Міністерство освіти і науки України
НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ «КИЄВО-МОГИЛЯНСЬКА АКАДЕМІЯ»
Кафедра інформатики факультету інформатики

Оптимізація роботи з залежностями у проєктах Kotlin Multiplatform Mobile за допомогою Swift Package

Текстова частина до кваліфікаційної роботи
за спеціальністю “Інженерія програмного забезпечення”

Керівник кваліфікаційної роботи старший викладач Франків О. О.

____

(підпис)

“___” ______________ 2024 р.

Виконав студент Джос О. К.

“___” ______________ 2024 р.

Київ 2024

Міністерство освіти і науки України
НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ «КИЄВО-МОГИЛЯНСЬКА АКАДЕМІЯ»
Кафедра інформатики факультету інформатики

ЗАТВЕРДЖУЮ

Зав. кафедри інформатики,
доцент, кандидат наук
____________ Гороховський С.С
(підпис)
"____" __________ 2024 р.

ІНДИВІДУАЛЬНЕ ЗАВДАННЯ

на кваліфікаційну роботу

студенту Джос Олексій 4-го курсу факультету інформатики

ТЕМА: Оптимізація роботи з залежностями у проєктах Kotlin Multiplatform Mobile за допомогою Swift Package

Зміст ТЧ до курсової роботи:

Індивідуальне завдання
Вступ
1. Огляд менеджерів залежностей для iOS розробки
2. Способи інтеграції КММ в iOS
3. Розробка інструменту для інтеграції SPM в KMM
Висновок
Список використаних джерел
Додатки

Дата видачі "___" _________ 2024 р.
Керівник _________________ Завдання отримано __________

Календарний План Виконання Кваліфікаційної Роботи

Тема: Оптимізація роботи з залежностями у проєктах Kotlin Multiplatform Mobile за допомогою Swift Package

№ з/п	ПЕРЕЛІК РОБОТ	Термін Виконання	Примітка
1	Вибір теми	Вересень – Початок жовтня 2023	
2	Затвердження теми з керівником	Середина жовтня - Початок листопада 2023	
3	Дослідження статичного та динамічного компонування	Листопад 2023 – Грудень 2023	
4	Написання Gradle плагіну для реалізації SPM у Kotlin Multiplatform	Грудень 2023 – Лютий 2024	
5	Аналіз отриманого результату та написання доповіді	Березень 2024 – Травень 2024	
6	Оформлення пояснювальної роботи та презентації	10.05.2024 – 13.05.2024	
7	Корегування з урахуванням попереднього захисту	15.05.2024	
8	Подання на кафедру задля перевірки вимогам доброчесності	17.05.2024	
9	Захист дипломної роботи	27.05.2024	

Студент Джос О. К. _________________

Керівник Франків О.О. _________________

“_____” ________________ 2024 р.

ЗМІСТ

АНОТАЦІЯ 5

СКОРОЧЕННЯ ТА УМОВНІ ПОЗНАЧЕННЯ 6

ВСТУП 7

1. ОГЛЯД МЕНЕДЖЕРІВ ЗАЛЕЖНОСТЕЙ ДЛЯ iOS РОЗРОБКИ 9

1.1. Вміст iOS додатків 9

1.2. Ручний спосіб підключення залежностей 14

1.3. Carthage 15

1.4. CocoaPods 16

1.5. Swift Package Manager 17

1.6. Kotlin Multiplatform Mobile 18

2. СПОСОБИ ІНТЕГРАЦІЇ KMM в iOS 20

2.1. Огляд впровадження KMM на прикладі CocoaPods 20

2.2. Дослідження теоретичної реалізації інтеграції KMM в SPM 26

2.3. Варіант інтеграції SPM у KMM з використанням XCode 30

3. РОЗРОБКА ІНСТРУМЕНТУ ДЛЯ ІНТЕГРАЦІЇ SPM В KMM 32

3.1. Створення Gradle плагіну 32

3.2. Недоліки та можливі покращення 40

3.3. Результати 42

ВИСНОВОК 46

СПИСОК ЛІТЕРАТУРИ 47

ДОДАТКИ 49

АНОТАЦІЯ

Мета кваліфікаційної роботи дослідити, проаналізувати особливості використання різноманітних способів роботи з залежностями на iOS та реалізувати програмний продукт задля оптимізації з залежностями в iOS додатках під час використання Kotlin Multiplatform Mobile за допомогою Swift Package Manager. Дана робота розкриває особливості розробки та використання бібліотек та пакетних менеджерів для iOS розробки зважаючи на особливості статичного та динамічного компонування.

СКОРОЧЕННЯ ТА УМОВНІ ПОЗНАЧЕННЯ

KMP – Kotlin Multiplatform Platform

KMM - Kotlin Multiplatform Mobile

SPM – Swift Package Manager

XCode – рекомендований IDE для роботи з iOS додатками

БД – база даних

Таргет – синонім до слова платформа. Наприклад таргет Android або iOS

Вендор – виробник, постачальник послуг

ОС – операційна система

CInterop – перетворення сі-подібного коду в Kotlin інтерфейси

Kotlin DSL – спеціальний опис властивостей в декларативному стилі

ВСТУП

На сьогоднішній день широко починають використовувати технологію Kotlin Multiplatform [1] для написання спільного коду між деякими платформами, як наприклад Android та iOS. Головна відмінність KMP від крос платформної розробки полягає у можливості розробникам обирати, яку частину коду ділити між всіма таргетами, а яку писати використовуючи нативні інструменти розробки (наприклад Swift для iOS та Kotlin для Android).

З іншої сторони, у сфері iOS починає активно розвиватися підтримка Swift Package Manager – пакетному менеджеру від компанії Apple, який має зручну інтеграцію з інструментами розробки та менший розмір додатку через особливості компонування. Найпопулярніший нині CocoaPods для роботи з залежностями в iOS світі під час надає великий спектр роботи з бібліотеками, проте відсутність підтримки зі сторони вендору та великий розмір вихідного додатку або швидкості запуску стає великими мінусом під час його використання.

На даний момент, розробка на Kotlin Multiplatform, використовує CocoaPods для інтеграції з Apple світом, проте даний варіант може не підходити командам, які вже використовують SPM для своїх потреб, або не хочуть залежати від інструменту розробленим не компанією Apple.

Зважаючи на обмовлені проблематики, тема моєї кваліфікаційної роботи являє собою оптимізацію роботи з залежностями у iOS додатках написаних на Kotlin Multiplatform використовуючи Swift Package Manager.

Метою роботи є дослідження використання різноманітних підходів для додавання залежностей у Kotlin Multiplatform у iOS частині та розробка програмного інструменту для інтеграції Swift Package Manager для оптимізації роботи з залежностями.

Об’єктом роботи є оцінка нинішніх варіантів підключення iOS залежності з Kotlin Multiplatform Mobile з використанням Regular Framework або CocoaPods з урахуванням плюсів та мінусів кожного рішення враховуючи потреби.

Предмет дослідження являє собою порівняння статичного та динамічного компонування в сфері iOS розробки з урахуванням плюсів та мінусів кожного підходу та особливостями, які використовують пакетні менеджери для інтеграції.

Структура дослідження поділена на логічні блоки від розуміння того, з чого складаються iOS додатку до опису проблематики інтеграції KMP у iOS додатки та вирішення поставлених проблем.

Перший розділ розкриває вміст iOS додатків та перевикористання коду між іншими проєктами, аби зрозуміти специфіку роботи з залежностями, які вони бувають, чим відрізняються, які переваги та недоліки кожного варіанту – детально розглядаються такі терміни як бібліотека, framework та пакунок. Також оглянуто менеджери залежностей для iOS проєктів, які надають розробникам інструменти задля спрощення підключення сторонніх бібліотеки. Описується прогрес від ручного завантаження у папку додатку, використання напівавтоматизації через Carthage до найбільш популярного CocoaPods та найновішого від Apple – Swift Package Manager.

Другий розділ присвячений опису проблематики використання CocoaPods та Regular Framework у проєктах з використанням Kotlin Multiplatform Mobile – цей розділ присвячений особливостям роботи використання. Дана частина розкриває питання технічних викликів, пов’язаних з кожним типом та відповідних шляхів вирішення.

У третьому розділі описуються вирішення проблем зазначених у попередніх частинах з використання практичного досвіду реалізації Gradle плагіну для використання Swift Package у проєктах написаних на Kotlin Multiplatform та відповідна демонстрація та інтеграція у існуючі проєкти.

1. ОГЛЯД МЕНЕДЖЕРІВ ЗАЛЕЖНОСТЕЙ ДЛЯ iOS РОЗРОБКИ

1.1. Вміст iOS додатків

Звичайний iOS додаток обов`язково складається з певного вихідного коду, який пише розробник та додаткових компонентів як ресурси, Bundle, файли конфігурації, тести або другорядні розширення, як наприклад, віджети.

Ресурси складаються зі зберігання картинок, шрифтів, різною локалізацією в залежності від обраної мови. Зазвичай використовують Assert Catalog для зручного менеджменту та додаткової інтеграції з XCode. Файли конфігурації складаються з описаних властивостей в залежності від середовища та етапу збірки. Info.plist спеціалізований файл, де описується версія додатку, його унікальний ідентифікатор та інша метаінформація про можливості додатку. Важливим атрибутом в iOS додатках є Bundle [2] – спеціальна структура файлів в кожному додатку, які слугують для зручного отримання доступу до ресурсів, локалізації, заголовних файлів, не знаючи внутрішню ієрархію під час роботи застосунку.

Вихідний код описує логіку та функціональність вашого додатку. Тут можуть бути описані вся структура від початкового запуску додатку до примусового закриття, та будь яка реалізація задля виконання технічного завдання. Нині рекомендованою мовою для написання коду є мова програмування Swift, проте досі існує застарілий код на Objective-C, або використанні інші мови програмування для крос-платформної розробки.

Код, який включається в результуючий iOS додаток може бути не тільки написаний розробником, а і сторонніми людьми, оскільки певну частину функціоналу можна перевикористовувати між проєктами та підключати як зовнішню залежність. Це може бути гарним варіантом, аби не писати кожного разу ідентичний код, обгортку або абстрагувати складну логіку під зручний інтерфейс. Для використання таких можливостей, у iOS світі використовують такі інструменти як бібліотеки, Framework [3] та пакунки [4]. Їх головна мета

стандартизувати методи доставки зовнішнього коду для використання в iOS проєктах та уніфікувати процес підключення в результуючий додаток компонувальником, [5] який відповідає за об`єднання всіх скомпільованих об’єктних файлів та зовнішньо підключених функцій в один виконуваний файл.

Розглянемо детальніше бібліотеки – вони складаються просто з набору коду без ресурсів для конкретної платформи, тобто під час компіляції виникає потреба бути зібраним під різні таргети та архітектури, наприклад, iOS x86 (для симулятору) або MacOS серії M (для нової архітектури ARM). Їх поділяють на два типи – статичні та динамічні, кожна з них має свої особливості з відповідними перевагами та недоліками для певного випадку.

Статична бібліотека [6] це набір коду у файлі з розширенням *.a, яка визначається ознакою – її код завантажується у пам'ять під час виконання, тобто відбувається повне копіювання підключеної залежності. Таким чином, розмір результуючого додатку значно зростає навіть якщо ми використовуємо всього одну структуру або функцію.

Оскільки ми маємо контроль над всім кодом під час використання статичної бібліотеки, можемо, проаналізувати та виконати процес Code Stripping або Shrink, який на етапі компіляції почистить наш проєкт від не використаного коду, задля оптимізації простору. Існують різні рівні коду – він видалення не використаних змінних до спрощення або змін у класах, структурах.

Будь-які зміни у статичній бібліотеці призводять до потреби повторного використання статичного компонувальника під час компіляції, аби отримати нові зміни в проєкті та почати використовувати їх, тому що компонувальник в під час компіляції виконує зв’язування та оновлює символи в результуючому об’єктному файлі.

Ще однією перевагою статичних бібліотек є швидкість запуску проєкту [6]. Під час старту додатку, операційна система завантажує символи з нашого виконуваного файлу використовуючи Page Fault [7]. Через те, що наша бібліотека знаходиться поруч з додатком, то цей процес використовує менше оперативної

пам’яті, ніж у варіанті, коли бібліотека знаходиться у декількох двійкових файлах.

Динамічна бібліотека – файл з розширенням *.dylib [8], в свою чергу, завантажується у пам'ять за потребою від додатку та зберігається на стеку, тому розмір додатку змінюється не суттєво при підключенні залежності, оскільки підключений код знаходиться поза зоною виконувального файлу та завантажується під час запуску додатку.

Під час старту проєкту, інструмент dyld завантажить всі потрібні динамічні бібліотеки, що може займати суттєвий час – чим більше залежностей ми маємо, тим довше відбувається старт проєкту. Apple з кожним релізом нової iOS покращує, оптимізує цей процес та наводить свої рекомендації по пришвидшенню запуску додатку, наприклад, відбувається завантаження у той момент, коли потрібна функціональність потрібна та рекомендує не використовувати більше ніж 6 динамічних залежностей.

Через те, що динамічні бібліотеки підключаються в процесі виконання та код яких не потрапляє під час компіляції в файл додатку, ми не маємо змоги перевірити, чи наданий код працює коректно. З іншого боку, це надає можливість не проводити повторну компіляцію коду нашого додатку, якщо оновилася залежність, оскільки новий код ніяким чином не впливає на виконуваний файл, за умови, що не змінилися виклики до самої бібліотеки.

Особливістю динамічних бібліотек в iOS є те, що Apple не дозволяє, окрім своїх рішень, завантажувати такі бібліотеки у систему, аби їх могли перевикористати інші додатки – вони існують тільки в межах пакунку додатку. Тобто у випадку, якщо ми маємо два додатки, які використовують однакову динамічну бібліотеку, вона буде завантажена двічі, оскільки повторно використати вже існуючу динамічні бібліотеку з іншого додатку, забороняє політика Apple.

Бібліотеки досить зручний та корисний механізм для можливості ділитися кодом або абстрагувати код, проте, їх перш початковий варіант не дуже підходить для iOS розробки, оскільки для мобільних додатків важливо мати

підтримку ресурсів, власних налаштувань тощо. Тому, було створено новий формат – Framework [9], файл з розширенням *.framework, який може містити в собі вихідний код, ресурси, динамічні бібліотеки та навіть інші фреймворки. Важливою відмінністю від бібліотек є наявність власного Bundle, за допомогою якого ми можемо доступатися до внутрішньої структури з власними налаштуваннями.

У такій варіацій бібліотеки, ми маємо можливість взаємодіяти з iOS додатками зважаючи на особливості мобільної платформи, проте ми не врахували підтримку різноманітних архітектури та платформ, наприклад MacOS (x84, arm) або iOS (simulator, x86, arm). У випадку з використанням Framework ми обмежені використанням однієї зв'язки архітектури та таргету, що накладало обмеження або на написання універсального коду, що не завжди доречно, або на включення у збірку декількох Framework під потрібні цільові таргети – тобто додавання декількох залежностей та вказування котрий компонувати для якої архітектури.

Для вирішення цієї проблеми створили fat або universal Framework, де за допомогою команди *lipo* створювався великий єдиний файл з усіх можливих варіацій архітектури, аби під час компіляції не думати про невідповідність, оскільки ми гарантуємо "універсальність" нашої залежності – буде працювати на будь якому пристрої. Вочевидь, такий підхід має недоліки як збільшення розміру та проблеми з сумісністю на перевірці в AppStore, тому що варіація для x86_64 не існує ля реальних пристроїв та повинна бути виключена під час доставки додатку кінцевим користувачам.

Наступним кроком було створення та впровадження XCFramework [10], який включає у себе структурований формат з включенням всіх таргетів, архітектури та маніфесту, які потребує додаток, а не створює гігантський фреймворк з усіма можливими таргетами - всі файли будуть упаковані у єдиний файл *.xcframework. Підключення та відповідне компонування виглядає як звичайне додавання залежності, проте на етапі компіляції, компілятор сам знайде

потрібний Framework у структурі в залежності від параметрів запуску, що спрощує роботу розробникам.

Даний перелік характеристик у фреймворків зарекомендував себе, що спричинило повсюдне використання у iOS проєктах, тобто де-факто стало стандартом в індустрії, через що, зіштовхнутися з чистими бібліотеками майже неможливо. Фреймворки вирішують проблему з перевикористання та структуризації, проте, розробники в такому варіанті поширюють готові до підключення залежності, а не вихідний код для організації коду.

Apple створили Swift Package [4] – універсальний пакунок, який може містить в собі бібліотеки, виконувані файли, ресурси. Збірка відбувається по однакому сценарію в незалежності від потреб – або це додаток, або окремий модуль, що дозволяє мати впевненість в ідентичному результаті. Головна властивість полягає у тому, що пакет дає можливість розбити проєкт на структуровані модулі за певною логікою, тобто дана варіація про те, як організувати код, отож надати готову бібліотеку для компонувальника, сховавши вихідний код – не вийде, що може бути мінусом під час розповсюдження для інших розробників.

Swift Package складається [4] з обов’язкового маніфесту Package.swift, де описується логіка підключення інших framework, бібліотеки або пакунку, шляхи до ресурсів, підтримувані таргети. Також, у пакунку наявний вихідний код, який реалізує конкретні вимоги до модуля. Пакет досить універсальний спосіб комбінувати модулі з тільки вихідного коду, або з додавання інших framework або пакетів. Для створення власного Swift Package можна використати команду *swift package init*, де створиться базова структура пакунку з мінімальним набором потрібної інформації для інтеграції у існуючі проєкти.

Тобто Swift Package, в порівнянні з framework або бібліотекою, відрізняється філософією використання. Універсальний пакет надає можливість рознести код по різним модулям та організувати код в залежності від потреб проєкту, що може пришвидшити збірку проєкту або в великих компаніях рознести відповідальність, аби не заважати один одному під час розробки.

1.2. Ручний спосіб підключення залежностей

Вочевидь, самий простий спосіб додавання залежності в проєкт, це завантаження бібліотеки чи framework власноруч та підключення у XCode прописуючи шлях до завантаженого файлу. Такий спосіб надає можливість швидко інтегрувати потрібну бібліотеку, мати повний контроль над версією, та уникнути додаткових витрат на дослідження сторонніх рішень для керування залежностями. Розробник сам несе відповідальність за роботу з бібліотеками та має можливість керувати ними власноруч. В такому варіанті, ми можемо особисто обрати який варіант зв`язування для кожної бібліотеки ми будемо використовувати, що може бути корисним, якщо ми розуміємо конкретні потреби, які пов’язанні з особливостями, чи то швидкого запуску , чи то розміру додатку.

Такий варіант підключення породжує незручність у командній роботі, оскільки кожен розробник повинен локально додавати на робочу машину залежність та прописувати індивідуальний шлях. Особливою потребою, може бути інтеграція для CI/CD, тому шо потрібно буде описувати скрипти для завантаження конкретної версії та додавання у налаштування збірки.

Важливим аспектом під час роботи з бібліотеками є їх оновлення, вихід нової версії тягне за собою процес ручного оновлення на кожного розробника. Для проєктів з великою кількістю залежностей буде витрачатися багато часу на налаштовування, а також, можливі, конфлікти між різними версіями бібліотек.

Аналізуючи плюси та мінуси ручного керування залежностями, можна вивести примітивну реалізацію простого менеджера, який би спростив описані нюанси використання та надавав зручну інтеграцію бібліотек. Вирішенням став би текстовий файл, у якому б були описані звідки брати залежність, якої версії, та bash скрипт, який би завантажував нові версії або брав з кешу, якщо немає нових. Даний варіант можна було б, як запускати одноразово для локальної розробки, якщо ми хочемо оновитися; так і покласти у фазу компіляції – Build Phase для CI/CD рішення.

1.3. Carthage

Carthage – примітивний децентралізований менеджер залежностей, який написаний повністю на мові Swift та використовується лише для опису потрібних залежностей та не модифікує проєкт, що залишає за розробниками повний контроль над підключенням. Важливо уточнити, що даний менеджер працює тільки з Framework та не підтримує бібліотеки та пакунки. За замовчуванням, залежності інтегруються як динамічний framework.

Для використання достатньо описати файл Cartfile з джерелом завантаження, версією framework та покласти поруч з XCode файлом. Завантаження відбувається за командою *carthage bootstrap* під час першого запуску, або *carthage update* для оновлення. Дані команди завантажать локально ресурси та зберуться у framework, якщо буде потрібно.

Оскільки Carthage не модифікує XCode проєкт, то виникає потреба у додаванні файлів власноруч, для цього існує два способи. Перший пропонує перенести всі framework з Carthage/Build в IDE, тобто за аналогією з ручним способом. Другий варіант це використання Run Script Phase під часу етапу компіляції. Для цього розробники Carthage надають команду *carthage copy-framework*, який на перший запуск проєкту додає дані framework у етап компіляції, та оновлює їх, якщо відбулися зміни у Carthage файлі (для цього існує Carthage.resolved який зберігає хеш суму).

З переваг такого рішення, це відсутність накладних витрат з викачування вручну залежностей та інкапсуляція цього процесу, аби розробники та CI/CD мали однакову поведінку під час збірки проєкту. До недоліків Carthage можна віднести складність підключення в проєкт, оскільки ручне або напів - автоматичне з прописуванням команд для запуску; потреба в ручному видалені залежності, які не потрібні.

Даний менеджер вирішує проблему ручного завантаження залежностей, проте оновлення, видалення, проте інтеграція у XCode залишає бажати кращого, що призвело до малої підтримки у спільноті.

1.4. CocoaPods

CocoaPods – пакетний менеджер написаний у вигляді Ruby бібліотеки. У відмінності від Carthage, він надає функціонал інтеграції з XCode для зручного використання. Така можливість надається через використання Workspace – колекції проєктів поєднаних разом у IDE.

Додавання залежностей відбувається через опис файлу Podfile, де вказується назва бібліотеки та опціонально назва. Після запуску команди *pod init*, CocoaPods згенерує .xcworkspace, в якому будуть поєднані два .xcodeproj – ваш особистий та "заготовка", куди будуть завантажуватися залежності. Таким чином, ми будемо мати повний контроль над своїм проєктом.

CocoaPods працює і з бібліотеками, і з фреймворками (є ключове слово *use_frameworks!*), проте на сьогоднішній момент, бібліотеками ніхто не користується, отож команда пакетного менеджеру та розробники залежностей наполегливо рекомендують використовувати саме фреймворки.

Щоб додати нову залежність, достатньо оновити файл Podfile з потрібною версією та прописати команду *pod update*. Далі менеджер залежностей, або візьме з кешу, або завантажить з репозиторію та покладе у папку Pods нашого другого .xcodeproj та згенерує .xcscheme для опису інформації про збірку фреймворку. Останнім кроком буде редагування project.pbxproj, який фактично описує, що кожний завантажений Pod є продуктом в Xcode. На етапі компіляції кожна залежність збирається у динамічну бібліотеку та підключається у основний проєкт. Вочевидь, тут ми не маємо можливості контролювати варіант компонування, отож маємо справу з динамічним компонування та його відповідними недоліками.

Оскільки CocoaPods завантажує вихідний код у XCode проєкт, тому ми маємо зручний доступ до перегляду потрібних інтерфейсів, функції та можливості "провалюватися" у код, щоб переглянути внутрішні реалізації. Дана можливість присутня через використання SourceKit-LSP, яка аналізує весь висхідний проєкт та будує дерево використання.

Недоліком використання даного пакетного менеджеру може бути відсутність підтримки потрібної бібліотеці, оскільки всі залежності повинні мати файл PodSpec з детальним описом назви, включених файлів, підтримуваних версій ОС та опубліковані в Pod репозиторій. Відповідно, якщо розробник бібліотеки не додав мета файл та не завантажив у реєстр, то використовувати у своєму проєкті з CocoaPods не можна буде. Проте даний недолік може бути опущений, оскільки CocoaPods найпопулярніший пакетний менеджер і складно не знайти залежність без підтримки даного рішення.

Використання CocoaPods вирішує такі проблеми як автоматичне підключення залежностей у проєкт та мінімальна інтеграція з Xcode – можемо налаштовувати бібліотеки, дивитися висхідний код всередині IDE та запуск проєкту під час етапів компіляції автоматично збирає на кожну залежність свою динамічну бібліотеку, компонує та збирає у результуючий додаток.

З оглядом на можливості CocoaPods, було б добре мати більш зручний метод додавання нових залежностей, оскільки нині це відбувається дописуванням руками нових залежностей у Podfile та подальшим запусканням команди у командному рядку для оновлення бібліотек, що хотілося б мати більш зручну інтеграцію з IDE. До того ж, ідея використання динамічного компонування залежностей може не підійти через вимоги до програмного забезпечення, як наприклад, швидкість завантаження додатку.

1.5. Swift Package Manager

З недавніх часів, Apple представила свій пакетний менеджер Swift Package Manager на базі Swift Package. Головною перевагою якого є тісна інтеграція з XCode, що дозволяє у середовищі робити потрібні маніпуляції з залежностями, наприклад додавати або оновлювати версію без використання сторонніх інструментів.

SPM не модифікує наш проєкт, що дозволяє додати нову бібліотеку без складнощів використовуючи XCode -> File -> Add Dependency -> вести

посилання на GitHub або завантажити локально папку. Тут виникає складність у тому, що підхід Swift Package використовує простий опис залежностей в Package.swift та не має ніякого репозиторія з файлами. Swift Package Manager притримується підходу використовувати відкритий код залежностей, аби розробники не зіштовхувалися з проблемою, що наданий бінарний файл має несумісний або застарілий код. Звідси існує відповідний недолік, якщо ми не хочемо розкривати код через політичні інтереси, то потрібно буде використати Binary Target та поставляти вже скомпільовану бібліотеку.

За замовчуванням, Swift Package використовує статичне зв`язування, якщо таке можливе, проте це можна змінити вказавши для конкретного продукту тип зі значення *dynamic*. Якщо ж у нас підключено в пакунок хоча б одна динамічна залежність, то в результаті, буде автоматично визначено тип зв`язування. Головною особливістю використання Swift Package, це пакування всіх динамічних залежностей в один динамічний фреймворк, що надає більшу швидкість запуску додатку, оскільки потрібно завантажити всього один бінарний файл під час старту. Така можливість доступна через те, що SPM на етапі компіляції збирає всі можливі ресурси та коди в одну "залежність" та збирає її як динамічну, що знижує кількість об`єктних файлів.

На даний момент, Swift Package Manager досить сирий, оскільки має перелік проблем з стабільною роботою у XCode, проте набирає свою популярність через простоту та інтегрованість для використання у Apple світі.

1.6. Kotlin Multiplatform Mobile

Kotlin Multiplatform Mobile – це новий підхід на використання крос платформної розробки для Android та iOS. Головна особливості полягає у повторному використані спільного коду написаного на Kotlin між іншими таргетами, тобто частину логіки, наприклад роботу з сервером, ми можемо написати на Kotlin та поставити на кожну платформу окрему, а інша частина логіки, наприклад UI, залишиться платформна. Мова програмування Kotlin, як і

більшість мов, має два етапи компіляції – Fronted та Backend, де на першому етапі відбувається парсинг коду у IR представлення. Другий етап залежить від компільованої платформи – для iOS це Kotlin Native, а для Android – Kotlin/JVM.

Kotlin/JVM перетворює IR код у Java байт код, який буде використовуватися для запуску на мобільних пристроях на базі Android. Kotlin/Native трошки складніше, оскільки він слугує Frontend для LLVM. Тобто відбувається перетворення Kotlin – Kotlin IR – Kotlin Native (LLVM Frontend) – LLVM Backend – Native Binary, де в результуючому випадку для нас буде готовий до використання фреймворк.

Все працює з використанням Gradle плагіну, який надає спектр функціональності для побудови потрібного формату для розповсюдження на інші платформи. На даний момент для iOS, KMM підтримує роботу з Framework (це може бути звичайний фреймворк, або universal, або XCFramework) та CocoaPods.

В першому варіанті, нам надається просто Framework який кладеться в папку build та прописується шлях до нього у самому XCode. Для побудови достатньо однієї Gradle задачі, котра збирає готовий до використання фреймворк. Нині, від використання способу з Framework відмовляються, через накладні процеси для доставки у iOS проєкти, проте, даний варіант існує у вигляді першого етапу у використання CocoaPods.

Для CocoaPods потрібно підключити інший Gradle плагін, котрий приносить можливості роботи зі сторонніми залежностями (які потім будуть інтегровані у остаточний проєкт), інтеграцію у існуючі проєкти використовуючи PodSpec, і в особливості метод *script_phase*, який викликається кожен раз під час збірки проєкту та оновлює наш фреймворк, котрий був описаний вище.

Спосіб з використанням Carthage немає, оскільки даний пакетний менеджер втрачає свою популярність та не вирішує проблему з автоматизацією додавання залежності у проєкт, а способу для SPM немає, оскільки в момент написання інтеграцій з іншими рішеннями, даного пакетного менеджеру ще не існувало, тому пропоную розглянути розробку такого інструменту.

2. СПОСОБИ ІНТЕГРАЦІЇ KMM в iOS

2.1. Огляд впровадження KMM на прикладі CocoaPods

Розглянемо існуюче рішення інтеграції KMM в iOS додатки з використанням CocoaPods та проаналізуємо, які важливі позитивні моменти потрібно врахувати під час використання SPM для зручного переходу на інший пакетний менеджер в межах Kotlin Multiplatform.

Першочерговим у використання будь якого інструмент є розуміння базової побудови та фундаментальних механізмів. Екосистема Kotlin Multiplatform побудована на Gradle – система збірки написана на Java, котра оперує задачами [11, с. 80]. Задача, це мінімальна одиниця роботи, яка виконує якийсь функціонал, та може залежить від іншої задачі, тобто існує граф залежностей одної від іншої з відповідним порядком їх роботи. Файли конфігурації Gradle називають скриптами та їх можна писати на Groovy або Kotlin (розширення файлу .kts)

Gradle завдання мають три етапи виконання:

1. Ініціалізація. На цьому етапі де ми отримуємо всю структуру проєкту аналізуючи файл settings.gradle.kts, у котрому підключаються інші модулі - вони повинні мати файл build.gradle.kts
2. Конфігурація, де аналізуються сценарії запуску та відбувається побудова графу залежностей завдання один від одного та запускаються блоки коду, які були вказані в кожному завданню
3. Виконання. Найголовніший процес, який відповідає за виконання завдання – спланує та запускає залежні таски, бере результати з кешу, намагається максимально запускати паралельно аби пришвидшити час отримання результату.

Особливості Gradle завдань є кешування, якщо вхідні параметри (input) не змінилися, то система збірки поверне output, котрий був порахований минулої ітерації.

Робота з Gradle завданнями доступна через використання Gradle плагіни [11, с. 201], які дозволяють інкапсулювати логіку, яка пов'язана з конкретним функціоналом. Для використання у Kotlin Multiplatform існує декілька важливих плагінів, які спрощують роботу, оскільки надають зручні Gradle завдання для побудови фреймворк в одну команду. Один з таких, це *"org.jetbrains.kotlin.multiplatform"* на Лістингу 2.1.1 – він додає функціонал налаштування конфігурації таргетів, додавання залежностей та купу Gradle завдань, які пов'язанні з компіляцією у потрібні бінарні файли, наприклад, для iOS це буде фреймворк, для Android – .aar або .jar.

```
1    plugins { id("kotlin-multiplatform") }
2
3    kotlin {  this: KotlinMultiplatformExtension
4        listOf(androidTarget(), iosX64(), iosArm64(), iosSimulatorArm64())
5        sourceSets { commonMain.dependencies {} }
6    }
```

Лістинг 2.1.1. Приклад налаштування плагіну

Розділення таргетів є важливим в контексті крос платформної розробки, оскільки спільний код пишеться у теці common, а залежний від платформи – у своїх теках. У розробників існує механізм expect/actual, який дозволяє розділяти нативну логіку, описуючи інтерфейс у common коді та відповідні реалізації на кожному таргеті окремо [12]. Наприклад, ми маємо взаємодію по Bluetooth Low Energy, окремих бібліотек для КММ не існує, отож ми повинні створити примітивний інтерфейс вигляду *expect classs BLEService()* в спільному коді, та відповідні реалізації для iOS – *actual class BLEiOSService()* та *actual class BLEAndroidService()*. Компілятор Kotlin має знання про ключові слова expect/actual, тому під час компіляції під конкретний таргет, підмінить реалізацію на потрібну.

Інший Gradle плагін, який використовується для додавання підтримки CocoaPods називається *"org.jetbrains.kotlin.native.cocoapods"*, його головне завдання розширити функціональність попереднього плагіну та інтегрувати

підтримку CocoaPods. Розробники додали нові Gradle завдання, які допомагають з налаштуванням та спрощують інтеграцію з пакетним менеджером.

Відповідно, якщо ми хочемо підготувати нашу залежність для інтеграції у iOS додаток з використанням CocoaPods, потрібно виконати певний перелік задач, які будуть зв’язані між собою:

1. Скомпілювати Kotlin код, котрий використовується у нашій бізнес логікі у IR представлення. Ця задача досить часто використовується, оскільки даний процес виконується для всіх таргетів.

2. Перевести IR код у Frontend LLVM, співставити код з проміжного представлення у загальноприйнятий стандарт LLVM з додавання віртуальної машини, оскільки Kotlin Native використовує автоматичне очищення сміття.

3. Запустити LLVM для генерації вихідного об`єктного файлу, де ми отримуємо побудоване бінарне представлення.

4. Зібрати PodSpec файл, котрий включає у себе налаштування для підключення – назву, версію, тип зв`язування, стороні залежності, і найважливіше, додано *script_phase*, який вказує, яким чином під час компіляції даного Pod, потрібно повторно зібрати бізнес логіку на Kotlin.

5. Виконати pod update, який запустить оновлення залежностей та підготує середовище розробки до запуску, наприклад, допише у Build Phase скрипт з *script_phase.*

6. Опціональним кроком є процес CInterop - на етапі конфігурацій завдань, Gradle проаналізує всі хедери у файлах залежностей, та створить відповідні інтерфейси без реалізації, для того щоб у розробників була можливість доступатися до Swift/Objective-C файлів з Kotlin коду, а вже на етапі компіляції, замість інтерфейсів, будуть використовуватися реальний код.

Наступним кроком після розуміння екосистеми розробки є налаштування плагіну задля коректної роботи зважаючи на потребу розробки. Під час створення залежності нами важливо вказати назву, підтримувані версії таргетів, шлях до залежності та тип підключення (динамічний чи статичний). Оскільки ми працюємо в середовищі Gradle скриптів, важливо мати гнучкість у налаштуванні

залежності, оскільки дописувати власні надбудови поверх досить проблематично. Gradle плагін для CocoaPods має великий спектр опціональних властивостей, наприклад, чи використовувати транзитивність, чи інтегрувати додаткові Gradle модулі у збірки або додавати -fembed-bitcode.

Проєкти без підтримки сторонніх залежностей зустріти досить складно, тому нам важливий функціонал з підключенням бібліотек. Для Kotlin Multiplatform існують два варіації – написані вже на мові Kotlin або нативні бібліотеки.

Залежності написані мовою Kotlin публікуються на Maven репозиторіях та мають можливість підключатися напряму в commonMain, як це показано у Лістингу 2.1. Код буде доступний на всіх таргетах без використання CInterop, проте, якщо ця бібліотека має специфіку під конкретну платформу, її є змога підключити у відповідний таргет використовуючи таргетMain.dependencies{}.

Інший варіант, це підключення iOS бібліотеки, яка була написана без урахування особливостей KMM. Для CocoaPods у Gradle плагіні ми маємо налаштування *pod()*, де можемо вказати назву залежності, її версії або додаткові параметри, як показано на Лістингу 2.1.2. В результаті, ці залежності пройдуть процес CInterop та будуть також доступні для виклику в Kotlin код для iOS таргету.

```
19    cocoapods { this: CocoapodsExtension
20        pod( name: "RealmSwift")
21        pod( name: "lottie-ios",  version: "~>4.2.0")
22        pod( name: "FBAEMKit") { this: CocoapodsExtension.CocoapodsDependency
23            version = "~>16.2.1"
24            extraOpts += listOf("-compiler-option", "-fmodules")
25        }
```

Лістинг 2.1.2. Приклад підключення бібліотек в CocoaPods плагіні

Якщо ж ми хочемо додати залежність як фреймворк без використання пакетного менеджеру, то для цього потрібно самостійно описати специфічний файл *.def з вказанням всіх хедерів бібліотеки. Для того, щоб під час збірки власного фреймворку, Gradle завдання запустив процес генерації інтерфейсів для використання у common коді - процес CInterop аналізує всі заголовні файли

залежності, перетворюю їх у IR представлення та переводить у відповідні Kotlin типи. Для налаштування потрібно вказати, для яких саме таргетів ми проводим процес CInterop та вказати шлях до *.def, щоб Gradle завдання на етапі конфігурації запустив процес генерації. Останнім кроком для компонування буде вказання параметром де брати фреймворк та його назву. Переглянути детальний код можна у Додатку Б. Як можна оцінити, у випадку користування CocoaPods, процес додавання нової iOS залежності суттєво простіший, ніж прописувати вручну.

В першому розділі ми обговорювали підтримку ресурсів, наприклад картинок, у фреймворках. На даний момент, немає готового рішення у KMM для використання у Kotlin бібліотеках ресурсів, проте існують стороні рішення, котрі пропонують тягнути за собою ще один фреймворк створений тільки з ресурсів. З досвіду використання таких рішень, вони мають більше проблем, оскільки вирішують біль одноразово, а не фундаментально. З гарних новин це те, що команда Kotlin знають про дану проблему та працюють над вирішенням.

Важливим етапом при підключення KMM в iOS проєкти є зручність інтеграції для команд, які не мають кваліфікації в Kotlin, проте мають взаємодіяти з фреймворком використовуючи XCode. Найголовнішим файлом конфігурації залежності є згенерований PodSpec файл (Додаток А), котрий оновлюється на кожну збірку проєкту при використанні CocoaPods.

З точки зору першочергового налаштування проєкту, достатньо виконати Gradle Build, який згенерує специфічний .podspec з заданими параметрами в плагіні. Після цього, виконання pod update задіє оновлення .xcodeproject, котрий включає у себе файли конфігурації запуску в XCode та оновить залежності в теці Pods, якщо вони змінилися.

Залишається відкрите питання, як саме з XCode ми можемо запустити додаток, та наші залежності оновилися автоматично без використання сторонніх команд у терміналі тощо. Розробники CocoaPods додали параметр до опису специфікації, котрий називається script_phases, виконання якого додається у Build Phase самої збірки iOS додатку. Даний скрипт складається з запуску Gradle

завдання, котре запустить нову генерацію фреймворку з використанням кешу та змін оточення, наприклад що це дебаг збірка. Весь процес виглядає як ланцюжок з запуску проєкту в XCode -> повторне створення фреймворку -> збірка додатку -> динамічне зв`язування. Таким чином, ми інкапсулюємо логіку роботи з Gradle для розробників, та надаємо готовий інструмент для роботи у iOS середовищі використовуючи знайомий CocoaPods.

У випадку користування CocoaPods, ми маємо можливість вибирати тип компонування для побудови Framework – статичне чи динамічне. За замовчуванням, KMM пропонує варіант з статичним зв`язуванням, оскільки результуючий фреймворк зазвичай не займає багато місця, що дозволяє приєднати його, до основного додатку без суттєвого збільшення місця. Звичайно, розробникам надається можливість змінити тип на динамічний, якщо ми маємо великий обсяг коду, аби зменшити розмір застосунку. Проте, окрім KMM фреймворку, ми можемо використовувати інші залежності, котрі поставляються як нативні, тобто написані не мовою Kotlin, що накладає певні складнощі у виборі типу компонування. Розглянемо дві ситуації, коли ми використовуємо бібліотеки в різних обсягах.

Перший варіант, ми маємо власну KMM бібліотеку та декілька додаткових iOS залежностей у кількості менше п`яти. При використанні статичного зв`язування буде більше розмір додатку ніж у динамічного, проте, швидкість запуску додатку при використанні динамічного суттєво не зросте, в порівнянні зі статичним, оскільки кількість підключених під час виконання бібліотек, мала.

Другий варіант, це власна KMM бібліотеку та кількість доданих інших бібліотек більше ніж 5, в такому випадку, Apple рекомендує обдумати специфіку використання динамічного зв`язування, оскільки починає страждати продуктивність динамічного компонувальник, тобто швидкість запуску додатку може суттєво зрости, що погіршить користувацький досвід. Використання статичного зв`язування з великою кількістю бібліотек призводить до великого розміру додатку.

В обох варіантах є свої недоліки та переваги, як то великий розмір додатку через використання статичного зв`язування або збільшений час запуску у динамічному зв`язування. Оцінивши дані проблеми, можливо запропонувати рішення, яке надає нам SPM, це побудова додатку без статичного підключення, та поєднання всіх залежностей в один фреймворк з наступним динамічним компонуванням. Таким чином, розмір додатку не зростає, а динамічне компонування не впливає на швидкість запуску, оскільки компонувальнику треба поєднати всього один фреймворк під час запуску. Отже, для оптимізації роботи з залежностями у KMM є сенс використовувати механізм роботи з залежностями Swift Package та реалізувати зручну інтеграцію у проєктах.

2.2. Дослідження теоретичної реалізації інтеграції KMM в SPM

Проаналізуємо можливі теоретичні варіанти реалізації інтеграції SPM у проєктах KMM. Першим кроком потрібно визначитися з використанням інструментів для розробки рішення, оскільки від цього залежить зручність подальшого впровадження іншими розробниками собі у проєкт.

Найочевидніший це використання консольних застосунків, таких як *kotlinc* та *kotlinc-native* для компіляції Kotlin коду, та подальшого використання результуючих файлів. Прописати детальні кроки та інструкції виконання можемо використовуючи shell скриптинг створивши простий *.sh файл з усім процесом.

Простий Swift Package (Додаток В) складається з опису підтримуваних операційних систем, опису, які залежності входять в пакунок, стороніх залежностей та відповідно підключених таргетів. Таргет у контексті пакунків являє собою опис сторонніх бібліотек з GitHub, локально розміщених або завантажених у вигляді бінарного файлу. Відповідно, під час додавання нової залежності в KMM фреймворк нам потрібно вказати всі можливі залежності в таргеті.

Розглянемо на прикладі Додатку В. Ми маємо бібліотеки OAuthFramework, яка інкапсулює логіку авторизації на різноманітних сайтах. В залежностях неї є GoogleSignIn, FacebookCore/Login, Logging. При додавання OAuthFramework як сторонньої залежності до нашого проєкту КММ потрібно вказати в самому пакунку залежні йому бібліотеки та вихідний пакет як на Лістингу 2.2.1. Специфіку використання пакунків потребує явного прописування транзитивності, що спрощує роботу пакетному менеджеру в пошуках внутрішніх залежностей та покладає цю роботу на розробників. У випадку написання автоматизованого інструменту для інтеграції SPM, для нас постає проблема для вирішення цієї задачі також, оскільки це одна з важливих функцій.

```
24        .target(
25            name: "TestSPM",
26            dependencies: [
27                .product(name: "FacebookCore", package: "facebook-ios-sdk"),
28                .product(name: "FacebookLogin", package: "facebook-ios-sdk"),
29                .product(name: "GoogleSignIn", package: "GoogleSignIn"),
30                .product(name: "Logging", package: "swift-log"),
31                .product(name: "OAuthFramework", package: "OAuthFramework"),
```

Лістинг 2.2.1. Приклад пакунку з сторонніми бібліотеками

Зважаючи на описані проблематики, можемо скласти з яких етапів буде складатися наша інструкція від наявного Kotlin коду до генерації SPM:

1. Передача параметрів налаштування SPM, як параметри запуску скрипту. Це може бути назва фреймворку, підтримувані операційні системи, стороні бібліотеки тощо. На стороні скрипту потрібно буде запровадити обробку цих значень.

2. Скомпілювати Kotlin код у IR. Даний процес можна виконати використовуючи консольний інструмент *kotlinc*, таким чином отримуючи опосередковане представлення для подальшої роботи.

3. Наступним кроком перевести Kotlin IR у вигляд LLVM синтаксису, аби звести до зрозумілого представлення з використанням інструменту *kotlinc-native*.

4. Запуск LLVM машини з передачею параметрів, що нам потрібно отримати бінарний файл для iOS у вигляді фреймворку. Важливим нюансом є

використання саме LLVM від Apple, оскільки результуюча бібліотека буде відрізнятися біткодом, що суперечить політиці компанії, тому під час перевірки додатків у App Store буде отримана відмова.

5. Завантаження бібліотек, як було зазначено раніше, якщо ми використовуємо стороні залежності, нам важливо вказати транзитивно всі назви в маніфесті. Отримати весь контент ми маємо змогу через API GitHub, оскільки SPM використовує ідею відкритого висхідного коду, тому скориставшись *curl*, ми маємо змогу завантажити весь код бібліотек та проаналізувати кожний маніфест окремо.

6. Генерація SPM пакету зважаючи на особливості Swift синтаксису. Потрібно вручну зібрати маніфест файл з вхідних параметрів скрипту, вказання бібліотек, створити правильну структуру папок.

7. Перевірка коректності SPM чи оформлений файл маніфесту згідно стандарту Apple. Існує команда *swift package describe*, яка повертає код 0, якщо не виникало ніяких помилок.

8. Інтеграція в XCode. Оскільки SPM немає аналогу *script_phases* від CocoaPods, то підключення даного рішення буде виглядати як додавання локально пакунку в IDE та прописування в Build Phase запуск нашого скрипту для оновлення на кожний запуск додатку.

Інтеграція CocoaPods з KMM для iOS проєктів використовує Gradle плагін, оскільки існує велика кількість готових завдань для спрощення роботи з Kotlin, в особливості KMM. Вище описані кроки для описання скрипту мають свої недоліки – відсутність кешування, складність тестування, великий шанс помилок, оскільки відсутні будь які перевірки на типізацію або валідації результату, та загалом на власне рішення досить складно знайти відповідь на форумі або в документації.

Використання Gradle плагіну найбільш підходящий спосіб інтеграції KMM та SPM, оскільки плагін спрощує та надає повний спектр доступних задокументованих та протестованих завдань для подальшого використання. Кожний етап з скрипту можна замінити на деякі готові рішення, а деякі написати

самостійно. Наприклад, процес перетворення Kotlin коду в iOS фреймворк потребує налаштування Gradle скрипту у вигляді опису підтримуваних таргетів та запуску команду *assembleXCFramework* замість 2-4 пунктів у shell скрипту.

Опис всіх підтримуваних таргетів, з кроку 1, вказується у базовому Gradle плагіні для КММ, отож нам достатньо описати новий Kotlin DSL для вказання додаткових налаштувань по інтеграції SPM, як наприклад в CocoaPods Додаток Г. Даний опис мовою програмування повинен підтримувати такі поля, як назва фреймворку, версія Swift, версію iOS/macOS, блок з залежностями – звідки завантажувати, версія/гілка коду.

Викачування бібліотек з GitHub з етапу 5 у екосистемі Gradle реалізовуємо через написання окремого Kotlin HTTP клієнту, який буде вивантажувати код у окрему теку з можливим використанням кешуванням, щоб повторно не виконувати даний крок, якщо нічого не змінилося у вхідних параметрах.

SPM має вихідний код, що дозволяє детальніше дізнатися про функціонал та приклади використання, як наприклад, аналіз іншого пакунку. На вхід подається шлях до Package.swift і ми маємо можливість отримати повну інформацію про граф транзитивних залежностей, списку підтримуваних операційних систем тощо. Це дозволяє вирішити проблему кроку 6, а саме, додавання всіх залежностей у результуючий новий маніфест з КММ, достатньо реалізувати запуск Swift коду з Gradle завдання.

Виконання валідації пакунку з етапу 7 можемо провести використовуючи можливість *exec* з екосистеми Gradle, яка дозволяє виконувати команди з терміналу. Після успішного створення Swift Package, залишається інтеграція у XCode. Нажаль, через закритість IDE, у нас немає можливості зручної взаємодії як у CocoaPods, оскільки ми не модифікуємо проєкт. Залишається описати shell скрипт, котрий буде викликати Gradle завдання, котра в свою чергу по черзі виконає потрібні етапи зважаючи на вказаний та додати його в Build Phase.

2.3. Варіант інтеграції SPM у KMM з використанням XCode

Можливим альтернативним рішенням було використання XCode – рідне середовище SPM – з функціоналом його плагінів, які дозволяють створювати надбудови над IDE задля додавання нового функціоналу без втручання Apple.

Першочергово, потрібно додати підтримку мови програмування Kotlin в IDE для зручного написання коду – автоматичне доповнення коду, підсвітка синтаксису. Даний функціонал можемо реалізувати в використовуючи server language, це надає можливість писати код в будь якому текстовому редакторі та отримувати постійні пропозиції для написання коду.

Наступним кроком інтеграції SPM буде генерації результуючого фреймворку з Kotlin коду з подальшим приєднанням до XCode. Для цього треба виконати певні етапи компіляції використовуючи, знову таки, консольні інструменти для формування остаточного фреймворку з вказанням всіх потрібних параметрів та кроків. За аналогією з використанням shell скриптинг можна описати інструкцію можна покласти в Build Phase та запускати на кожну збірку додатку.

Вибір додаткових залежностей можна оформити через використання готового інструменту XCode, який пропонує пошук бібліотек за назвою або посиланням на GitHub або вказати локальний шлях до іншого пакунку. Таким чином, спрощується парсинг сторонніх бібліотек, оскільки ми використовуємо готову обгортку від Apple. Додані залежності ми додаємо в результуючий проєкт та маємо доступ до них, що означає відсутність потреби організовувати роботу на стороні Kotlin коду, проте це впливає на відсутність можливості оформити процес CInterop, оскільки ми не маємо доступ до підключених залежностей, що означає на неможливість отримання можливих інтерфейсів в Kotlin коді задля використання у написані бізнес логіки.

Недоліками такого рішення є маленька кількість можливостей, який надає XCode для плагінів та стабільність IDE, котра має особливості кожної нової версії додаткові проблеми. У нас немає можливості використовувати Debug з

відповідними зупинками на брейкпоінтах для сторонніх мов програмування або отримати всі підключені Swift Package, щоб провести CInterop та отримати доступ до сторонніх методів бібліотек. В такому варіанті, у нас зникають найпроблемніші етапи з додаванням залежностей у проєкт, що дозволяє не будувати різноманітні інструменти для взаємодії з Swift код, як наприклад, парсинг графу залежностей, оскільки даний функціонал інкапсулює XCode.

Порівнюючи з Gradle плагінами, ми втрачаємо протестовані методи роботи з Kotlin кодом, як компіляція в одну команду, зручний DSL для налаштування фреймворку. Найголовніше, з використанням XCode плагінів ми так само не маємо можливості модифікувати проєкт, аби додати ще один етап збірки в додатку.

У результаті, варіант з інтеграцією SPM для KMM у купі з Apple IDE не позбавлена сенсу, оскільки надається зручний інструментарій для роботи з Swift та Swift Package. Проте, відсутність повноцінного Kotlin редактора та велика наявність багів у XCode ставлять під сумніви доречність даного рішення навіть з урахуванням написання власного плагіну, який має обмежений API.

3. РОЗРОБКА ІНСТРУМЕНТУ ДЛЯ ІНТЕГРАЦІЇ SPM В KMM

3.1. Створення Gradle плагіну

Першочергово, потрібно створити Kotlin DSL для налаштування нашого Swift Package, який буде генеруватися з опису в Gradle плагіні. Всі можливі характеристики для роботи описані в Додатку Ґ, цими полями будуть – мінімальна версія iOS, опціонально версія macOS, якщо ми хочемо побудувати фреймворк під цей таргет, назва фреймворку, блок з залежностями та версія Swift, оскільки від цього залежить генерація маніфесту.

Зупинимося детальніше на сторонніх бібліотеках, та яким чином ми можемо їх підключати у SPM. Як було підмічено у першому розділі про пакунки, під час використання пакетного менеджеру від Apple, всі залежності зберігаються на GitHub, відповідно для отримання доступу нам достатньо вказати посилання у вигляді https://github.com/username/repository. Розширенням такого способу є вказання додатково гілки або версії-тег, або використання локального шляху до вказання пакету на персональному комп'ютері. Просто вказання посилання без додаткових значень буде брати головну вказану гілку на GitHub. Для підтримки різноманітних варіантів додавання залежностей було розроблено DLS як на Додатку Д, який включає в себе такий API як *url(link)*, *branch(branch , link)*, *version(tag, link)* та *path(file)* в контексті блоку dependencies. Для цього, у функції ми використовуємо патерн Builder для того, щоб можна було прописати різноманітні варіації підключення бібліотеки та зібрати в список використовуючи метод *build*().

Важливим параметром у KotlinMultiplatformSwiftPackageExtension є Project, котрий інкапсулюється на етапі створення плагіну у блоці виконання *apply()*, який виконується відразу під час першого етапу Gradle – ініціалізації проєкту, таким чином, на етапі налаштування пакунку, у нас є можливість доступитися до різноманітних властивостей проєкту та провести перевірку, якщо виникне така потреба.

Наступним кроком інтеграції є створення самого плагіну та побудова відповідних етапів зі створення готово рішення. Для цього потрібно прописати дві головні точки входу, як продемонстровано у Додатку Е – зареєструвати наш плагін та написати реалізацію унаслідувавши клас Plugin та метод *apply()*.

У блоці gradlePlugin у скрипті модуля описуємо, яким саме чином ми реєструємо наш плагін прописуючи назву для Accessor, який згенерує інтерфейс для DSL, ідентифікатор для підключення у блоці *kotlin { id() }*. У результаті ми отримаємо згенерований інтерфейс для конфігурації SPM у блоці *swiftPackage{}* де у лямбда виразі маємо доступ до нашого класу налаштувань та відповідно маємо змогу налаштувати за потреби, як на Додатку Є.

Другим важливим моментом створення власного плагіну після його конфігурації, це реалізація методу *apply()*, приклад у Додатку Ж, де нам потрібно описати всю логіку взаємодії, можна сприймати як метод main() в мовах програмуваннях, звідки починається виконання. Спочатку створимо об’єкт нашого класу налаштувань пакунку через використання *project.extenstions.create*, таким чином, ми отримаємо доступ до полів властивостей KotlinMultiplatformSwiftPackageExtension задля подальшої обробки. Аналогічно, але використовуючи *project.extensions.find* знайдемо об’єкт налаштувань базового KotlinMultiplatformExtension, який надає інформацію про підключені таргети. Важливо, що дану логіку потрібно виконувати після конфігурація Gradle модулів з використанням функції *project.afterEvalute*, що ми не отримали NullPointerException, тому що дані не були проініціалізовані.

Оскільки ми плануємо використовувати частину готової логіки з використанням KMM Gradle плагіну, нам потрібно зареєструвати завдання, яке буде відповідати для генерації готового XCFramework. Для цього достатньо ітеруватися по всім KotlinNative таргетам та додати кожен в об`єкт *XCFramework*() вказавши назву фреймворку, яку ми можемо отримати з KotlinMultiplatformSwiftPackageExtension. Таким чином, ми використовуємо готовий протестований API, у Додатку З, від компіляції Kotlin коду до отримання бінарного представлення готово до підключення в iOS проєкти.

Gradle скрипти виконуються послідовно, тому нам потрібно зареєструвати Gradle завдання, щоб побудова або оновлення пакунку відбувалося тільки за викликом. Для цього, використаємо метод *project.tasks.register* з назвою та опишемо у лямбда виразі, які кроки будуть виконуватися під час запуску та додаємо залежність від завдання, яке відповідає створенню XCFramework з назвою *assemble(Name)(Debug/Release)Framework*.

Важливо, під час iOS розробки ми маємо два різних типу збірки – release та debug, які відповідають різним властивостям, наприклад, наявність дебаг таблиці або обсуфікація коду. XCode під час збірки додає в зміні оточення параметр CONFIGURATION, який вказує на обраний тип збірки, таким чином, у нас є можливість передати дане значення у параметри Gradle завдання використовуючи *–buildType=$CONFIGURATION* та обравши, який фреймворк нам потрібен.

Додатковою можливість використання фреймворків це підтримка їх зі сторони MacOS таргетів, тобто створивши XCFramework з фреймворком для MacOS, у нас з'являється можливість підключати його не тільки в iOS проєкти. Якщо ми вказуємо *macos("")*, в нашому Gradle скрипті, то повідомляємо про намір збирати відповідну конфігурація для підключення в Apple проєкт. З точки зору створення фреймворку, вся робота для Apple таргетів інкапсулюється базовим Gradle плагіном KMM, для цього достатньо їх описати як на Лістингу 3.1.1. З боку інструкцій, які виконуються під час побудови пакунку, важливо перевірити умову, якщо ми вказали мінімальну версію підтримки MacOS, то і MacOS таргети описані в KMM повинні бути, тому що немає сенсу формувати пакунок з вказанням підтримки операційної системи Apple для персональних комп'ютерів та відсутності цієї підтримки у фреймворку.

```
7   kotlin { this: KotlinMultiplatformExtension
8       macosArm64()
9       macosX64()
10  }
```

Лістинг 3.1.1. Приклад налаштування MacOS

Перед роботою з залежностями треба підготувати оточення для роботи з SPM, а саме згенерувати інший Swift Package - парсер, який буде виконувати роль обробки бібліотеки по локальному шляху на вхід, та JSON файл на виході зі списком всіх транзитивних залежностей для формування власного результуючого пакунку. Парсер використовує залежності з вихідного коду SPM, отож у нас виникає потреба у завантаженню коду з GitHub та створення пакунку з додаванням локального шляху до маніфесту. Для цього опишемо інструкцію як у Додатку І, яка використовуючи метод *exec* для запуску командної стрічки з виконанням *git clone* за адресою зберігання коду. У результаті, у папці build/swift-package-manager модуля ми отримаємо вивантажений репозиторій з маніфестом та під час наступних запусків Gradle завдання, ми будемо перевіряти, чи існує тека з SPM та пропустимо крок, якщо вона вже є.

Наступним кроком буде генерація парсеру, який буде включати у себе пошук графу залежностей та транзитивних бібліотек. Нам достатньо реалізувати файли маніфесту та файл для запуску у вигляді звичайних файлів використовуючи *File.writeText()*. Package.swift (Додаток Ї) складається з executable таргета, оскільки ми хочемо виконувати в CLI наш парсер використовуючи в Gradle завданні метод *exec* та двох залежностей - локальним з вказанням build/swift-package-manager і логувальником для розуміння етапу роботи.

Файл для запуску з назвою *parser.swift* буде складатися з одного статичного методу main() та анотацією @main, котра вказує на вхідну точку нашого парсеру. Для початку отримуємо аргументи запуску, аби отримати шлях до маніфесту та результуючого JSON. Після цього, використовуючи API SPM дістаємо всю можливу інформацію з пакунку (Додаток Й), таким чином, ми маємо повне представлення про всі транзитивні бібліотек, тому нам достатньо ітеруватися по кожній з них та дістати додаткові дані про версію та назву продукту. Повний файл парсеру можна побачити у Додатку К, зі створенням структури *Parser* та відповідним записуванням у файл, котрий ми вказали під час запуску.

Оскільки нам потрібно отримати результат виконання парсеру в Gradle середовищі, то потрібний спосіб передати дані через різні процеси, тому, був обраний варіант зберігання в окремий JSON файл, аби потім його зчитувати у Gradle завданні після успішного виконання. Для цього, на стороні Swift були створенні *Codable* структури, які після запису у файл з вказанням *JSONEncoder()* перетворювалися на потрібний нам формат. Дана структура зберігає назву пакунку, версію коміту, список підтримуваних таргетів та продуктів та версії залежностей для кожного таргету, що дозволяє у результуючому графі залежності впевниться, що у нас немає суперечок у транзитивних бібліотеках, наприклад, фреймворк А використовує бібліотеку Log версії 1.1.0, а фреймворк Б використовує таку саму бібліотеку іншої версії. У такому випадку під час компіляції, компонувальник може зіштовхнутися з проблемою, яку саме версію треба покласти в результуючий додаток, тому, ми можемо аналізувати граф та вказати на можливі конфлікти до запуску проєкту п опередивши розробника про можливі наслідки.

Результуючий парсер має структуру як на Лістингу 3.1.2. та надає API використання якого виглядає як *swift run swift-package-tools –from={} –to={},* де параметром from виступає шлях до маніфесту, а параметр to – шлях до результуючого JSON файлу. Дані файли створюються, якщо відсутні, під час виконання Gradle завдання та підставляться локальні значення, як наприклад, шлях до build/swift-package-manager, котрий використовуються як залежність. Таким чином, для кожної підключеної залежності, викликавши Gradle завдання з методом *exec,* ми отримуємо відповідну повну інформація про транзитивні бібліотеки, що дозволяє генерувати результуючий пакунок.

Лістинг 3.1.2 Структура парсера

Для роботи з залежностями нам потрібно завантажувати їх локально та отримати інформація про транзитивні залежності. SPM використовує GitHub, отож нам потрібно використовувати API для отримання та завантаження вихідного коду. Запити на сервер будемо викликати за допомогою Kotlin бібліотеки Ktor, котра надає простий інтерфейс роботи з мережею. GitHub надає зручний спосіб отримання мета інформації репозиторії у вигляді api.github.com/repos/username/repo, але оскільки ми маємо можливість підключити залежності з вказанням додатково версії або гілки, то нам потрібно використовувати і інші запити. У випадку використання локального шляху до залежності, цей етапі можна пропускати, тому що ми маємо висхідний код для обробки парсером. Розглянемо варіації роботи з GitHub у випадку різних підключень (Додаток Л):

- Підключення бібліотеки за посиланням з вказанням гілки. Для такого варіанту застосовуємо api.github.com/repos/username/repo/branches/branch з вказанням імені автора, назвою репозиторію та назвою гілки. У результаті запиту отримуємо хеш комміту.
- Підключення за посиланням без вказанням додаткових параметрів. Використовуємо api.github.com/repos/username/repo з вказанням імені автора та назвою репозиторію. Нам повертається JSON з мета інформацією, а саме гілка за замовчуванням, звідки потрібно брати вихідний код та робимо запит аналогічно попередньому пункту отримуючи хеш комміту.
- Підключення бібліотеки за посиланням з вказанням версії. Для такого варіанту застосовуємо api.github.com/repos/username/repo/branches/tags, де на виході маємо інформацію про всі версії коду, та відфільтруємо в пошуках потрібного та візьмемо його хеш комміту.

Всі випадку зводяться до отримання хешу, тому даний параметр можна використовувати як ключ для кешування. Якщо ми намагаємося отримати повторно бібліотеку, але маємо у наявності папку з відповідною назвою, то можемо не завантажувати, якщо дана тека існуюча. Для завантаження файлу архіву за коммітом використовуємо

api.github.com/repos/username/repo/zipball/commit та копіюємо вихідні байти у тимчасовий архів. У Додатку М описаний код для роз-архівації, результат якого ми кладемо у папку build/deps/username/commit для подальшої роботи та додаємо у сховище HashMap з ключем по структурі залежності та значенням – шляху до залежності – для подальшої обробки.

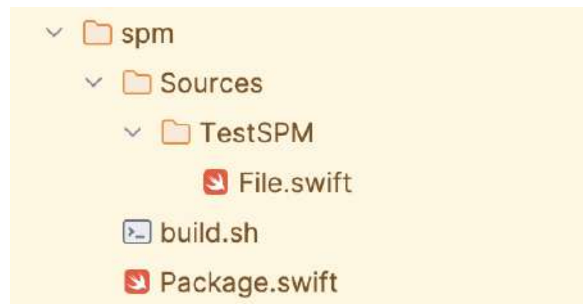
Після отримання всіх залежностей локально, наступним етапом є отримання мета інформації про кожні підключену бібліотеку за допомогою нашого парсеру (swift-package-tools). Для цього пройдемося ітеративно по всім залежностями зі сховища та запустимо у командному рядку як у Додатку Н. Результат виконання, а саме JSON покладемо поруч у папці з бібліотекою та десеріалізуєм файл використовуючи *kotlinx.serialization* для отримання структури парсера у Kotlin коді та покладемо у сховище з новою HashMap, де ключем буде об’єкт залежності а значенням – результат запуску swift-package-tools. Таким чином, на даному етапі, ми маємо все що потрібно для формування пакунку – транзитивні залежності, мінімальна версія iOS/macOS, версія Swift, сформований XCFramework.

Для побудови повноцінного пакунку, ми повинні створити правильну структуру тек та файлів. У Apple вимогах існує пункт, що тека з кодом не повинна бути пустою, тому просто покладемо порожній swift файл. Найскладнішими у генеруванні є маніфесту, оскільки він потребує всю отриману інформацію впродовж виконування Gradle завдання. Генерування маніфесту можна розбити на декілька етапів у Додатку О:

1. Формування опису продукту та версій ОС. Додаємо інформацію про версію Swift для компілятора, вказуємо імпорт PackageDescription та починаємо створювати об’єкт Package з назвою, підтримуваними платформами вказаних у Gradle скрипті. Та створюємо опис продукту, а саме, що ми маємо на увазі створити бібліотеку з такими таргетами як наш фреймворк та таргетом з усіма бібліотеками. Вказуємо їх у форматі вказаної назви пакунку для бібліотек та назва пакунку + Framework для бінарного файлу.

2. Наступним етапом формування пакунку це додавання інформації звідки брати залежності, тобто вказання всіх залежностей як посилання на GitHub та вказанням або хеш коміту, або тег, або гілку

3. Остаточній етап це опис таргетів. Один з них, це binaryTarget з локальним відносним шляхом до побудованого XCFramework та назви, яка складається з назви пакунку та префіксу Framework.



Лістинг 3.1.3 Структура результуючого пакунку

Для формування пакунку використовуємо Java API для роботи з файлами та запис у ці файли тексту, котрий ми сформували у три етапи для побудови пакунку.

Оскільки SPM не модифікує проєкт, нам потрібно створити shell скрипт, який буде виконуватися на кожній запуск додатку, задля повторного запуску нашого Gradle завдання та оновлення пакунку. Функція описана у Додатку П та представляє з себе *export* інформації про назву таски, назви Gradle модуля, перехід в директорію проєкту та запуск Gradle завдання з відповідними параметрами як шлях до модуля, назва Gradle таски та тип запуску iOS додатку (Release/Debug). Важливим є коментар на початку, котрий включає в себе команду для запуску цього скрипту з відповідними правами, тобто ми генеруємо самі себе та надаємо розробнику, що саме він повинен додати в Prebuild Phase до XCode.

Таким чином, достатньо однин раз запустити команду вручну, після чого Gradle плагін пройде по всім кроками описаних для виконання createSPM та побудує пакунок як на Лістингу 3.1.2. Розробнику достатньо в XCode обрати Add Package Dependencies -> Add Local Package та обрати теку build/spm. Після чого XCode почне індексацію пакунку, а саме завантаження файлів залежностей та

підключення у основний проєкт. Ліворуч в IDE можна буде побачити список всіх залежностей та наш локальний пакунок, розгорнувши його потрібно перейти у файл build.sh скопіювати коментар та додати його в Build Phase.

Останнім кроком інструкцій (Додаток Р) по побудові пакунку є перевірка коректності маніфесту, оскільки нам важливо впевнитися, що XCode зможе прочитати даний файл та почати роботу з Swift Package. Для того, щоб отримати інформацію про валідність достатньо виконати команду *swift package describe* використовуючи метод *exec* та взяти значення коду результату виконання. Якщо повертається 0, шо означає успіх виконання та відповідно пакунок відповідає вимогам Apple та готовий до інтеграції у проєкту.

3.2. Недоліки та можливі покращення

Розроблене рішення має деякі несуттєві недоліки та майбутні покращення. Першочергово це підтримка різноманітних версій Swift, оскільки від цього залежить вигляд пакунку. У розробці використовується старий формат для 5.6, де потрібно вказувати версію комміту, якщо використовується залежність без вказання версії або гілки. У більш нових версіях можна його не вказувати, і компілятор за нас знайде останній комміт у гілці за замовчуванням. Також, існує потреба у підтримці декількох версій Swift у межах одного пакету, тому є сенс створювати різноманітні Package.swift з префіксом @ та вказанням версії. Дани функціонал можливо реалізувати розширивши API Kotlin DSL додавши список Swift версій.

Ще однією можливістю для покрашення це оформлення попереджень після отримання всіх транзитивних залежностей про можливу несумісність бібліотек. Наш парсер надає повний перелік версій, що дозволяє пройтися по кожній та проаналізувати, чи використовується залежності з однаковою назвою та видати помилку/інформація у консолі, якщо розробник вказав прапорець для цього функціоналу. Реалізація достатньо примітива у вигляді проходження по

всім залежностями, проте, може достатньо багато займати часу на кожний запуск, оскільки кількість бібліотек може бути суттєвою.

Продовжуючи тему швидкості збірки, варто повернутися до однієї з фундаментальних можливостей Gradle, наприклад, чому цього не вміють shell скрипти, це кешування завдань. Побудова фреймворку ми інкапсулювали віддавши дане завдання базовому KMM Gradle плагіну, що має реалізацію від компанії JetBrains і нам не важливо, що саме знаходиться там. Залишаються такі етапи як – підготовка середовища, завантаження залежностей, парсинг залежностей, генерація пакунку та перевірка валідності.

Підготовка середовища складається з завантаження вихідного пакунку та генерації парсера – ці дії ми виконуємо одноразово та в наступні запуски просто перевіряємо, чи папки існують, якщо так, пропускаємо даний етап, в протилежному випадку виконуємо *git clone* для менеджера та *File.writeText()* для парсеру.

Наступними кроком є завантаження залежностей і тут ситуація трошки складніше, оскільки для отримання кешу підключеної бібліотеки, нам потрібно отримати його значення, а для цього потрібно виконати 2-3 запити на GitHub API, що займає суттєвий час. Вирішенням цієї проблеми може стати створення текстового або json документу, у котрий ми будемо записувати пари відповідності, які бібліотеки ми вже завантажили. Операція читання з файлу достатньо проста та вимагається всього одноразово під час запуску завдання у відмінності від запитів до серверу.

Альтернативним варіантом покращення роботи з залежностями є завантаження тільки маніфесту з GitHub. У такому варіанті ми скорочуємо час викачування всього коду та спрощуємо структуру проєкту, оскільки для отримання інформації з парсеру достатньо вказати тільки маніфест. Проте у такому випадку, ми не маємо доступу до всього вихідного коду та не зможемо скористатися функціоналом CInterop для отримання Kotlin інтерфейсів у KMM коді. Більш правильним варіантом оптимізації сторонніх бібліотек є викачування відразу залежності в теку *~/Library/Caches/org.swift.swiftpm/repositories*, куди

XCode складає всі пакунки та кешує. Даний варіант зменшує подальший час індексації у IDE, оскільки не завантажує залежності повторно, оскільки в існуючому варіанті ми маємо бібліотеки в build/deps для KMM проєкту, та аналогічні бібліотеки дублює XCode.

Замість прописування Build Phase для додавання повторної генерації фреймворку, можна використати нову можливість SPM, а саме Swift Plugin, які виконуються перед запуском проєкту та написані мовою Swift. Вони надають зручний інтерфейс для запуску коду до початку збірки, що надає можливість запустити, наприклад, лінтери, генерацію коду тощо. Таким чином, нам достатньо описати виконання Gradle завдання та вказати одноразово в Build Settings про свою наміру виконувати даний Plugin. Єдиним недоліком може стати відсутність підтримки вхідних аргументів, оскільки це може бути критичним у випадку потреби зовнішніх параметрів.

Найцікавішим процесом, котрий має сенс, проте не був реалізований, це створення CInterop до кожного з пакунків для подальшого використання у Kotlin коді. Для цього потрібно на кожну залежність створювати *.def файл з додаванням всіх заголовних файлів, додавання назви пакунку та створенням відповідних Gradle завдань на кожну з бібліотек. Процес пошуку всіх header файлів можна реалізувати через використання рекурсивного пошуку по файлам, а додавання ще одного етапу в інструкцію виглядало як ітерація по всіх залежностей та створення відповідних *cinterops.creating{}* для кожного з елементів. Нам не важливо додавати параметри в linkerOpts() для вказання фреймворку, оскільки iOS компілятор сам підключить пакунки.

3.3. Результати

У результаті розробки ми отримали Gradle плагін, який додає можливість інтегрувати SPM у KMM проєкти. Для поширення даного рішення було використано Maven сховище, яке зберігає jar файли різноманітних Java бібліотек. Існують різноманітні варіації сервісів, які реалізують доставку залежностей,

проте мною було обрано jetpack.io, де автоматично при публікації нової версії на GitHub запускається завдання *gradlew publishMavenLocal* та відбувається збірка бібліотеки, котра буде доступна за ідентифікатором com.username:repository:version [13].

Отож, для підключення Gradle плагіну достатньо додати jitpack.io в скрипті налаштування проєкту у блоці *pluginManagment.repositories{}* посилання на https://jitpack.com та у конкретному модулі в блоці plugin{} додати id("kotlin-multiplatform-spm"). Після етапу ініціалізації ми будемо мати доступ до Kotlin DSL *swiftPackage{}* та можливість налаштовувати під наші потреби.

Проведемо порівняння результуючих додатків між використанням KMM у зв'язці з CocoaPods та SPM. Для цього використаємо функціонал Archive, котрий компілює та збирає проєкт у спеціальний формат, котрий призначений для подальшого залиття в AppStore та поширенню на пристрої на базі iOS. Для цього створимо два аналогічні проєкти з однаково підключеними бібліотеками як у Додатку У та порівняємо результуючий розмір додатків:

- Проєкт тільки з бібліотеками CocoaPods займає 28 мегабайт
- Проєкт у вигляді зв'язки KMM та використання існуючого рішення з CocoaPods займає 28.6 мегабайт
- Проєкт у вигляді зв'язки KMM та написаним власної інтеграції з SPM займає 12.8 мегабайт

Різниця у 0.6 мегабайт займає порожній фреймворк, котрий поставляється як базові реалізації примітивів, структур Kotlin для доступу в результуючому додатку. Даний варіант був продемонстрований через розуміння скільки займає місця KMM фреймворк без опису логіки в ньому.

Різниця між використанням CocoaPods та SPM полягає у підході до компонування сторонніх бібліотек. Якщо ми заглянемо в середину додатку, то можемо побачити папку Frameworks, де зберігаються всі залежності, котрі будуть компонуватися під час виконання додатку. У додатку CocoaPods, як на Додатку Ф, є наш фреймворку від KMM та всі підключені наші бібліотеки як окрема тека з об'єктним кодом.

Якщо переглянемо вміст додатку зібраним SPM у Додатку X, то побачимо набагато менше фреймворків – один з назвою нашого KMM фреймворків та декілька інших невеликих за обсягом. Найважчий фреймворк включає у себе всі зібрані залежності в один, тобто компілятор проаналізував всі підключені бібліотеки та з'єднав у один об'єктний файл позбувшись дублікату та не використовуваного коду.

Інші невеликі за розмір фреймворку, які займають менше 40 кілобайтів включають у себе всередині простий опис для інших залежностей, що для використання даної бібліотеки потрібно звернутися до іншої, оскільки весь код знаходиться там. Розглянемо наприклад фреймворк FBSDKCoreKit.framework, під час використання CocoaPods він має розмір 4 мегабайти та включає у собі весь код, а у SPM варіанті – займає всього 37 кілобайт з описом куди треба звертатися, якщо дану бібліотеки хтось використає.

Ще одним важливим критеріям порівнянням є швидкість збірки, оскільки під час неї, у випадку використання динамічного компонування ми не витрачаємо час, оскільки код потрапляє у додаток під час роботи. Насправді витрачаємо час тільки при першому запуску додатку, оскільки з вихідного коду повинні побудуватися всі фреймворки та в наступні рази просто використовуватися, якщо версія не змінилася, але у порівнянні зі статичним, там ми завжди перекомпільовуємо код бібліотек на кожний запуск. У випадку з SPM, певний додатковий час йде на побудову єдиного фреймворку з всіх підключених бібліотек, що може впливати на швидкість в порівнянні з використанням CocoaPods. Розглянемо швидкість збірки:

- Час запуску проєкту з нуля для CocoaPods складає 99 секунд, з яких 80 секунд це побудова KMM фреймворку та 19 збірка iOS додатку включаючи побудову всіх фреймворків
- Час запуску проєкту з нуля для SPM складає 117 секунди, з яких 93 секунди це побудова KMM фреймворку та 24 секунд на побудову єдиного фреймворку.

- Час запуску з інкременальною збіркою зі зміною в KMM фреймворку для CocoaPods складає 54 секунди, де виконується Gradle завдання близько 45 секунд та 9 секунд на збірку iOS додатку.
- Час запуску з інкременальною збіркою зі зміною в KMM фреймворку для SPM складає 83 секунд, де виконується Gradle завдання близько 70 секунд та 13 секунд на збірку iOS додатку.

Порівнявши дані значення, можна прийти до висновку, що написане рішення по інтеграції SPM в KMM проекти займає трошки більше часу ніж аналогічне рішення для CocoaPods, а саме, етап побудови фреймворку з Kotlin коду. Це обумовлено недоліками кешування котрі були вказані в розділі 3.2 та мають перспективи до вирішення поставленої проблематики. Швидкість збірки самого додатку під час використання SPM майже така сама як і у CocoaPods, оскільки ми формуємо постійно нову KMM залежність, що впливає на потребу у постійній перекомпіляції фреймворку. У випадку використання пакунку ми витрачаємо час на побудову одного єдиного фреймворку з усіх бібліотек, а у випадку подів – окремий KMM фреймворк.

ВИСНОВОК

У ході даного дослідження було проведено аналіз використання наявних підходів для додавання залежностей у Kotlin Multiplatform для iOS частини. Зокрема розглянуті методи у вигляді ручного додавання бібліотеки та інтеграцію CocoaPods, що дозволило оцінити переваги та недоліки кожного з варіантів.

Було розроблено програмний інструмент у вигляді Gradle плагіну для інтеграції Swift Package у Kotlin Multiplatform проєкти з урахуванням таких особливостей як аналіз підключених залежностей, формування готового пакунку та інтеграція у XCode.

Порівнюючи стороннє рішення для роботи з CocoaPods та написаний інструмент для Swift Package, можна виділити зменшений розмір результуючого додатку, що дозволяє зекономити місце на пристроях користувачів та зменшити час запуску. Серед недоліків інструменту зазначимо зменшену швидкість збірки проєкту, що впливає на продуктивність розробників. Однак, в роботі окреслено можливі способи усунути цей недолік в майбутньому.

СПИСОК ЛІТЕРАТУРИ

1. Petrova E. Kotlin Multiplatform Is Stable and Production-Ready [Електронний ресурс] / Ekaterina Petrova. – 2023. – Режим доступу до ресурсу: https://blog.jetbrains.com/kotlin/2023/11/kotlin-multiplatform-stable/#use-the-power-of-the-growing-kotlin-multiplatform-ecosystem.

2. Bundle a representation of the code and resources stored in a bundle directory on disk. [Електронний ресурс] – Режим доступу до ресурсу: https://developer.apple.com/documentation/foundation/bundle.

3. Bansal S. iOS Library, Bundle, and Frameworks [Електронний ресурс] / Shilpa Bansal // Walmart Global Tech Blog. – 2022. – Режим доступу до ресурсу: https://medium.com/walmartglobaltech/ios-library-bundle-and-frameworks-8fdc0aac6952.

4. Creating a standalone Swift package with Xcode. Bundle executable or shareable code into a standalone Swift package. [Електронний ресурс] – Режим доступу до ресурсу: https://developer.apple.com/documentation/xcode/creating-a-standalone-swift-package-with-xcode.

5. Linker Code Size Optimization for Native Mobile Applications [Електронний ресурс] / [G. Liu, U. Farooq, C. Zhao та ін.] // ACM Digital Library. – 2023. – Режим доступу до ресурсу: https://dl.acm.org/doi/abs/10.1145/3578360.3580256.

6. Slinky: Static Linking Reloaded [Електронний ресурс] / [C. Collberg, J. Hartman, S. Babu та ін.] // USENIX Annual Technical Conference. – 2005. – Режим доступу до ресурсу: https://www.usenix.org/legacy/events/usenix05/tech/general/full_papers/collberg/collberg.pdf.

7. Page Fault Handling in Operating System [Електронний ресурс] // GeeksforGeeks. – 2019. – Режим доступу до ресурсу: https://www.geeksforgeeks.org/page-fault-handling-in-operating-system/.

8. Dynamic Library Programming Topics. [Електронний ресурс]. – 2012. – Режим доступу до ресурсу: https://developer.apple.com/library/archive/documentation/DeveloperTools/Conceptual/DynamicLibraries/000-Introduction/Introduction.html.

9. Embedding Frameworks In An App [Електронний ресурс]. – 2016. – Режим доступу до ресурсу: https://developer.apple.com/library/archive/technotes/tn2435/_index.html.

10. TrueEngineering. Xcode and XCFrameworks — new format of packing frameworks [Електронний ресурс] / TrueEngineering // True Engineering. – 2019. – Режим доступу до ресурсу: https://medium.com/trueengineering/xcode-and-xcframeworks-new-format-of-packing-frameworks-ca15db2381d3.

11. Muschko B. Gradle in Action / Benjamin Muschko., 2014. – 480 c.

12. Джос О. Порівняльна характеристика крос-платформних рішень для побудови мобільних застосунків / Джос О.К. – 2023.

13. Джос О. Висхідний код Gradle плагіну [Електронний ресурс] / Олексій Джос. – 2024. – Режим доступу до ресурсу: https://github.com/NaUKMA-Programistich/Kotlin-Multiplatform-Swift-Package-Manager.

ДОДАТКИ

Додаток А. Повний PodSpec файл, котрий генерується для XCode

```
Pod::Spec.new do |spec|
    spec.name                     = 'shared'
    spec.version                  = '1.0'
    spec.homepage                 = 'Link to the Shared Module homepage'
    spec.source                   = { :http=> ''}
    spec.authors                  = ''
    spec.license                  = ''
    spec.summary                  = 'Some description for the Shared Module'
    spec.vendored_frameworks      = 'build/cocoapods/framework/shared.framework'
    spec.libraries                = 'c++'
    spec.ios.deployment_target = '16.0'
    spec.dependency 'FBAEMKit', '~>16.2.1'
    spec.dependency 'FBSDKCoreKit', '~>16.2.1'
    spec.dependency 'FBSDKGamingServicesKit', '~>16.2.1'
    spec.dependency 'FBSDKLoginKit', '~>16.2.1'
    spec.dependency 'FBSDKShareKit', '~>16.2.1'
    spec.dependency 'OAuthFramework'
    spec.dependency 'RealmSwift', '~>10'
    spec.dependency 'lottie-ios', '~>4.2.0'

    if !Dir.exist?('build/cocoapods/framework/shared.framework') || Dir.empty?('build/cocoapods/framework/shared.framework')
        raise "

        Kotlin framework 'shared' doesn't exist yet, so a proper Xcode project can't be generated.
        'pod install' should be executed after running ':generateDummyFramework' Gradle task:

            ./gradlew :shared:generateDummyFramework

        Alternatively, proper pod installation is performed during Gradle sync in the IDE (if Podfile location is set)"
    end

    spec.pod_target_xcconfig = {
        'KOTLIN_PROJECT_PATH' => ':shared',
        'PRODUCT_MODULE_NAME' => 'shared',
    }

    spec.script_phases = [
        {
            :name => 'Build shared',
            :execution_position => :before_compile,
            :shell_path => '/bin/sh',
            :script => <<-SCRIPT
                if [ "YES" = "$OVERRIDE_KOTLIN_BUILD_IDE_SUPPORTED" ]; then
                  echo "Skipping Gradle build task invocation due to OVERRIDE_KOTLIN_BUILD_IDE_SUPPORTED environment variable set to \"YES\""
                  exit 0
                fi
                set -ev
                REPO_ROOT="$PODS_TARGET_SRCROOT"
                "$REPO_ROOT/../gradlew" -p "$REPO_ROOT" $KOTLIN_PROJECT_PATH:syncFramework \
                    -Pkotlin.native.cocoapods.platform=$PLATFORM_NAME \
                    -Pkotlin.native.cocoapods.archs="$ARCHS" \
                    -Pkotlin.native.cocoapods.configuration="$CONFIGURATION"
            SCRIPT
        }
    ]

end
```

Додаток Б. Приклад використання CInterop

build.gradle.kts (:shared)

```kotlin
// Example kotlin multiplatform project gradle
plugins {
    kotlin("multiplatform")
}

kotlin {
    listOf(
        iosX64(),
        iosArm64(),
        iosSimulatorArm64()
    ).forEach { target ->
        target.compilations.getByName("main") {
            val cinterop by cinterops.creating {
                defFile("src/nativeInterop/example.def")
            }
        }
        target.binaries.all {
            linkerOpts("-L/*.framework", "-l*name")
        }
    }
}
```

example.def

```
headers = Example.h
package = Example
```

Додаток В. Приклад Swift Package

```swift
let package = Package(
    name: "OAuthFramework",
    platforms: [
        .iOS(.v13)
    ],
    products: [
        .library(
            name: "OAuthFramework",
            targets: ["OAuthFramework"])
    ],
    dependencies: [
        .package(url: "https://github.com/google/GoogleSignIn-iOS.git", from: "6.0.2"),
        .package(url: "https://github.com/facebook/facebook-ios-sdk.git", from: "16.2.1"),
//        .package(url: "https://github.com/AzureAD/microsoft-authentication-library-for-objc.git", from: "1.2.18"),
        .package(url: "https://github.com/apple/swift-log.git", from: "1.0.0"),
        .package(url: "https://github.com/realm/SwiftLint", branch: "main")
    ],
    targets: [
        .target(
            name: "OAuthFramework",
            dependencies: [
                .product(name: "GoogleSignIn", package: "GoogleSignIn-iOS"),
                .product(name: "Logging", package: "swift-log"),
                .product(name: "FacebookCore", package: "facebook-ios-sdk"),
                .product(name: "FacebookLogin", package: "facebook-ios-sdk")
            ],
            plugins: [.plugin(name: "SwiftLintPlugin", package: "SwiftLint")]
        )
    ]
)
```

Додаток Г. Приклад Kotlin DSL в CocoaPods

```
22    cocoapods { this: CocoapodsExtension
23        summary = "Some description for the Shared Module"
24        homepage = "Link to the Shared Module homepage"
25        version = "1.0"
26        ios.deploymentTarget = "16.0"
27        podfile = project.file("../iosApp/Podfile")
28        framework { this: Framework
29            baseName = "shared"
30            isStatic = true
31        }
32        pod( name: "RealmSwift") { this: CocoapodsExtension.CocoapodsDependency
33            version = "~>10"
34            extraOpts += listOf("-compiler-option", "-fmodules")
35        }
36    }
```

CocoapodsExtension.kt

```
181     }
182
183
        Add a CocoaPods dependency to the pod built from this project.
187     fun pod(name: String, configure: CocoapodsDependency.() -> Unit) {
188         // Empty string will lead to an attempt to create two podDownload tasks.
189         // One is original podDownload and second is podDownload + pod.name
190         require(name.isNotEmpty()) { "Please provide not empty pod name to avoid ambiguity" }
191         val dependency = project.objects.newInstance(CocoapodsDependency::class.java, name, name.asModuleName())
192         dependency.configure()
193         addToPods(dependency)
194     }
```

Додаток Ґ. Властивості Swift Package

```
7   open class KotlinMultiplatformSwiftPackageExtension(private val project: Project) {
8       internal var packageName: String? = null
9       internal var swiftVersion: Double? = null
10      internal var iosVersion: String? = null
11      internal var macosVersion: String? = null
12
13      internal var dependencies: List<Dependency> = emptyList()
14
        Dzhos Oleksii
15      fun packageName(packageName: String) {
16          this.packageName = packageName
17      }
18
        Dzhos Oleksii
19      fun swift(version: Double) {
20          this.swiftVersion = version
21      }
22
        Dzhos Oleksii *
23      fun dependencies(block: DependencyListBuilder.() -> Unit) {
24          this.dependencies = DependencyListBuilder().apply(block).build()
25      }
26
        Dzhos Oleksii
27      fun ios(version: String) {
28          this.iosVersion = version
29      }
30
        Dzhos Oleksii
31      fun macos(version: String) {
32          this.macosVersion = version
33      }
        Programistich +1
34      override fun toString(): String {
35          return "(packageName=$packageName, swiftVersion=$swiftVersion," +
36              "iosVersion=$iosVersion, dependencies=$dependencies)"
37      }
38  }
39
```

Додаток Д. Приклад Kotlin DSL для залежностей

```
34  class DependencyListBuilder {
35      private val dependencies = mutableListOf<Dependency>()
36
        Dzhos Oleksii +1
37      fun version(version: String, git: String) {
38          val dependency: Dependency = Dependency.Github.Version(git, version)
39          dependencies.add(dependency)
40      }
41
        Dzhos Oleksii +1
42      fun branch(branch: String, git: String) {
43          val dependency: Dependency = Dependency.Github.Branch(git, branch)
44          dependencies.add(dependency)
45      }
46
        Dzhos Oleksii
47      fun path(path: String) {
48          val dependency: Dependency = Dependency.Path(path)
49          dependencies.add(dependency)
50      }
51
        Dzhos Oleksii +1
52      fun url(path: String) {
53          val dependency: Dependency = Dependency.Github.Default(path)
54          dependencies.add(dependency)
55      }
56
        Dzhos Oleksii
57      fun build(): List<Dependency> {
58          return dependencies.toList()
59      }
60  }
```

Додаток Е. Реєстрація плагіну

```
build.gradle.kts (:spm:plugin)

20  gradlePlugin {  this: GradlePluginDevelopmentExtension
21      plugins {  this: NamedDomainObjectContainer<PluginDeclaration!>
22          create( name: "kmm-spm") {  this: PluginDeclaration
23              id = "kotlin-multiplatform-spm"
24              implementationClass = "com.programistich.spm.plugin.KotlinMultiplatformSwiftPackagePlugin"
25          }
26      }
27  }
```

```
KotlinMultiplatformSwiftPackagePlugin.kt

1   package com.programistich.spm.plugin
2
3   import ...
12
    Dzhos Oleksii +2
13  class KotlinMultiplatformSwiftPackagePlugin : Plugin<Project> {
        Dzhos Oleksii +2
14      override fun apply(project: Project) {...}
29  }
30
```

Додаток Є. Приклад налаштування SPM плагіну

```
1   plugins { this: PluginDependenciesSpecScope
2       id("kotlin-multiplatform-spm")
3   }
4
5   swiftPackage { this: KotlinMultiplatformSwiftPackageExtension
6       packageName("TestSPM")
7       swift( version: 5.6)
8       ios( version: "15")
9       macos( version: "10_15")
10
11      dependencies { this: DependencyListBuilder
12          version("3.3.0",  git: "https://github.com/airbnb/lottie-ios.git")
13          url( path: "https://github.com/facebook/facebook-ios-sdk")
14          url( path: "https://github.com/apple/swift-log")
15          url( path: "https://github.com/realm/realm-swift.git")
16          url( path: "https://github.com/NaUKMA-Programistich/OAuthFramework.git")
17          branch("main",  git: "https://github.com/gonzalezreal/swift-markdown-ui")
18      }
19  }
```

Додаток Ж. Код реалізація інтерфейсу плагіну

```
13  class KotlinMultiplatformSwiftPackagePlugin : Plugin<Project> {
        ± Dzhos Oleksii +2
14      override fun apply(project: Project) {
15          val spmExtension = project
16              .extensions
17              .create<KotlinMultiplatformSwiftPackageExtension>(
18                  name = SwiftPackageConstants.GRADLE_DSL,
19                  constructionArguments = arrayOf(project)
20              )
21
22          project.afterEvaluate { this: Project
23              val kmmExtension = project.findKotlinMultiplatformExtension() ?: return@afterEvaluate
24              val frameworkName = spmExtension.packageName + PREFIX_FRAMEWORK
25              project.generateXCFramework(kmmExtension, frameworkName)
26              project.registerCreateSPM(spmExtension, frameworkName, kmmExtension)
27          }
28      }
29  }
```

Додаток З. Підключення формування XCFramework

```
8   internal fun Project.generateXCFramework(
9       kmmExtension: KotlinMultiplatformExtension,
10      frameworkName: String,
11  ) {
12      val xcFrameworkConfig = XCFrameworkConfig( project: this, frameworkName)
13
14      val targets = kmmExtension
15          .targets NamedDomainObjectCollection<KotlinTarget>
16          .toList() List<KotlinTarget!>
17          .filterIsInstance<KotlinNativeTarget>()
18
19      kmmExtension.apply { this: KotlinMultiplatformExtension
20          targets.forEach { it: KotlinNativeTarget
21              it.binaries.framework { this: Framework
22                  baseName = frameworkName
23                  xcFrameworkConfig.add(this)
24              }
25          }
26      }
27  }
28
```

Додаток И. Перевірка підтримки MacOS таргету.

```
9   internal fun Project.checkMacosTarget(
10      spmExtension: KotlinMultiplatformSwiftPackageExtension,
11      kmmExtension: KotlinMultiplatformExtension
12  ) {
13      val targets = kmmExtension
14          .targets NamedDomainObjectCollection<KotlinTarget>
15          .toList() List<KotlinTarget!>
16          .filterIsInstance<KotlinNativeTarget>() List<KotlinNativeTarget>
17          .map { it.konanTarget } List<KonanTarget>
18          .filter { it.family == Family.OSX }
19
20      val isMacosTargetKMP = targets.isNotEmpty()
21      val isMacosTargetSPM = spmExtension.macosVersion != null
22
23      if (isMacosTargetSPM && !isMacosTargetKMP) {
24          error("macos target is defined in SPM, but not defined in KMM")
25      }
26  }
27
```

Додаток І. Завантаження вихідного коду SPM

```
7   internal fun Project.downloadSwiftPackageTools() {
8       if (project.buildDir.resolve(SPM_PATH).exists()) {
9           return
10      }
11
12      exec { this: ExecSpec
13          workingDir = project.buildDir
14          commandLine = listOf("git", "clone", "https://github.com/apple/${SPM_PATH}")
15          standardOutput = ByteArrayOutputStream()
16      }
17  }
18
```

Додаток Ї. Маніфест парсеру

```
3   import PackageDescription
4
5   let package = Package(
6       name: "swift-package-tools",
7       platforms: [ .macOS(.v13) ],
8       dependencies: [
9           .package(name: "swift-package-manager", path: "some-path"),
10          .package(url: "https://github.com/apple/swift-log.git", from: "1.0.0"),
11      ],
12      targets: [
13          .executableTarget(
14              name: "swift-package-tools",
15              dependencies: [
16                  .product(name: "SwiftPM", package: "swift-package-manager"),
17                  .product(name: "Logging", package: "swift-log")
18              ]
19          ),
20      ]
21  )
```

Додаток Й. Отримання всієї інформації про залежність

```
let observability = ObservabilitySystem({ _, _ in })
let workspace = try Workspace(forRootPackage: packagePath)
let manifest = try await workspace.loadRootManifest(at: packagePath, observabilityScope: observability.topScope)
let rootPackage = try await workspace.loadRootPackage(at: packagePath, observabilityScope: observability.topScope)
let graph = try workspace.loadPackageGraph(rootPath: packagePath, observabilityScope: observability.topScope)
```

Додаток К. Парсер Swift Package

```
6    private let FROM_ARG : String  = "--from="
7    private let TO_ARG : String  = "--to="
8
9    let logger = Logger(label: "SwiftPackageTools")
10
11   @main
12   @available(macOS 13, iOS 15, tvOS 15, watchOS 8, *)
13   struct SwiftPackageTools {
14       static func main() async throws {
15           let arguments : [String]  = CommandLine.arguments
16           logger.info("Arguments \(arguments)")
17           let jsonPath = URL(fileURLWithPath: toArg)
18
19           logger.info("SPM path \(packagePath)")
20           logger.info("JSON path \(jsonPath)")
21
22           var parserTargets: [TargetParser] = []
23           var parserProducts: [ProductParser] = []
24
25           let observability = ObservabilitySystem({ _, _ in })
26           let workspace = try Workspace(forRootPackage: packagePath)
27           let manifest = try await workspace.loadRootManifest(at: packagePath, observabilityScope: observability.topScope)
28           let rootPackage = try await workspace.loadRootPackage(at: packagePath, observabilityScope: observability.topScope)
29           let graph = try workspace.loadPackageGraph(rootPath: packagePath, observabilityScope: observability.topScope)
30
31           let targets = manifest.targets
32           let commit = rootPackage.identity.description
33           let depsName = manifest.displayName
34
35           for target in targets.filter({ !$0.isTest }) {
36               var deps: [DependencyParser] = []
37
38 >             for dependency in target.dependencies {…}
83               parserTargets.append(TargetParser(name: target.name, dependencies: deps))
84           }
85
86           for product in rootPackage.products.filter({ $0.type != .test }) {
87               parserProducts.append(ProductParser(name: product.name))
88           }
89
90           let parser = Parser(products: parserProducts, targets: parserTargets, commit: commit, name: depsName)
91
92           do {
93               let encoder : JSONEncoder  = JSONEncoder()
94               encoder.outputFormatting = .prettyPrinted
95               let data = try encoder.encode(parser)
96
97               try data.write(to: jsonPath, options: [.atomicWrite])
98
99               logger.info("Success")
100          } catch {
101              logger.error("Error on write to json \(error.localizedDescription)")
102          }
103      }
```

Додаток Л. Робота з GitHub API

```
12  object GithubApi {
        Programistich
13      suspend fun getDefaultBranch(githubData: GithubData): String {
14          val url = "https://api.github.com/repos/${githubData.organization}/${githubData.repository}"
15          val response: GithubDefaultDto = httpClient.get(url).body()
16          return response.defaultBranch
17      }
18
        Programistich
19      suspend fun getCommitByBranch(githubData: GithubData, branch: String): String {
20          val url = "https://api.github.com/repos/${githubData.organization}/${githubData.repository}/branches/$branch"
21          val response: GithubBranchDto = httpClient.get(url).body()
22          return response.commit.sha
23      }
24
        Programistich
25      suspend fun getCommitByTag(githubData: GithubData, tag: String): String {
26          val url = "https://api.github.com/repos/${githubData.organization}/${githubData.repository}/tags"
27          val response: List<GithubTagDto> = httpClient.get(url).body()
28          return response.first { it.name == tag }.commit.sha
29      }
30
        Programistich
31      suspend fun downloadByCommit(githubData: GithubData, commit: String, file: File) {
32          val url = "https://api.github.com/repos/${githubData.organization}/${githubData.repository}/zipball/$commit"
33          val response = httpClient.get(url).bodyAsChannel()
34          response.copyAndClose(file.writeChannel())
35      }
36  }
```

Додаток М. Розархівування

```
24  fun File.unzip(destDirectory: String) {
25      File(destDirectory).mkdirs()
26
27      ZipFile(this).use { zip ->
28          zip.entries().asSequence().forEach { entry ->
29              val entryPathSegments = entry.name.split(File.separator).drop( n: 1)
30              processZipEntry(entryPathSegments, destDirectory, entry, zip)
31          }
32      }
33  }
34
    Programistich
35  private fun processZipEntry(
36      entryPathSegments: List<String>,
37      destDirectory: String,
38      entry: ZipEntry,
39      zip: ZipFile
40  ) {
41      if (entryPathSegments.isNotEmpty()) {
42          val filePath =
43              destDirectory + File.separator + entryPathSegments.joinToString(File.separator)
44          if (!entry.isDirectory) {
45              zip.getInputStream(entry).use { input ->
46                  extractFile(input, filePath)
47              }
48          } else {
49              File(filePath).mkdirs()
50          }
51      }
52  }
53
    Programistich
54  private fun extractFile(inputStream: InputStream, destFilePath: String) {
55      BufferedOutputStream(FileOutputStream(destFilePath)).use { bos ->
56          val bytesIn = ByteArray(BUFFER_SIZE)
57          var read: Int
58          while (inputStream.read(bytesIn).also { read = it } != -1) {
59              bos.write(bytesIn, 0, read)
60          }
61      }
62  }
```

Додаток Н. Парсинг всіх залежностей

```
12  internal fun Project.parseDependencies() {
13      DependencyStorage.getAll().forEach { (dependency, path) ->
14          processParseDependency(path, dependency)
15      }
16  }
17
    Programistich
18  private fun Project.processParseDependency(path: String, dependency: Dependency) {
19      val commitName = path.substringAfterLast( delimiter: "/")
20      val jsonDeps = File(path, "$commitName.json")
21
22      val fromDeps = File(path).absolutePath.replace( oldValue: " ", newValue: "\\ ")
23      val toDeps = jsonDeps.absolutePath.replace( oldValue: " ", newValue: "\\ ")
24
25      val cli = listOf("bash", "-c", "swift run swift-package-tools --from=$fromDeps --to=$toDeps")
26      val cliString = cli.joinToString( separator: " ")
27
28      val workDirectory = File(project.buildDir, SwiftPackageConstants.SPM_TOOLS_PATH)
29
30      logger.warn( msg: "Processing dependencies for $dependency")
31      logger.warn( msg: "Command: $cliString")
32      logger.warn( msg: "Work directory: $workDirectory")
33
34      val output = ByteArrayOutputStream()
35
36      exec { this: ExecSpec
37          workingDir = workDirectory
38          commandLine = cli
39          standardOutput = output
40      }
41      logger.warn( msg: "Output: $output")
42      val parser = Json.decodeFromString<Parser>(jsonDeps.readText())
43      DependencyStorage.setParser(dependency, parser)
44  }
```

Додаток О. Результуючий приклад пакунку

```swift
// swift-tools-version:5.6
// First step
import PackageDescription

let package = Package(
    name: "TestSPM",
    platforms: [
        .iOS(.v15),
        .macOS(.v10_15),
    ],
    products: [
        .library(
            name: "TestSPM",
            targets: ["TestSPM", "TestSPMFramework"]
        ),
    ],
    // Second step
    dependencies: [
        .package(url: "https://github.com/gonzalezreal/swift-markdown-ui", .branch("main")),
        .package(url: "https://github.com/NaUKMA-Programistich/OAuthFramework.git", .revision("39e4
        .package(url: "https://github.com/apple/swift-log", .revision("c1ff8be7bc1d877eb24d1ce76b1d
        .package(url: "https://github.com/realm/realm-swift.git", .revision("6f8d5e390139c32b5b27d5
        .package(url: "https://github.com/airbnb/lottie-ios.git", from: "3.3.0"),
        .package(url: "https://github.com/facebook/facebook-ios-sdk", .revision("7295c425d8ca789a78
    ],
    // Third step
    targets: [
        .binaryTarget(
            name: "TestSPMFramework",
            path: "../XCFrameworks/debug/TestSPMFramework.xcframework"
        ),
        .target(
            name: "TestSPM",
            dependencies: [
                .product(name: "MarkdownUI", package: "swift-markdown-ui"),
                // Another dependencies
                .product(name: "FacebookGamingServices", package: "facebook-ios-sdk"),
            ]
        ),
    ]
)
```

Додаток П. Shell скрипт для оновлення пакунку.

```kotlin
private fun generateContent(
    spmBuildPath: String,
    rootFilePath: String,
    subproject: String
): String {
    val rootGradleSh = "$" + "ROOT_GRADLE"
    val subprojectSh = "$" + "SUBPROJECT"
    val gradleTaskSh = "$" + "GRADLE_TASK"
    val configuration = "$" + "CONFIGURATION"

    return """
        #!/usr/bin/env sh

        # Add this script in pre build phase
        # chmod +x $spmBuildPath && $spmBuildPath

        set -ev

        export SUBPROJECT="$subproject"
        export GRADLE_TASK="$gradleTaskSh"

        cd $rootFilePath

        ./gradlew $rootGradleSh $subprojectSh:createSPM -PbuildType=$configuration
    """.trimIndent()
}
```

Додаток Р. Валідація пакунку

```kotlin
internal fun Project.validateSwiftPackage() {
    val cli = listOf("bash", "-c", "swift package describe")

    val result = exec {
        workingDir = File(project.buildDir, SwiftPackageConstants.FOLDER)
        commandLine = cli
        standardOutput = ByteArrayOutputStream()
    }
    check(result.exitValue == 0) { "Swift package validation failed" }
}
```

Додаток С. Приклад CInterop

```kotlin
listOf(
    iosX64(),
    iosArm64(),
    iosSimulatorArm64(),
    macosX64(),
    macosArm64()
).forEach {
    it.compilations.getByName("main") {
        val library by cinterops.creating {
            defFile("path.def")
        }
    }

    it.binaries.all {
        linkerOpts("-L/path", "-lbname")
    }
}
```

Додаток Т. Повний набір інструкцій

```
      Programistich +1 *
9     internal fun Project.registerCreateSPM(
10        spmExtension: KotlinMultiplatformSwiftPackageExtension,
11        frameworkName: String,
12        kmmExtension: KotlinMultiplatformExtension
13    ) {
14        val project = this
15        tasks.register( name: "createSPM") { this: Task
16            description = "Create Swift Package for Kotlin Multiplatform Mobile"
17            when (project.getBuildType()) {
18                BuildType.DEBUG -> dependsOn( ...paths: "assemble${frameworkName}DebugXCFramework")
19                BuildType.RELEASE -> dependsOn( ...paths: "assemble${frameworkName}ReleaseXCFramework")
20            }
21
22            doLast { this: Task
23                project.checkMacosTarget(spmExtension, kmmExtension)
24
25                // SPM Tools
26                project.downloadSwiftPackageTools()
27                project.generateSwiftPackageToolsParser()
28
29                // Work with dependencies
30                project.downloadDependencies(dependencies = spmExtension.dependencies)
31                project.parseDependencies()
32
33                val swiftPackage = project.createSwiftPackageModel(spmExtension, frameworkName)
34                project.generateSPMBuildFolder(swiftPackage)
35                project.generatePreBuildScript()
36
37                // Validate Swift Package
38                project.validateSwiftPackage()
39            }
40        }
41    }
42
      Dzhos Oleksii +1
43    internal fun Project.getBuildType(): BuildType {
44        val rawBuildType: String? = project.findProperty(SwiftPackageConstants.PROP_TYPE_BUILD) as String?
45        return BuildType.fromString(rawBuildType) ?: BuildType.RELEASE
46    }
47
```

Додаток У. Назви бібліотек для порівняння

```
51    swiftPackage {
52        packageName("TestSPM")
53        swift(5.6)
54        ios("15")
55        macos("10_15")
56
57        dependencies {
58            url("https://github.com/realm/realm-swift.
59            url("https://github.com/airbnb/lottie-ios.
60            url("https://github.com/facebook/facebook-
61            url("https://github.com/apple/swift-log")
62            url("https://github.com/NaUKMA-Programisti
63        }
64    }
65
```

```
21        cocoapods {
22            pod("RealmSwift")
23            pod("lottie-ios")
24            pod("FBAEMKit")
25            pod("FBSDKShareKit")
26            pod("FBSDKLoginKit")
27            pod("FBSDKGamingServicesKit")
28            pod("FBSDKCoreKit")
29            pod("OAuthFramework")
30        }
31
32
33
34
35
```

Додаток Ф. Вміст додатку з використанням CocoaPods

Name	Date	Size	Kind
_CodeSignature	29.04.2024	--	Folder
embedded.mobileprovision	29.04.2024	93 KB	Develo...Profile
Frameworks	29.04.2024	--	Folder
AppAuth.framework	29.04.2024	--	Folder
FBAEMKit.framework	29.04.2024	--	Folder
FBSDKCoreKit_Basics.framework	29.04.2024	--	Folder
FBSDKCoreKit.framework	29.04.2024	--	Folder
FBSDKGamingServicesKit.framework	29.04.2024	--	Folder
FBSDKLoginKit.framework	29.04.2024	--	Folder
FBSDKShareKit.framework	29.04.2024	--	Folder
GoogleSignIn.framework	29.04.2024	--	Folder
GTMAppAuth.framework	29.04.2024	--	Folder
GTMSessionFetcher.framework	29.04.2024	--	Folder
Logging.framework	29.04.2024	--	Folder
Lottie.framework	29.04.2024	--	Folder
_CodeSignature	29.04.2024	--	Folder
Info.plist	29.04.2024	784 bytes	Property List
Lottie	29.04.2024	3,8 MB	Unix Ex...ble File
OAuthFramework.framework	29.04.2024	--	Folder
Realm.framework	29.04.2024	--	Folder
_CodeSignature	29.04.2024	--	Folder
Info.plist	29.04.2024	784 bytes	Property List
Realm	29.04.2024	9,7 MB	Unix Ex...ble File
realm_objc_privacy.bundle	29.04.2024	1 KB	bundle
RealmSwift.framework	29.04.2024	--	Folder
_CodeSignature	29.04.2024	--	Folder
Info.plist	29.04.2024	794 bytes	Property List
realm_swift_privacy.bundle	29.04.2024	1 KB	bundle
RealmSwift	29.04.2024	3,8 MB	Unix Ex...ble File
Info.plist	29.04.2024	1 KB	Property List
iosApp	29.04.2024	703 KB	Unix Ex...ble File
PkgInfo	29.04.2024	8 bytes	Document

Додаток Х. Вміст додатку з використанням SPM

Name	Date	Size	Kind
_CodeSignature	04.03.2024	--	Folder
embedded.mobileprovision	04.03.2024	93 KB	Develo...Profile
Frameworks	04.03.2024	--	Folder
FBAEMKit.framework	04.03.2024	--	Folder
_CodeSignature	04.03.2024	--	Folder
FBAEMKit	04.03.2024	35 KB	Unix Ex...ble File
Info.plist	04.03.2024	793 bytes	Property List
FBSDKCoreKit_Basics.framework	04.03.2024	--	Folder
_CodeSignature	04.03.2024	--	Folder
FBSDKCoreKit_Basics	04.03.2024	35 KB	Unix Ex...ble File
Info.plist	04.03.2024	816 bytes	Property List
FBSDKCoreKit.framework	04.03.2024	--	Folder
_CodeSignature	04.03.2024	--	Folder
FBSDKCoreKit	04.03.2024	35 KB	Unix Ex...ble File
Info.plist	04.03.2024	801 bytes	Property List
FBSDKGamingServicesKit.framework	04.03.2024	--	Folder
_CodeSignature	04.03.2024	--	Folder
FBSDKGamingServicesKit	04.03.2024	35 KB	Unix Ex...ble File
Info.plis KMMSPMExample.app	04.03.2024	792 bytes	Property List
FBSDKLoginKit.framework	04.03.2024	--	Folder
_CodeSignature	04.03.2024	--	Folder
FBSDKLoginKit	04.03.2024	35 KB	Unix Ex...ble File
Info.plist	04.03.2024	803 bytes	Property List
FBSDKShareKit.framework	04.03.2024	--	Folder
_CodeSignature	04.03.2024	--	Folder
FBSDKShareKit	04.03.2024	35 KB	Unix Ex...ble File
Info.plist	04.03.2024	803 bytes	Property List
TestSPMFramework.framework	04.03.2024	--	Folder
_CodeSignature	04.03.2024	--	Folder
Info.plist	04.03.2024	1 KB	Property List
TestSPMFramework	04.03.2024	2,8 MB	Unix Ex...ble File
Info.plist	04.03.2024	1 KB	Property List
KMMSPMExample	04.03.2024	9,6 MB	Unix Ex...ble File
PkgInfo	04.03.2024	8 bytes	Document
Realm_Realm.bundle	04.03.2024	1 KB	bundle
Realm_RealmSwift.bundle	04.03.2024	1 KB	bundle