        )
    }

    override fun process(
        annotations: MutableSet<out TypeElement>?,
        roundEnv: RoundEnvironment?
    ): Boolean {
//...
}

annotation class Injected
```

Property wrappers на стороне iOS

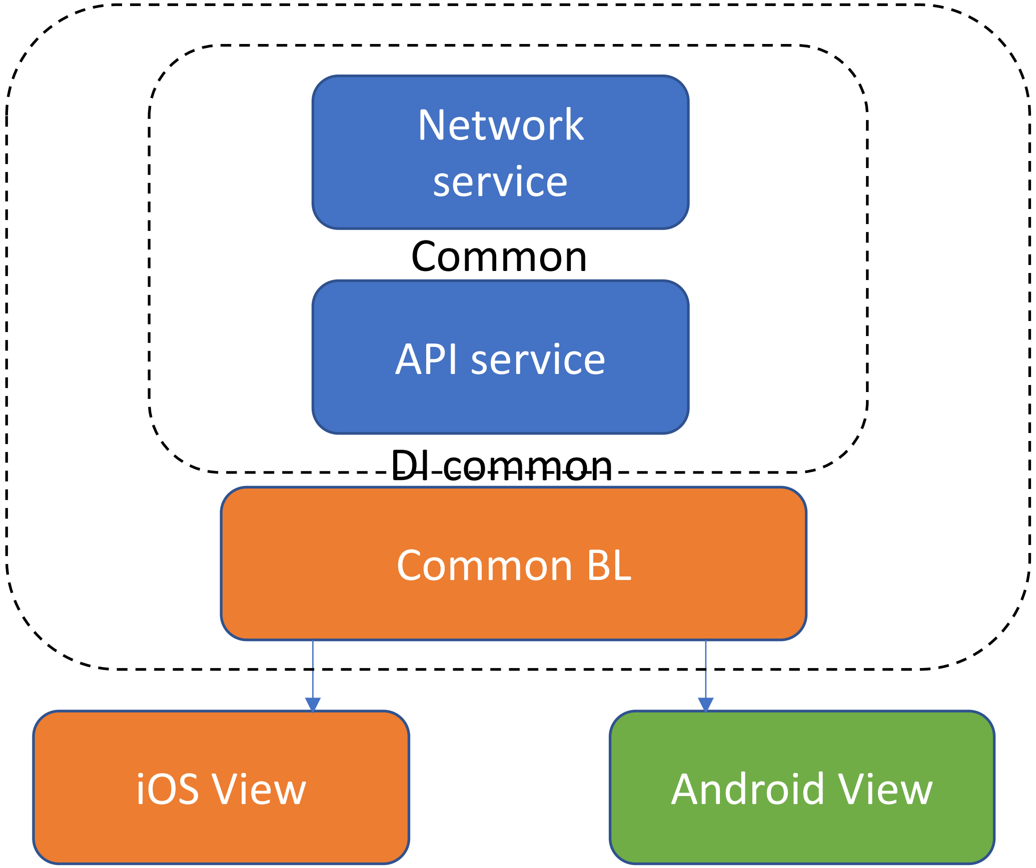
На стороне iOS прячем обращение к Common DI в property wrapper

```
@propertyWrapper
struct Injected<T:AnyObject> {
    private var service: T?
    public  var container: DIManager? = nil
    public init() {}
    public init(container: DIManager?) {
        self.container = container
    }
    public var wrappedValue: T? {
        mutating get {
            if self.service == nil {
                self.service = container?.resolve(clazz: T.self) as? T
            }
            return service
        }
        mutating set { service = newValue  }
    }
    public var projectedValue: Injected<T> {
        get { return self }
        mutating set { self = newValue }
    }
}
```

Осталось соединить с View

Связь View с Common

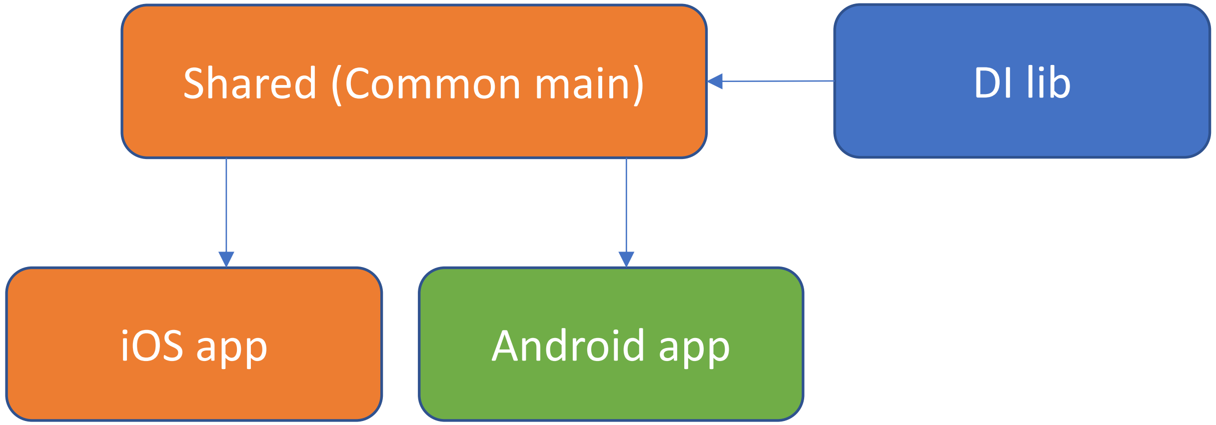
iOS View?
Android View?



Связь View с Common

DI multiplatform library -> Common Main

https://github.com/anioutkazharkova/di-multiplatform-lib



Инициализация на стороне натива

Прокинем вызовы DI-библиотеки в commonMain

https://github.com/anioutkazharkova/di-multiplatform-lib

```
class DIManager {
    //DIContainer из di-multiplatform-lib
    val appContainer: DIContainer by lazy { DIContainer() }

    fun<T:Any> resolve(type: KClass<T>):Any? {
        return appContainer.resolve(type)
    }

    fun<T:Any> addToScope(scope: ScopeType, type: KClass<T>, fabric: ()->T?) {
        appContainer.register(type, ScopeType.Graph,fabric)
    }
    /*...*/
}
```

Инициализация на стороне натива

Общая точка входа

https://github.com/anioutkazharkova/di-multiplatform-lib

```
class ConfigurationApp {
val appContainer: DIManager = DIManager() //не используем синглтоны из принципа

   init {
       setup()
   }

    fun setup() {
        //Сервисы будут синглтонами
        appContainer.addToScope(ScopeType.Container,NetworkClient::class) {
            NetworkClient()
        }
        appContainer.addToScope(ScopeType.Container,MoviesService::class) {
            val nc = appContainer.resolve<NetworkClient>(NetworkClient::class)
            as? NetworkClient
            MoviesService(nc)
        }
    }
}
```

USETECH

Инициализация на стороне натива

На стороне iOSapp и AndroidApp делаем точку входа

https://github.com/anioutkazharkova/di-multiplatform-lib

```
import shared

class Util{
    static var shared = Util()
    private lazy var config: ConfigurationApp = {
        return ConfigurationApp()
    }()

    lazy var container: DIManager = {
        return config.appContainer
    }()
}
```

Инициализация на стороне натива

На стороне iOSapp и AndroidApp делаем точку входа

https://github.com/anioutkazharkova/di-multiplatform-lib

```
import com.azharkova.sampledi.shared.config.ConfigurationApp

@HiltAndroidApp
class App : Application() {
    companion object {
        val config = ConfigurationApp()
        val container = config.appContainer
    }
/*...*/
}
```

Связь View с Common

Сделаем фабрику конфигураторов
для BL модуля кода

```
interface IConfigurator {
    fun create(view: IView):IInteractor?
}

class ModuleConfig {
    fun config(view: IView):IInteractor? {
        if (view is IMoviesListView) {
            return MoviesListConfigurator.instance.create(view)
        }
        /*....*/
        return null
    }
}
```

Связь View с Common

Внутрь конфигуратора надо прокинуть ссылку на View

```
class MoviesListConfigurator : IConfigurator {
    /*...*/
    override fun create(view: IView): IInteractor? {
        val interactor = MoviesListInteractor()
        val presenter = MoviesListPresenter()
        interactor.presenter = presenter
        presenter.view = view as? IMoviesListView
        return interactor
    }
}
```

Связь View с Common

На стороне iOS прячем в propertyWrapper

```swift
@propertyWrapper
struct InjectedInView<T:AnyObject> {
    //...
    public weak var view: IView?

    public var wrappedValue: T? {
        mutating get {
            if self.service == nil {
                guard let view = self.view else {return service}
                self.service = container.resolve(view: view) as? T
            }
            return service
        }
        mutating set { service = newValue  }
    }
    //...
}
```

USETECH

2021 г.

Связь View с Common

На стороне iOS прячем в propertyWrapper

```
class MoviesVC : UIViewController, IMoviesListView {
    //Нельзя капчить self здесь
    //@InjectedInView(self)  var interactor: IMoviesListInteractor?

  @InjectedInView  var interactor: IMoviesListInteractor?

  override func viewDidLoad() {
        super.viewDidLoad()
        $interactor.view = self
        //...
  }
  //...
}
```

Связь View с Common

На стороне Android – Kotlin delegate property

```kotlin
class Interactors<T:Any>(private val clazz: KClass<T>,
private val view: IView,
private val container: DIManager):
    ReadWriteProperty<Any?, T?> {
    private var wref : WeakReference<T>? = null
    /*...*/
     override fun getValue(thisRef: Any?, property: KProperty<*>): T? {
        if (wref == null) {
            wref = WeakReference(container.resolve(view) as? T)
        }
        return wref?.get()
    }
}
```

Связь View с Common

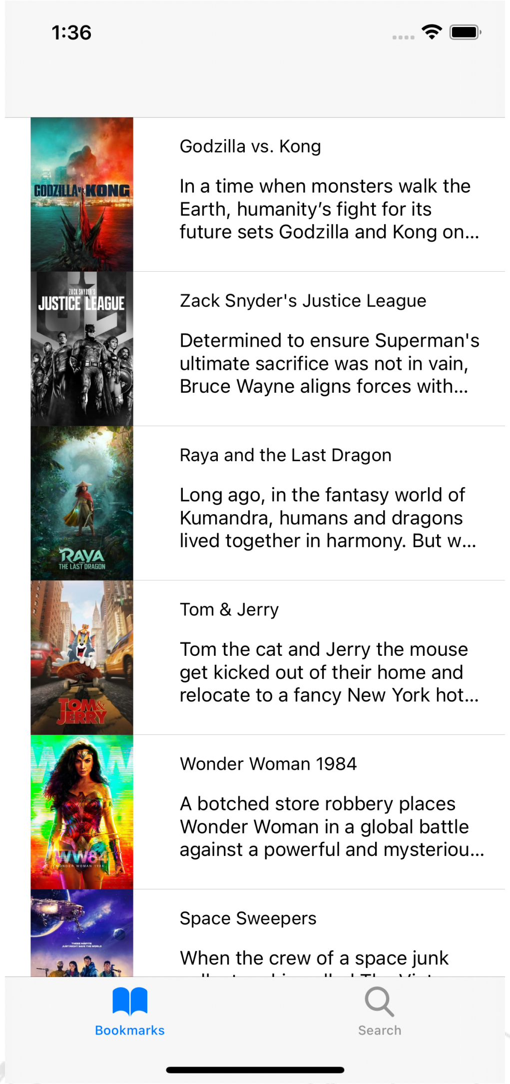
На стороне Android – Kotlin delegate property

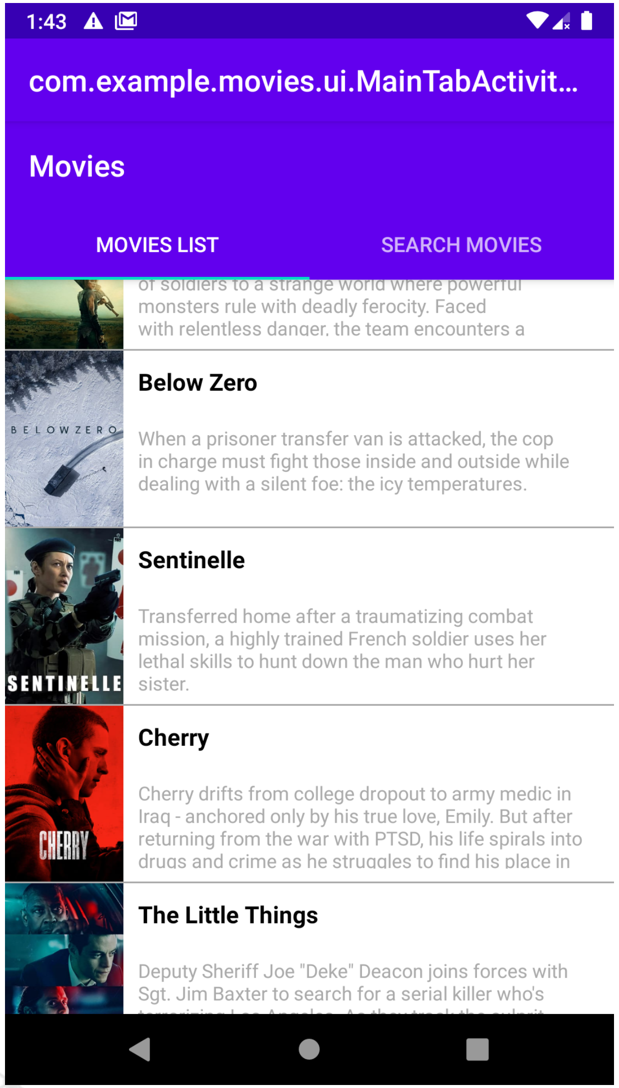
```
//simply call
public inline fun <reified T:Any> interactors(view: IView)
= Interactors(T::class, view)


class MoviesListFragment : Fragment(), IMoviesListView {
  val interactor: IMoviesListInteractor? by interactors(this)
  /*...*/
}
```

USETECH

Summary. Пример





Sources

https://github.com/anioutkazharkova/di_multiplatform_sample

https://github.com/anioutkazharkova/di-multiplatform-lib

https://kotlinlang.org/docs/native-overview.html

https://github.com/JetBrains/kotlin-native

Спасибо за внимание!

@anioutkajarkova

azharkova
prettygeeknotes

USETECH