Міністерство освіти і науки України

Національний університет «Києво-Могилянська академія»

Факультет інформатики

Кафедра інформатики

Кваліфікаційна робота

освітній ступінь - бакалавр

на тему: **«Автоматизований аналіз рівня використання трафіку мобільними застосунками»**

Виконав: студент 4-го року навчання,
Спеціальності
122 «Комп’ютерні науки»

Студента Кучеренка Данііла

Керівник Франків О.О

Старший викладач

«18» травня 2025 р.

Київ – 2025

Національний університет «Києво-Могилянська академія»

Факультет інформатики

Кафедра інформатики

Освітній ступінь бакалавр

Спеціальність 122 «Комп'ютерні науки»

Освітня програма бакалавр

ЗАТВЕРДЖУЮ

Завідувач кафедри інформатики

Гороховський С. С.

„____”_____2025 р.

ЗАВДАННЯ

ДЛЯ КВАЛІФІКАЦІЙНОЇ РОБОТИ СТУДЕНТУ

Кучеренко Даніілу

1. Тема роботи «**Автоматизований аналіз рівня використання трафіку мобільними застосунками**», керівник Франків Олександр Олександрович, старший викладач

2. Строк подання студентом роботи 30 травня 2025

3. План роботи

   Анотація

   Вступ

   Розділ 1 Інструменти аналізу коду

   1.1 Загальний опис

   1.2 Аналізатор трафіку від Apple

   1.3 Бібліотека SwiftSyntax

ГРАФІК ПІДГОТОВКИ КВАЛІФІКАЦІЙНОЇ РОБОТИ ДО ЗАХИСТУ

№ з/п	*1.1.* ПЕРЕЛІК РОБІТ	Термін виконання	Дата ознайомлення наукового керівника	Підпис наукового керівника	Примітки
1.	Вибір теми, затвердження її на засіданні кафедри та закріплення наукового керівника Узгодження календарного графіка підготовки кваліфікаційної роботи. Ознайомлення студента з критеріями оцінювання кваліфікаційної роботи (п. 8.5).	30 вересня 2024			
2.	Вивчення джерел літератури, матеріалів архівів, періодичних видань, збір та узагальнення фактів, даних	1 жовтня 2024 – 8 жовтня 2024			
3.	Складання плану каліф. роботи та узгодження з науковим керівником	18 листопада 2024			
4.	Написання розділів роботи	2 листопада 2024 – 01 березня 2025			
5.	Проміжний контроль виконання роботи	01 лютого 2025			
6.	Написання кваліфікаційної роботи в цілому, ознайомлення з її першим варіантом наукового керівника	11 січня 2025 – 29 березня 2025			
	Розділ 1 (постановка проблеми, теоретичні основи, огляд літературних джерел)	25 січня 2025			
	Розділ 2 (аналітично-дослідницька частина)	01 березня 2025			
	Розділ 3 (проектно-рекомендаційна частина)	28 березня 2025			
7.	Повне завершення написання кваліфікаційної роботи, оформлення її згідно з вимогами й подання на відгук науковому керівнику	20 квітня 2025 – 5 травня 2025			
8.	Вдосконалення та імплементація покращеної версії методики оцінки коду на наявність антипатернів	10 травня 2025 – 20 травня 2025			
9.	Створення слайдів для доповіді та написання доповіді.	15 травня 2025			
10.	Подання кваліфікаційної роботи для перевірки письмових робіт студентів НаУКМА на відповідність вимогам академічної доброчесності,	30 травня 2025			
11.	Публічний захист кваліфікаційної роботи перед екзаменаційною комісією	згідно з розкладом роботи ЕК			

Графік узгоджено 30 вересня 2025 р.

Науковий керівник Франків Олександр Олександрович

Виконавець кваліфікаційної роботи Кучеренко Даніїл

ЗМІСТ

Анотація

Кваліфікаційна робота присвячена дослідженню антипатернів у коді мобільних застосунків для операційної системи iOS, що призводять до нераціонального використання інтернет-трафіку. У роботі детально розглянуто принципи виявлення антипатернів. Особливу увагу було приділено адаптації існуючих досліджень для оцінки мережевих операцій у Swift, зокрема, налаштування HTTP-запитів, кешування, полінгу, та конфігурація фонових задач.

У рамках роботи розроблено інструмент на базі Swift Package, який проводить аналіз Swift коду за допомогою бібілотеки SwiftSyntax. Інструмент використовує Swift Package Plugin для інтеграції в процес збірки проектів через Xcode.

Вступ

Зростання доступності інструментів для розробки та розповсюдження мобільних застосунків без структурного та детального огляду застосунку дозволив недосвідченим розробникам програмного забезпечення легко поширювати власні застосунки, що може торкнутись значної кількості користувачів. Як наслідок, ринок додатків налічує велику кількість застосунків, доступних до використання. На момент 2024-го року, відповідно до дослідження Tekrevol [1], App Store має близько 1.87 мільйона застосунків, а Google Play - 2.8 мільйони. Обрати якісно написаний застосунок із такої великої кількості – непроста задача, тим більше, якщо розробники цього програмного забезпечення описують застосунок лише з того боку, як він може бути використаний. Споживання батареї чи мобільного трафіку застосунком зазвичай не викликає занепокоєння у користувачів, однак, коли це негативно впливає на досвід використання, наприклад, спричиняє вичерпання тарифного плану, воно стає ключовим фактором вибору [2]. Відповідно до дослідження Ericsson [3], середні показники використання мобільного трафіку користувачем, на період Q4 2024-го року сягають 20 гігабайтів на місяць, у той час, як у 2014-му році, середні показники були лише 7 гігабайтів [4]. Таким чином, актуальність роботи зумовлена потребою в оптимізації використання трафіку, що сприятиме підвищенню якості застосунків, які користувачі отримуватимуть на виході в магазині застосунків.

Для вдосконалення процесу створення застосунків було вирішено провести дослідження, спрямоване на аналіз мобільних застосунків та їхнього впливу на обсяг споживаного трафіку. Це є значущою частиною додатка, оскільки напряму впливає на користувача і його користувацький досвід. На сьогодні багато застосунків напряму працюють із мережею для синхронізації, оновлення або завантаження інформації, тож якісне використання трафіку в застосунку є критичним.

Об’єктом дослідження є використання трафіку мобільними застосунками на мобільній операційній системі iOS. Дослідження фокусується на iOS через її значну частку ринку, що складає понад 20 відсотків глобального ринку

смартфонів у 2024 році, за даними IDC [5] та унікальні особливості управління ресурсами, які впливають на поведінку застосунків.

Предметом дослідження цієї кваліфікаційної роботи є конкретні антипатерни, які можна зустріти в коді мобільних застосунків, що можуть впливати на використання трафіку. Антипатерни, що стосуються безпосереднього керування трафіком у програмному коді, охоплюють кілька критичних аспектів. До них належить неоптимальне використання додаткових параметрів HTTP-запитів, неправильна конфігурація полінгу, а також помилки в налаштуванні кешування. Крім того, ці антипатерни можуть виявлятись у неефективній роботі фонових запитів, які використовуються для завантаження даних, синхронізації або оновлення інформації про користувача, наприклад, передачі геопозиції.

Метою цієї кваліфікаційної роботи є дослідження рівня використання трафіку, що базується на конкретних антипатернах у коді, що призводять до надмірного або нераціонального використання трафіку. Насамперед, для досягнення поставленої мети треба було визначити ці патерни та зрозуміти як їх можна виявити в коді. Для аналізу було використано інші дослідження як для операційної системи iOS, так і для Android, оскільки патерни, що визначають роботу з трафіком, є схожими на обох платформах [35]. Наступною задачею було визначити чи описані антипатерни є досі актуальними, оскільки Apple – компанія, що слідкує за розвитком своїх продуктів, а мова програмування Swift знаходиться у відкритому доступі, тож будь-які проблемні аспекти написання коду можуть бути покращені за рахунок спільноти. Щоб оцінити критичність виявлених антипатернів, необхідно було розробити чіткі критерії для визначення їхнього впливу на якість і продуктивність коду. Після цього було вибрано метод написання програми, яка допомагала б розробникам в передчасному виявленні факторів, що впливають на використання трафіку та подальшій можливості їхнього виправлення.

Перший розділ присвячено огляду інструментів, що спрощує процес аналізу споживання мобільного трафіку в кодовій базі мобільного застосунку. У

ньому розглянуто методи та підходи, що дозволяють розробникам отримувати детальну інформацію про використання трафіку на етапі розробки.

У другому розділі акцентовано увагу на типових антипатернах, що виникають під час роботи з мережевим трафіком у мобільних застосунках на платформі iOS

Третя частина буде сфокусована над описом створення готового аналізатора, який буде допомагати усувати розглянуті раніше антипатерни. Ця частина буде присвячена використанню наявних методів аналізу коду та інтеграції аналізаторів у середовище розробки.

Результатом дослідження є розроблений інструмент, який дає змогу розробнику більш ефективно підходити до питання налаштування трафіку в мобільному застосунку та запобігти появі досліджених антипатернів в коді. Цей інструмент спростить розробку нових мобільних застосунків та покращить взаємодію користувача з додатком, закладаючи фундамент застосунку правилами виявлення антипатернів.

Інструмент є зручним та інтуїтивно зрозумілим, що спрощує розробку. Він може бути інтегрований у вже наявні проекти з метою покращення якості коду та логіки, пов’язаної з використанням трафіку застосунком.

РОЗДІЛ 1. Інструменти аналізу коду

1.1. Загальний опис

У процесі розробки мобільного застосунку, розробники працюють із налаштуванням використання трафіку. Цей етап є важливим, оскільки напряму впливає на продуктивність, енергоефективність та економічність самого трафіку. Незважаючи на це, основне середовище розробки, Xcode, пропонує лише обмежений набір інструментів, який не має достатніх метрик для оцінки використання трафіку задля передчасного виявлення проблем, із якими можна стикнутись під час створення застосунку. Такі інструменти дозволяють проаналізувати загальне використання трафіку, але не виявляють конкретних проблем у коді, пов'язаних із нераціональним використанням трафіку.

Тож виникає задача створення цих інструментів власноруч. Подальший аналіз може бути здійснений або на рівні коду користувача, або безпосередньо на використанні застосунком трафіку. На основі отриманих даних, користувач сам визначає, що може бути проблемою надмірного використання трафіку. Обидва підходи дають змогу розробнику отримати більше інформації про використання трафіку, але не є взаємозамінними. Один із них дозволяє виявляти помилки, безпосередньо, під час написання програми, а інший – під час роботи із застосунком.

Аналіз програмного коду поділяють на два види: статичний та динамічний. Статичний аналіз є необхідним для передчасного виявлення проблем, оскільки перевіряє код без його виконання і дозволяє виявляти конкретні проблеми, такі як налаштування кешування, використання застарілих бібліотек або методів та налаштування URLSession. Динамічний аналіз, зі свого боку, вимагає запуску застосунку, адже він аналізує лише фактичне використання трафіку: швидкість передачі даних, час між посиланням запиту та надходженням відповіді сервера.

Очевидно, дізнатись про проблеми свого застосунку є бажаним на моменті написання коду. Саме тому розробка таких аналізаторів є цінною та актуальною задачею для розробників, адже буде створювати гнучкі інструменти, які можна буде налаштувати залежно від проєкту. Вони будуть не лише спрощувати роботу

розробнику надаючи підказки, а й підвищувати якість вихідних продуктів, які потім будуть доступні користувачам.

1.2. Аналізатор трафіку від Apple

У 2021-му році Apple представила свій інструмент для аналізу трафіку застосунку. Цей інструмент є частиною Instruments – потужний та якісний додаток по аналізу та тестуванню продуктивності [6].

Він фокусується на зборі детальної інформації, аналізі часу запиту та швидкості повернення відповіді сервером і далі вже збирає цю інформацію для зручного представлення користувачу на графічному інтерфейсі.

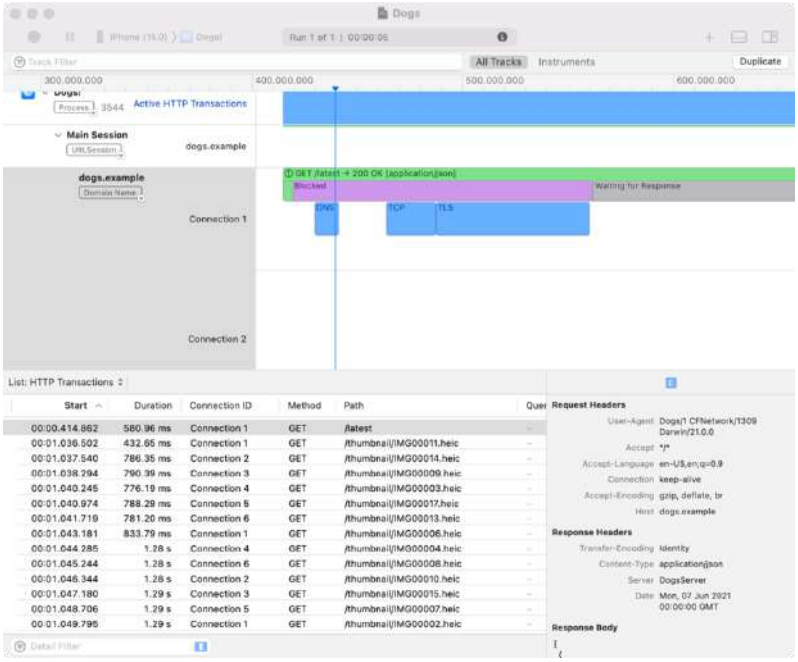

Рис. 1.1 Графічний інтерфейс Network Instruments by Apple

Використання інструменту надає такі переваги:

- **Аналіз часу запитів**: кожен запит має інформацію про час надсилання запиту, встановлення з'єднання з сервером та подальшим відправленням відповіді.
- **Типи запитів**: містить інформацію про метод запиту (GET, POST, PUT, DELETE), HTTP заголовки та розмір даних у відповіді та самому запиті.
- **Зручна візуалізація**: інструмент дає змогу наочно розглядати надіслані запити відносно часу, що є зручним задля виявлення неправильно налаштованих таймерів, наприклад, для полінгу або фонових задач.

Як зазначалося раніше, цей аналізатор є суто динамічним, тобто працює лише після запуску додатка, що ускладнює виявлення проблемних місць застосунку конкретно в коді, а радше є аналізатором постфактум. Не дивлячись на зручності, що пропонує цей інструмент, він не надає розробнику розв'язання проблеми, а залишає місце для власного аналізу даних і пошуку місця для усунення проблеми з використанням трафіку. Також він не інтегрується напряму в Xcode і саме тому, не може бути використаним, безпосередньо при написанні коду.

1.3. Бібліотека SwiftSyntax

Враховуючи, що використання динамічних аналізаторів не завжди є зручним, то виникає потреба в дослідженні бібліотек та фреймворків, що дозволятимуть аналізувати сам код застосунку під час його написання. Одним з таких є SwiftSyntax.

SwiftSyntax – це набір бібліотек з відкритим вихідним кодом, що працюють над репрезентацією коду користувача у вигляді SwiftSyntax дерева [7]. Вона дозволяє декодувати, аналізувати, генерувати та редагувати користувацький код. Бібліотека базується на представленні кожної частини коду у вигляді вузлів синтаксичного дерева, де кожен вузол представляє собою структуру, яка відповідає за певний тип у коді, наприклад такі, як функція, змінна, клас тощо).

SwiftSyntax використовується для створення великої кількості лінтерів, що також допомагають розробнику при написанні коду. Одним з найбільш відомих є SwiftLint, який допомагає дотримуватись певного стилю та конвенцій проєкту задля досягнення стандартів кодування (Code Style).

SwiftSyntax дає змогу парсити код, для того, аби проходитись по вихідному коду Swift і таким чином дозволяє аналізувати структуру кодової бази без потреби його виконання. В бібліотеці наявні API для проходження синтаксичного дерева, що дозволяє шукати конкретні патерни програми. Окрім цього, SwiftSyntax дозволяє не лише аналізувати, але й редагувати код, до якого він був застосований. Це також може бути зручним задля автоматизації форматування або автоматичного виправлення помилок у коді, з попередньо налаштованими правилами для цього у конфігураційному файлі.

Насправді ж, бібліотека є обгорткою навколо іншої бібліотеки, зі зручними для використання Swift зв'язками, – libSyntax. libSyntax відповідає за представлення вихідного файлу у вигляді абстрактного синтаксичного дерева.

Бібліотека SwiftSyntax надає зручні інтерфейси для аналізу та форматуванню коду шляхом парсингу: SyntaxVisitor, SyntaxFormatter, відповідно.

Parser – це інтерфейс, що перетворює Swift код застосунку у SwiftSyntax дерево. Його реалізація розділена між набором файлів названих за класом синтаксичних вузлів, які вони парсять [36]. Наприклад, розбір оголошень відбувається у Declaration.swift, а розбір виразів – в Expression.swift. Після парсингу користувач отримує готове синтаксичне дерево з головним вузлом – SourceFileSyntax.

Syntax – це інтерфейс, якому відповідають інші типи токенів, що складають з себе синтаксичне дерево. До синтаксів відносяться IntegerLiteralExprSyntax, ArrayLiteralExprSyntax тощо. Після цього для аналізу коду використовується SyntaxVisitor, який має метод visit(_ : Syntax). За допомогою цього методу, SyntaxVisitor проходиться по кожному з типів Syntax.

Відповідно до поведінки способу парсингу коду, стає зрозуміло, що SwiftSyntax використовує Visitor дизайн патерн.

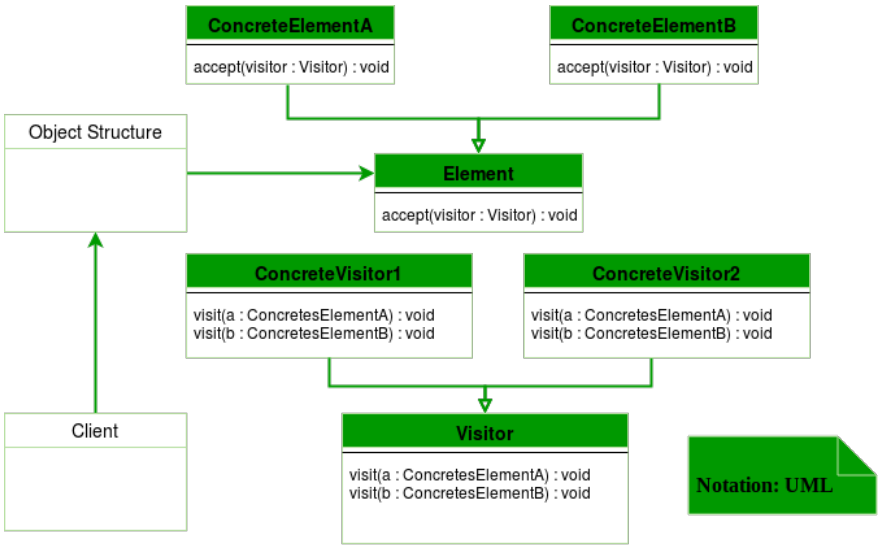

Рис. 1.2 Діаграма структури патерну Visitor

Це поведінковий дизайн патерн у програмуванні, що дозволяє розділити методи над об'єктами по яким він проходиться і самі об'єкти. В даному прикладі роль Visitor відіграє протокол SyntaxVisitor, який оголошує набір методів

visit(_ :), який визначає операцію над вузлом відповідного типу. Конкретними відвідувачами є класи, які відповідають протоколу SyntaxVisitor з подальшою імплементацією потрібних методів visit.

1.4. Abstract Syntax Tree

Абстрактне синтаксичне дерево (AST) – це структура даних, яка являє собою синтаксичне дерево вихідного коду програми, де кожен вузол відповідає елементу.

Воно утворюється шляхом проходження певними кроками представлення через лексичний аналізатор та синтаксичний аналіз. Розглянемо детальніше як компілятор бачить наш вихідний код.

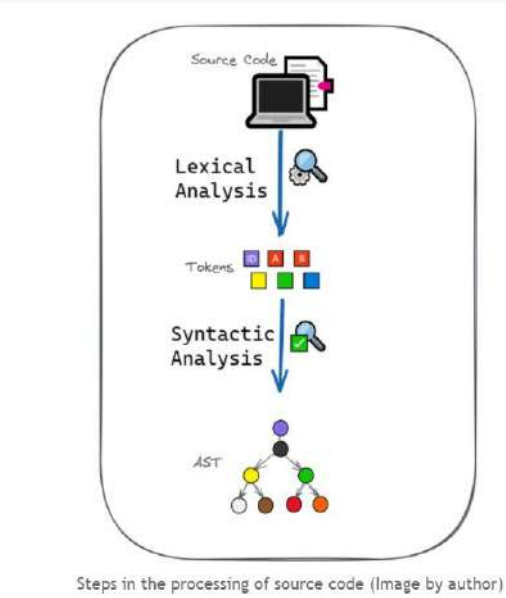

Рис. 1.3 Кроки побудови абстрактного синтаксичного дерева

Спочатку код передається у лексичний аналізатор або токенізатор. Там він поділяється на окремі символи задля утворення токенів – найменших одиниць вихідного коду, яка має значення в синтаксисі мови і є частиною вхідних даних для синтаксичного аналізатора, а в майбутньому – побудови абстрактного синтаксичного дерева. На цьому кроці ми можемо отримати наші літерали, оператори, ключові слова та дужки.

Далі токени передаються в парсер, де, знаючи граматику мови програмування, а саме який синтаксис мають функції, цикли тощо, аналізуються

наявні токени, що групуються у вузли, які представляють синтаксичні конструкції для подальшої побудови абстрактного синтаксичного дерева.

Для Swift зручним у використанні є Swift AST Explorer, який дає наочніше зрозуміти з чого складається написаний код на Swift і дає його представлення у вигляді абстрактного синтаксичного дерева.

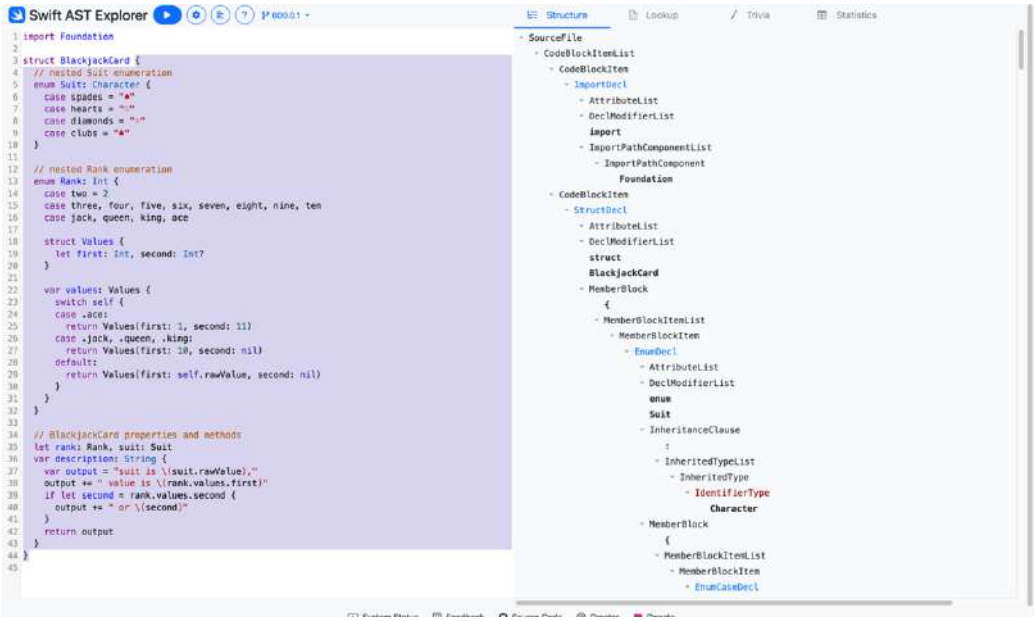

Рис. 1.4 Swift AST Explorer

Swift AST Explorer одразу дає представлення у вигляді синтаксичного дерева SwiftSyntax, що спрощує процес розробки аналізаторів з використанням найбільш актуальних версій SwiftSyntax.

1.5. Висновок до розділу 1

У цьому розділі було розглянуто теоретичні аспекти інструментів аналізу коду в контексті розробки мобільних застосунків. Проаналізовано два підходи до аналізу трафіку: статичний та динамічний.

Також було приділено увагу бібліотеці SwiftSyntax як основному інструменту для статичного аналізу коду для мови програмування Swift. Описано її роботу і можливість представлення програмного коду у вигляді абстрактного синтаксичного дерева, що дозволяє аналізувати, генерувати та редагувати код без його виконання. Разом з тим, коротко розглянуто патерн Visitor, який використовується в основі цієї бібліотеки для обходу та аналіз.

Розглянуто процес побудови AST, що дає розуміння внутрішнього представлення коду компілятором.

РОЗДІЛ 2. Антипатерни роботи з трафіком в мобільних застосунках на операційній системі iOS

2.1. Загальний опис

Впровадження антипатернів у коді є простим явищем, адже не кожен розробник володіє достатньою експертизою в певній галузі розробки застосунків. Варто враховувати можливість помилок під час налаштування трафіку є досить критичною для фінальної версії застосунку, адже користувач напряму буде бачити прояви цієї проблеми під час використання, тому це є важливим аспектом налаштування продуктивності.

Повільне завантаження та відклик на дії користувача спонукають відмовитись від застосунку на користь інших. З іншої сторони, додатки, що мають високий відклик та не мають затримки в обробці запитів, часто мають більшу залученість та вищу частку повернення користувачів у наступну сесію застосунку. Це є особливо критичним для додатків, що працюють з покупками, оскільки це напряму стосується генерування доходів та конверсії користувача.

Налаштування HTTP запитів, кешування та фонових задач не є примітивною задачею, тому дослідження цієї частини застосунку є пріоритетною. Ці антипатерни створюють чимало проблем, серед яких є надмірне споживання трафіку, зниження продуктивності додатка через споживання оперативної пам’яті мобільного пристрою, що призводить до швидкого виснаження батареї, а також погіршення користувацького досвіду.

Неоптимальна робота з мережею може проявлятись у тривалих завантаженнях контенту, нестабільній роботі застосунку при слабкому мобільному зв’язку, великим витратам мобільного трафіку, що є дорогим у країнах Євросоюзу та Сполучених Штатах Америки тощо [27].

Також проблема полягає в тому, що подібні проблемні місця важко знайти розробнику, адже тестування відбувається не на середньостатистичних пристроях, на яких потім буде використовуватися додаток. Як правило, всі тестування відбуваються в близьких до ідеальних умовах, через що неможливо належним чином виявити ці негаразди [34].

Окрім цього, варто зважати на те, що кожен запит несе певне навантаження на сервери. На серверах зберігається інформація для цього ж додатку або які взаємодіють з користувачем через них, тож підтримка їх стабільності важлива. Це може призводити до проблем не тільки на клієнтській стороні, але й на стороні сервера. Неправильно налаштоване кешування або надмірний полінг призводить до повторюваних запитів на сервер, які при різкому зростанні користувачів можуть ставати причиною відімкнення серверів.

Нижче наведено перелік проблем, вирішення яких дасть змогу уникнути розробнику подальших ускладнень з використанням трафіку.

2.2. HTTP заголовки

HTTP заголовки – це додаткові поля, що передаються у HTTP запитах. Вони можуть нести в собі додаткові умови про запит або метадані про передану у запиті інформацію. Популярними заголовками є заголовки авторизації, які використовуються для авторизації через токени, кешування, що надають інформацію клієнту щодо кешування відповіді сервера на клієнті, cookies та безпекові.

Всупереч їх значущості в оптимізації, розробники часто не надають їм достатню увагу. Це може бути через їх необов’язковість у взаємодії клієнту з сервером.

2.2.1. Заголовки кешування

Впровадження заголовків кешування не тільки мінімізує затримку, але й зменшує використання трафіку [9].

Основними заголовками є:

- **Cache-Control:** найбільш універсальний заголовок, який надає можливості встановлювати правила кешування і на стороні проксі-серверів, і на стороні клієнта. Він дозволяє встановлювати максимальний час зберігання кешу (max-age), необхідність ревалідації ресурсу (must-revalidate), а також повну заборону кешу (no-store)
- **Expires:** заголовок відповіді сервера, що вказує конкретну дату й час після якої кеш зникає

- **Content-Encoding:** також заголовок відповіді серверу, який компресує тип медіа. Заголовок надає розуміння клієнту, який тип кодування він буде підтримувати
- **Last-Modified:** вказує дату та час останнього редагування ресурсу. Його можна використовувати для подальшої перевірки чи була внесена якась зміна за допомогою If-Modified-Since

Правильне кешування може значно зменшити навантаження на сервер, тим більше, в моменти великого навантаження трафіком. Близько 80% навантаження на сервер може бути зменшено [9]. Це, своєю чергою, сприяє підвищенню доступності ресурсів незалежно від кількості трафіку та покращеного досвіду користувача. Бізнеси ж, зі своєї сторони, виграють через зменшення операційних витрат.

Згідно з результатами дослідження, додавання директив public та max-age у заголовку Cache-Control, дозволяє ресурсам ефективно бути закешованими та бути перевикористаними [9]. Затримка на одну секунду може зменшувати конверсію на 7% [9]. Це доводить значущість налаштування Cache-Control заголовку. Проблеми можуть виникнути у випадку, коли логіка кешування застосунку або її відсутність намагаються перезаписати та не зважати увагу на те, що повертає серверна частина [26]. Це може слугувати додатковим отриманням даних з серверу.

На рисунку нижче зображено процес як працює заголовок Cache-Control.

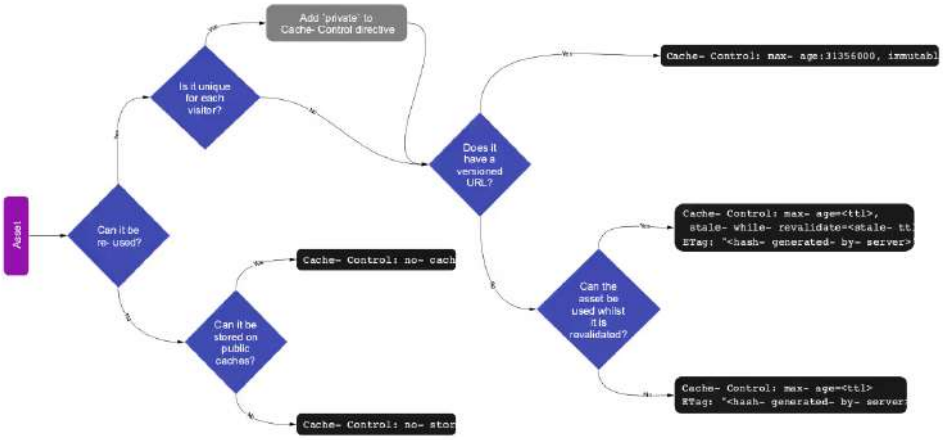

Рис. 2.1 Поведінка заголовку Cache-Control [26]

Заголовок Expires також відіграє значну роль для правильного використання закешованих даних з серверу. Шляхом визначення конкретного часу закінчення терміну дії даних, клієнтська сторона визначає чи користувач отримає свіжі ресурси, чи попередньо збережені дані. Конфігурація ресурсів, з можливістю кешування на довший період ніж стандартно, значно зменшує запити на сервер, покращуючи загальний відклик. На противагу цьому, використання застарілих ресурсів може призводити до відображення неактуальної інформації.

Згідно з даними з дослідження, ефективне налаштування заголовку Expires сприяє зменшенню навантаження на сервер до 30%.

Strategy	Effect on Load Time (%)	Server Requests Reduction (%)
Short Cache Duration	10	5
Medium Cache Duration	25	15
Long Cache Duration	40	30

Рис. 2.2 Вплив налаштування часу кешування на сервер [9]

Ще одним важливим заголовком є Content-Encoding. Одним з важливих способів зменшення затримки між запитами і відповіддю сервера є налаштування кодування даних. Дослідження доводять, що, імплементуючи заголовки Content-Encoding, дозволяє досягти зменшенню розміру переданих даних на 70%, покращуючи досвід користувача [9].

Популярні кодування як Gzip і Brotli можуть бути застосовані на велику кількість типів файлів.

- **Gzip:** є одним з найбільш поширених алгоритмів стиснення. В собі він використовує алгоритм DEFLATE, який комбінує алгоритм LZ77 та кодування Хаффмана [10]. Gzip забезпечує високий ступінь стиснення при прийнятних витратах ресурсів. Позначається в заголовку як Content-Encoding: gzip.

- **Brotli:** більш сучасний, розроблений Google. Також побудований на основі LZ77, але використовує попередньо визначені словники [11]. Позначається як Content-Encoding: br.

Такі методи не лише покращують швидкість отримання відповіді сервера, але й зменшують використання серверних ресурсів. Налаштування цього заголовку є важливим в таких застосунках як чатах, де впевненість у швидкому надсиланні та отриманні повідомлення є важливою для користувача. В дослідженнях йдеться, що залученість користувача збільшується на 50%, демонструючи наскільки важливим є фактор швидкого завантаження інформації для користувача [9].

2.2.2. Безпекові заголовки

Попри те, що для більш економного використання трафіку рекомендують якомога менше використовувати HTTP заголовків, нехтувати заголовками, що відповідають за безпекову частину застосунку є стратегічно неправильним рішенням.

Заголовки безпеки – це атрибути HTTP запиту, що є заголовками відповіді сервера, які використовуються для комунікації з клієнтом описуючи безпекову політику. При систематичному використанні заголовків безпеки зменшується ймовірність XSS та DDoS атак на сервери. Їх застосування запобігають несанкціонованому доступу та сприяють безпечній взаємодії в Інтернеті [12].

Основними безпековими заголовками є:

- **Content-Security-Policy:** цей заголовок, по суті, не дозволяє неавторизованим ресурсам доступ до серверної частини. Згідно з дослідженням, імплементація цього заголовка зменшує ризик XSS атаки на 90 відсотків [9].
- **X-Content-Type-Options:** виступає визначником чи заголовки типу MIME у Content-Type заголовках можуть бути змінені на сервері. Встановлення значення nosniff для цього заголовку є критичним, оскільки це не дозволяє браузерам інтерпретувати файли як MIME-тип,

відмінний від оголошеного, таким чином блокуючи потенційно шкідливе виконання скриптів [12].

- **X-Frame-Options:** Ще одним важливим заголовком безпеки є заголовок X-Frame-Options, який захищає від атак клікджекінг, що можуть скомпрометувати дані користувача [13]. Клікджекінг – це тип атаки, який спрямовує юзера натискання невидимих або елементів, під якими є інші елементи, що можуть перевести на іншу сторінку або почати завантажування якогось шкідливого програмного забезпечення. Однак, неправильне налаштування цього заголовка може наражати користувачів на ризики, дозволяючи несанкціоновані дії через вбудовані iframe теги HTML [12].
- **Strict-Transport-Security:** HSTS має вирішальне значення для запобігання атакам типу man-in-the-middle і забезпечення цілісності та конфіденційності даних, що передаються між сервером і клієнтом [12]. Низький рівень його впровадження вказує на те, що багато вебсайтів все ще покладаються на незахищені HTTP-з'єднання або не налаштовані на використання HTTPS.
- **X-XSS-Protection:**

  Відповідно до дослідження, відсоток сайтів, що використовують цей заголовок є малим – 22 відсотки [12]. Це свідчить про недостатнє використання навіть базового захисту від XSS атак. Це додатковий шар захисту від XSS атак, який підтримують сучасні браузери [9].

Таким чином правильне використання HTTP заголовків, що відповідають за безпекову частину застосунку є виправданим для забезпечення безпеки клієнтської частини та захисту користувачів від кіберзагроз. Тому дотримування правила мінімізування додаткових заголовків не є обґрунтованим в даному контексті.

2.2.3. Налаштування cookies

У мові програмування Swift, налаштування використання cookies відбувається через URLSession.

URLSession – це клас, який надає потрібні API для вивантаження або завантаження даних з кінцевих точок (ендпоінтів) у вигляді інтернет-покликань. Також URLSession виконує функцію API для фонових задач, коли застосунок вимкнений або знаходиться у фоновому стані [14].

URLSession має можливість до конфігурації, Для цього використовується URLSessionConfiguration. Цей об’єкт визначає поведінку та принципів, які слід використовувати при вивантаженні або завантаженні даних через URLSession [15].

Задля налаштування cookie в URLSessionConfiguration використовуються параметри: httpCookieAcceptPolicy та httpShouldSetCookies.

httpCookieAcceptPolicy – директива, що відповідає за події, коли cookie будуть прийматись на клієнтську сторону застосунку [16].

Цей параметр має три значення типу HTTPCookie.AcceptPolicy:

- **always:** приймання всіх cookie
- **never:** відхилення від усіх cookies
- **onlyFromMainDocumentDomain:** приймання cookies тільки з основного доменного документа

Це є критичним для правильного менеджменту сесій авторизації або в будь-якому іншому випадку, коли cookies є важливими для правильної роботи застосунку.

Використання значення .always відповідає за те, що всі cookies будуть прийняті з серверу, що може впливати на використання трафіку.

Натомість httpShouldSetCookies відповідають за автоматичне додавання cookies до заголовків HTTP запиту. Це передбачає автоматичне відправлення cookies, що керуються HTTPCookieStorage, спільного сховища зберігання cookies в поточній сесії. Існує варіант використання або цього параметру зі значенням, або передачу в заголовку Cookie через httpAdditionalHeaders, а також шляхом використання об’єкту NSURLRequest з передачею при кожному запиті.

2.2.4. httpAdditionalHeaders

Використання httpAdditionalHeaders теж впливає на використання трафіку застосунком. Додаткові заголовки у Swift також містять повний список заголовків, що використовуються й у веброзробці.

httpAdditionalHeaders – це додаткові заголовки HTTP запиту, що користувач може додавати до кожного запиту для передачі додаткових параметрів або перезаписуванню наявних. У Swift, об’єкт URLSession вже містить певні, попередньо визначені, заголовки, які рекомендуються до використання компанією Apple.

У параметрі httpAdditionalHeaders Заголовки, пов’язані з авторизацією, автентифікацією, Connection та Host не рекомендується перезаписувати. Наприклад, для заголовка Content-Length, довжина даних у тілі запиту визначається автоматично, якщо контент переданий через об’єкт NSData [18]. Варто використовувати мінімальну кількість заголовків, що вимагає API.

2.2.5. Кешування даних

Як було розглянуто у попередньому розділі, кешування може грати велику роль в економії інтернет-трафіку. Swift також надає можливість налаштування кешування даних зі сторони клієнту. Це проявляється у повному налаштуванні URLCache та cache policy.

2.2.6. URLCache

URLCache – механізм налаштування кешування даних, який був представлений у бібліотеці Apple Foundation. Він кешує дані з сервера та видає їх користувачу, коли він цього потребує без потреби надсилання додаткових запитів [19].

Для налаштування кешування використовується об’єкт URLCache, в якому користувач задає параметри розміру пам’яті та диску. Після цього потрібно присвоїти новостворений об’єкт до URLCache.shared. Як правило, ці дії відбуваються при запуску застосунку в AppDelegate.

```swift
func application(_ application: UIApplication,
                 didFinishLaunchingWithOptions launchOptions: [UIApplication.LaunchOptionsKey: Any]?
                 ) -> Bool {
  // Set up URLCache
  let memoryCapacity = 20 * 1024 * 1024
  let diskCapacity = 100 * 1024 * 1024
  let cache = URLCache(memoryCapacity: memoryCapacity, diskCapacity: diskCapacity, diskPath: "myCache")
  URLCache.shared = cache

  return true
}
```

Рис. 2.3 Приклад налаштування URLCache у Swift

Далі, використовуючи метод storeCachedResponse, користувач має можливість зберегти відповідь сервера, що прийшла.

```swift
let cachedResponse = CachedURLResponse(response: response, data: data)
URLCache.shared.storeCachedResponse(cachedResponse, for: request)
```

Рис. 2.4 Приклад кешування відповіді сервера

Важливим фактором є визначення розміру кешу. Ресурси, розмір яких перевищує 10 відсотків від загальної місткості кешу, не підлягають кешуванню [20]. Таким чином, при роботі з великими файлами, слід ретельно ставитись до вибору розміру кешу. Треба звертати на це особливу увагу з огляду на відсутність конкретних чисел для правильного кешування в документації Apple.

2.2.7. requestCachePolicy

Нещодавно, Apple запропонувала зручні директиви для контролю правил кешування. Це прийшло на заміну створення власних NSCache [21]. В URLSessionConfiguration є заздалегідь визначені значення для requestCachePolicy.

До них входять такі директиви:

- **useProtocolCachePolicy:** вона працює таким чином, що коли закешованої відповіді локально не знайдено, надсилається ще один запит для отримання даних. В іншому випадку, якщо закешовані дані не мають ревалідації щоразу при отриманні і якщо їх актуальний термін не закінчився, то система повертає закешовану відповідь сервера. Якщо ж закешована відповідь застаріла або потребує ревалідації, то надсилається

запит на перевірку чи є оновлені дані після закешованих. У випадку, якщо ці дані в тому ж стані, повертаються закешовані дані. Якщо ж ні, то система отримає дані з початкового джерела. Відповідно до документації, Apple наголошує на використанні директиви reloadIgnoringLocalCacheData, якщо запити відбуваються у байтовому діапазоні [22].

Приклад роботи цього методу наведено нижче:

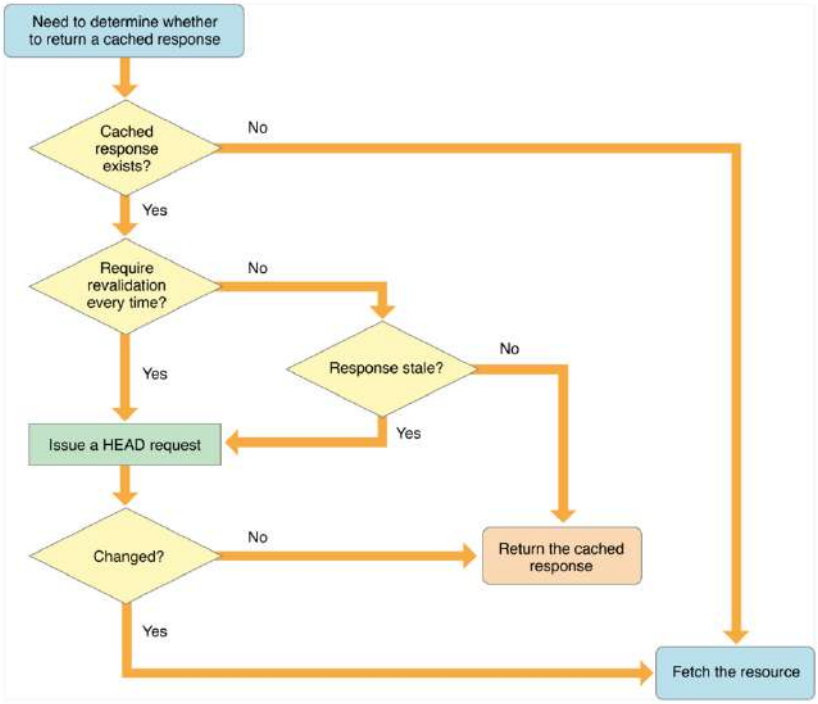

Рис. 2.5 Приклад роботи директиви useProtocolCachePolicy [22]

У випадку використання цієї директиви є 5 варіантів поведінки.

1. **Відсутність Cache-Control та вимкнений ETag**: Застосунок не взаємодіє з сервером, а URLSession ігнорує оновлення. Цей випадок є нетиповим, оскільки ETag зазвичай застосовується для синхронізації даних.
2. **Відсутність Cache-Control, ETag змінюється кожні 2 секунди**: URLSession регулярно оновлює відповідь сервера відповідно до змін ETag.
3. **Cache-Control встановлено на 4 секунди, ETag змінюється при зміні ресурсу:** Застосунок звертається до сервера кожні 3 секунди,

але URLSession щосекунди повертає успішну відповідь, оновлюючи ресурс.

4. **Cache-Control встановлено на 0 секунд, ETag відсутній**: Застосунок звертається до сервера при кожному запиті, але URLSession завжди повертає той самий ресурс.
5. **Cache-Control встановлено на 0 секунд (або не встановлено), ETag змінюється при кожному запиті**: Як і у випадку з reloadIgnoringLocalCacheData, застосунок отримує найактуальнішу версію ресурсу з сервера при кожному запиті.

- **returnCacheDataElseLoad:** ця директива спирається лише на закешовані дані. Тільки коли застосунок не матиме можливість знайти кеш на диску, то він надсилатиме запит на сервер для їх отримання [23]. В цьому випадку, незалежно від того чи буде в нас встановлений Etag та Cache-Control, клієнт буде отримувати закешовані дані попередніх запитів до моменту очищення локального кешу.

```
URLCache.shared.removeAllCachedResponses()
```

Рис. 2.62 Приклад очищення локального кешу

- **reloadIgnoringLocalCacheData:** саме ця директива є найбільш впливовою на використання трафіку, адже вона повністю ігнорує закешовані дані. Попри значення Cache-Policy та Etag, кожен запит щосекунди буде надходити на сервер і найбільш актуальні дані будуть повернуті [21].

- **returnCacheDataDontLoad:** директива дуже схожа на поведінку застосунку в офлайн режимі. Він завжди повертатиме закешовані дані незважаючи на Cache-Control або Etag і відповідні параметри цих директив, що вказуються термін дії кешу. Якщо ж закешованих даних не буде знайдено, то запит провалиться.

2.3. Тайм-аути та повторення запитів

Тайм-аути допомагають налаштуванню правильної поведінки застосунку при помилках, з якими може стикнутись клієнт при надсиланні запитів на сервер. Правильно налаштований тайм-аут сприяє очікуваній та обдуманій поведінці сервера щодо надсилання повторних запитів або часу на отримання інформації через великий обсяг даних. Повторні запити збільшують використання мобільного трафіку та підвищують навантаження на сервер. Це також може сприяти некоректній роботі застосунку пов'язаній з нестабільною інтернет-мережею.

З іншої сторони, занадто великі тайм-аути також можуть сприяти неправильній роботі додатку, тож за розробником стоїть задача визначення найкращого значення тайм-аутів конкретно для їх сфери роботи застосунку.

Тайм-аути, аналогічно до типу кешування, налаштовується в URLSessionConfiguration. Існує дві основні директиви: виставлення часу очікування даних та загальний час тайм-ауту на ресурс відповідного запиту.

Частою помилкою розробників є попередня перевірка доступності ресурсу, а не виставлення очікування на з'єднання. Модуль Apple, який працює з інтернет-запитами легко підлаштовується під модель "Waits for Connectivity": якщо поточний запит «губиться», то застосунок чекатиме і продовжить завантаження даних з сервера, коли з'єднання відновиться [25].

2.3.1. timeoutIntervalForRequest

Директива, що відповідає за визначення тайм-ауту на очікування відповіді на конкретний запит [24]. Використовується для стандартних запитів та визначає час, скільки в секундах задача повинна очікувати додаткових даних. Важливо зазначити, що при кожному отриманні даних, таймер, що відповідає за цей тайм-аут буде оновлено [25].

Присвоєння низьких значень для цієї директиви є нераціональним, оскільки кожного разу запит буде провалюватись. Після цього буде надіслано новий, ідентичний, запит. Це буде спричиняти більше використання трафіку на однакові запити і у випадку використання нестабільного з'єднання буде

відбуватись часто. Такі часті запити можуть імітувати повторну активацію LTE, коли той знаходиться в стані очікування.

Оптимальним значенням, відповідно до документації Apple, є одна хвилина [24]. За замовчуванням, ця директива має значення саме одну хвилину. Тож встановлення низьких або занадто великих значень є впливовим на використання трафіку. Окрім того, це може призводити до того, що застосунок перестане відповідати та споживатиме системні ресурси без можливості відновлення. Якісна логіка надсилання повторних запитів має мінімізувати рукостискання між клієнтською та серверною частиною додатку.

2.3.2. timeoutIntervalForResource

Правило, що відповідає за очікування завершення передачі даних у відповіді запиту до моменту скасування запиту. Цей тайм-аут призначений для довших задач: вивантаження або завантаження даних, які проводяться у фоні. Аналогічно до timeoutIntervalForResource, якщо запит досягає цього тайм-ауту, то він провалюється. Значення за замовчуванням, яке рекомендує Apple, сягає сімох днів.

```
let configuration = URLSessionConfiguration.default
configuration.waitsForConnectivity = true
configuration.timeoutIntervalForRequest = 60 // 1 minute
configuration.timeoutIntervalForResource = 60 * 60 // 1 hour
```

Рис. 2.7 Приклад налаштування timeoutIntervalForRequest та timeoutIntervalForResource

2.4. Керування з'єднанням

Налаштування з'єднання є важливим для фонових задач. Swift пропонує декілька параметрів, що можуть впливати на використання трафіку фоновими задачами та, таким чином, покращувати роботу додатка.

2.4.1. allowsCellularAccess

allowsCellularAccess – це директива, що відповідає за надання доступу до використання мобільного трафіку.

В більшості країн Євросоюзу та Сполучених Штатах Америки мобільний трафік є набагато дорожчим ніж в Україні [27]. Тож використання мобільного трафіку не є пріоритетним. Тим більше, якщо це стосується фонових завантажень, що можуть завантажувати великі обсяги даних без відомості користувача.

За замовчуванням, це правило має значення true. Тож цей параметр має важливе значення для розробників, бо якщо його значення не перемкнути на false, то програми будуть використовувати мобільний трафік в момент, коли Wi-Fi з'єднання буде недоступне.

Окрім того, відповідно до досліджень, встановлення значення false для фонових задач запобігає запуску завантаження або вивантаження даних у мобільній мережі [28]. Якщо ж завантаження було ініційоване через Wi-Fi, то фонова задача продовжується і після втрати з'єднання. Таким чином, директива allowCellularAccess не є повною забороною на використання мобільного трафіку. В цьому випадку необхідне використання директиви isDiscretionary.

Варто зазначити, що налаштування цієї директиви має відбуватись до створення об'єкта URLSession, щоб це працювало як очікується.

2.4.2. networkServiceType

networkServiceType – це правило URLSessionConfiguration, що слугує визначенням типу мережевого сервісу, пов'язаного з запитами на основі цієї конфігурації. Це допомагає виставити правильні пріоритети між запитами в межах одного URLSession. Може бути застосовано для фонових та першочергових задач.

Використання відповідного networkServiceType може дозволити використовувати Cellular Network Slicing для застосунків, що його підтримують. Cellular Networking Slicing – це концепція створення множинних віртуальних мереж на базі одної фізичної інфраструктури. Кожен такий слайс може бути незалежно налаштований та оптимізований для конкретних вимог окремих сервісів.

networkServiceType має такі значення, визначені Apple:

- **default:** визначає стандартну поведінку для більшості задач з використанням трафіку
- **background:** вказує на використання трафіку для фонових задач, що допомагає покращити ефективність та зменшити затримку передачі даних залежно від умов мережі та стану пристрою.
- **voice:** визначає трафік, що використовується для інтерактивної комунікації голосом, вимагаючи малої затримки у передачі даних.
- **video:** трафік, що використовується під відеодані, який потребує вищої пропускної здатності.
- **responsiveData:** використовується для даних, що є чутливою до затримки.
- **avStreaming:** призначена для аудіо та відео стрімінгу.
- **callSignaling:** для викликів та управління їх трафіком.

Загалом ініціалізація конфігурації для фонових задач з networkServiceType типу .video може слугувати неправильним пріоритетам на роботу з трафіком.

```
let sessionConfig = URLSessionConfiguration.background(withIdentifier: "com.example.backgroundupload")
sessionConfig.networkServiceType =.video
let session = URLSession(configuration: sessionConfig, delegate: self, delegateQueue: nil)
```

Рис. 2.8 Налаштування networkServiceType для фонових задач

Аналогічним чином, виставлення неправильного значення, наприклад, .responsiveData для основних задач може повпливати на пріоритет їх виконання. Тим самим, сповільнюючи роботу застосунку.

```
let configuration = URLSessionConfiguration.default
configuration.networkServiceType =.responsiveData
let client = HTTPClient(baseURL: "...", configuration: configuration)
```

Рис. 2.9 Неправильне налаштування networkServiceType для пріоритетних задач

2.4.3. waitsForConnectivity

waitsForConnectivity – це правило, що було представлено Apple в iOS 11.0. Воно описує як себе має поводити застосунок при нестабільному інтернет-з'єднанню чи коли воно повністю пропадає.

Якщо у цього параметру стоїть значення true, то під час очікування, сесії надсилає сповіщення делегату через метод urlSession(_ : taskIsWaitingForConnectivity:). Якщо ж правило має значення false, то запит провалиться з помилкою.

Для пріоритетних задач ця директива може значно покращити досвід користувача у моменти, коли інтернет-з'єднання може бути нестабільним. Це виключатиме можливість крешу застосунку через неможливість завантажити потрібні дані протягом сесії роботи. Для фонових задач це не буде впливати на процес завантаження, оскільки для них ця директива ігнорується.

Схожим методом, як і timeoutIntervalForRequest, використання цієї директиви замінює попередню перевірку клієнтом доступу до сервера на можливість застосунку працювати з очікуванням на повернення з'єднання.

2.5. Доставлення вмісту

Задля кращого доставлення даних з сервера та забезпечення ефективного використання ресурсів, зокрема в умовах нестабільного та обмеженого трафіку, важливе значення мають директиви httpMaximumConnectionPerHost та allowsExpensiveNetworkAccess.

2.5.1. httpMaximumConnectionsPerHost

httpMaximumConnectionsPerHost – це правило, що вказує кількість одночасних з'єднань з одним хостом. Цей ліміт виставляється для конкретної сесії. Також залежно від якості інтернету, сесія може використовувати і нижчий ліміт від вказаного [29].

Рекомендованими значеннями є 4-6. Стандартним значенням для мобільного трафіку є 6 одиниць, а для Wi-Fi – 4. Також рекомендується використовувати мультиплексування HTTP/2, що дозволяє паралельно передавати дані через одне з'єднання та тим самим підвищувати продуктивність.

2.5.2. allowsExpensiveNetworkAccess

allowsExpensiveNetworkAccess – це директива, що відповідає за дозвіл на використання дорогого трафіку, який класифікується системою iOS. Рішення приймається на основі природу цього мережевого інтерфейсу та інших факторах.

iOS 13 вважає увесь мобільний трафік і персональні точки доступу дорогим трафіком [30]. Якщо недорогий трафік не є доступним і директива має значення false, то будь-яка задача створена з цієї сесії провалиться.

Кращим рішенням обмеження дорогого трафіку є використання директиви allowsConstrainedNetworkAccess. Це дозволить користувачам більш автономно визначати якість даних та трафік, який буде використано. Наприклад, при ввімкненому Low Data Mode, користувачу буде запропоновано завантажити дані гіршої якості, якщо в цьому буде потреба. Поширеною практикою є адаптація використання трафіку саме для Low Data Mode, що дозволяє мінімізувати негативний вплив на користувацький досвід, адже не існуватиме категоричних обмежень, що не матимуть альтернативних підходів до взаємодії користувача [31].

2.6. Префетчинг та фонові дані

Правильне налаштування фонових задач теж є важливим фактором налаштування трафіку. Неправильне виконання з каскадністю може слугувати причиною завеликого використання інтернету через необізнаність про це користувача.

2.6.1. sessionSendsLaunchEvents

sessionSendsLaunchEvents – це параметр, що відповідає за поведінку застосунку при закінченні виконання фонових задач.

Одним з сценаріїв неправильного використання може бути каскадне виконання фонових задач. Якщо завершення одного фонового завдання викликає фоновий запуск застосунку, а код, що відповідає за запуск додатка може ініціювати інший мережевий запит, то може утворювати каскад фонових задач без явної взаємодії з користувачем. Це може призводити до значного використання трафіку і розрядженню акумулятора. Ще одним прикладом можуть слугувати часті оновлення, ініціюючи кожного разу окрему мережеву сесію.

Гарною практикою є перевірка налаштувань Low Data Mode перед запуском великих, за споживанням ресурсів, операцій у фоновому режимі.

2.6.2. multipathServiceType

multipathServiceType – це директива, що відповідає за правила з'єднання Multipath TCP для передачі даних через Wi-Fi чи мобільний трафік.

На відміну від Linux, macOS та iOS використовують MPTCP API. У цих операційних системах менеджер шляхів не обробляється ядром, але це може бути задачею програми.

Існує 3 варіанти використання цієї директиви:

- ***handover:*** це тип сервісу, що дозволяє максимально непомітно передавати виконання запитів між Wi-Fi та мобільним трафіком для уникнення розриву з'єднання.
- ***interactive:*** сервіс, при якому MTCP намагається використовувати інтерфейс з найменшою затримкою.
- ***aggregate:*** цей тип об'єднує інші варіанти MTCP з метою збільшення пропускної здатності та мінімізації затримок. Він є найбільш універсальним.

2.7. Полінг

В контексті iOS застосунку, полінг це періодичне надсилання запитів, використовуючи HTTP або інші інтернет-протоколи, щоб зрозуміти чи доступні новіші дані за наявні і отримати їх в цьому випадку.

Частий полінг має недоліки, що може суттєво вплинути на продуктивність та споживання ресурсів застосунків. Основним є марне використання трафіку, що в контексті мобільного трафіку є критичним. Підтримка постійного мережевого з'єднання шляхом надсилання постійних запитів також збільшує енергоспоживання пристрою. З боку ж серверу, велика кількість клієнтів створює навантаження на серверну інфраструктуру.

2.7.1. Полінг з використанням таймера

Одним з видів полінгу в Swift є полінг з використанням об'єкта Timer.

```
Timer.scheduledTimer(withTimeInterval: 5.0, repeats: true) { timer in
    // виклик HTTP запиту
}
```

Рис. 2.10 Приклад полінгу з використанням таймеру

Будь-який полінг є неефективним щодо використання ресурсів, але полінг через таймер може нести ще більше проблем. Відповідно до рекомендацій Apple щодо використання полінгу, потрібно передавати лише потрібну кількість інформації для ефективного використання батареї [32].

Тож на заміну таймерам пропонується використання Apple Push Notification Service (APNS). Це допомагає уникати потреби у повторювальних запитах. Натомість пуш-повідомлення будуть автоматично приходити, коли застосунок має оновити дані з сервера. Також полінг може працювати лише коли застосунок активний, що унеможливлює оновлення даних у фоновому режимі без використання фонових задач або APNS.

Розглядаючи вплив таймерів на загальну продуктивність системи, слід зазначити, що виведення систему зі стану очікування вимагає витрат енергії, коли процесор та інші компоненти виводяться зі стану низького енергоспоживання. Якщо таймер змушує систему виходити з такого стану, він несе за собою ці витрати.

2.7.2. Рекурсивний полінг

Іншим варіантом полінгу є рекурсія. В цьому випадку використовується DispatchQueue.main.asyncAfter з певною затримкою.

```
func startPolling() {
    fetchResource { success in
        DispatchQueue.main.asyncAfter(deadline: .now() + 5.0) {
            self.startPolling()
        }
    }
}
```

Рис. 2.11 Приклад полінгу через рекурсію

Виконання важких операцій на головному потоці застосунку можуть негативно впливати на його швидкість реакції на дії користувача. Помітна тенденція до збільшення часу виконання кожної наступної ітерації задачі, де, наприклад четверта ітерація займає вдвічі більше часу ніж перша [33].

Багаторазова диспетчеризація завдань на головну чергу може призвести до її перенавантаження викликаючи помітні затримки інтерфейсу користувача. Крім того, рекурсія спричиняє постійне зростання стеку викликів, що при тривалому використанні призводить до вичерпання оперативної пам’яті. Різниця мережевої затримки також вносить непередбачуваність в процес полінгу, роблячи поведінку системи менш стабільною.

2.7.3. Полінг за допомогою нескінченних циклів з затримкою

Застосування нескінченних циклів у програмуванні є антипатерном, що може призвести до непередбачуваної поведінки системи та неефективного використання ресурсів. Використання цих циклів для полінгу не є меншою проблемою.

```
while shouldContinuePolling {
    fetchResource()
    Thread.sleep(forTimeInterval: 5.0)
}
```

Рис. 2.12 Приклад полінгу з нескінченним циклом

Використання цього принципу спричиняє блокування потоку командою Thread.sleep, що повністю зупиняє виконання цього потоку. У випадку виконання на головному потоці, це призведе до повного зависання інтерфейсу користувача.

Безперервно робочий потік також перешкоджає ефективному керуванню енергоспоживання iOS навіть під час режиму Low Data Mode. Такі цикли можуть активувати системний механізм контролю (watchdog), що призведе до примусового завершення роботи застосунку. Крім того, мережеві помилки в цьому прикладі можуть призвести до зупинки полінгу.

2.8. Висновок до розділу 2

У цьому розділі було детально розглянуто низку антипатернів, пов’язаних з роботою з трафіком у мобільних застосунках на оперативній системі iOS. Усунення виявлених антипатернів та впровадження рекомендованих практик дозволить розробникам значно покращити продуктивність застосунків, оптимізувати використання трафіку та забезпечити кращий користувацький досвід.

РОЗДІЛ 3. Реалізація аналізатора для виявлення антипатернів роботи з трафіком

3.1. Загальний опис

Аналізатор антипатернів роботи з трафіком розроблено для виявлення проблем у Swift коді, які можуть негативно впливати на використання трафіку мобільних додатків.

Основна мета – допомогти розробникам оптимізувати операції з використанням трафіку, зменшити тим самим чином споживання трафіку та покращити продуктивність застосунків.

3.2. Використані методи

Аналізатор побудовано на базі macOS Command Line Tool проєкту. Він містить дві залежності: SwiftSyntax та ArgumentParser. Ці інструменти дозволяють зручним чином парсити користувацький код за допомогою власних імплементацій класів Visitor.

Спочатку було створено додатковий протокол, що буде об’єднувати всі класи Visitor для легшої взаємодії з масивами Visitors.

```
protocol Visitable: SyntaxVisitor {
    var properties: [String] { get set }
    var warnings: [AntipatternWarning] { get set }
}
```

Лістинг. 3.1 Код протоколу Visitable

Він містить параметр properties, який відповідає за визначення антипатернів, що має шукати відвідувач в коді та warnings – масив попереджень, які було виявлено протягом аналізу коду.

На кожен тип антипатерну, розглянутого в роботі, було створено окремий клас-сервіс, що містить усі класи-відвідувачі. Всередині масиву visitors ми прописуємо усіх відвідувачів, що відносяться до типу конкретного класу-сервісу. Далі відвідування відбувається за допомогою методу .walk і

фільтрується лише те, що може впливати на використання трафіку, відповідно до описаного дослідження.

Кожен клас-сервіс також має свій протокол для зручного узагальнення в коді. Було створено протокол VisitableService для перевикористання методу analyzeSyntaxTree аби спростити процес виклику методів аналізу у кожного відвідувача класу-сервісу.

```
protocol VisitableService {
    var visitors: [Visitable] { get set }
    func analyzeSyntaxTree(_ tree: SourceFileSyntax) -> [AntipatternWarning]
}

extension VisitableService {
    func analyzeSyntaxTree(_ tree: SourceFileSyntax) -> [AntipatternWarning] {
        visitors.flatMap { visitor in
            visitor.walk(tree)
            return visitor.warnings
        }
    }
}
```

Лістинг. 3.2 Приклад коду протоколу сервісу

Додатково було визначено клас AntipatternWarning, що зберігає інформацію про кожен антипатерн. Він зберігає інформацію про шлях до файлу, рядок та колонку про знайдений антипатерн, значення змінної для цього антипатерну, якщо значення наявне та рекомендації, що виводяться користувачу під кожен антипатерн. Саме такі поля повинна мати структура, щоб бути правильно представленою у вигляді попереджень у середовищі розробки.

Основний файл, в свою чергу, містить всіх відвідувачів-сервісів, що відповідають за свою частину антипатернів використання мережевого трафіку. За потреби виклику аналізатору з терміналу, користувач має змогу вказати вхідний та вихідний файл через параметри -i та -o. При цьому використовується бібліотека ArgumentParser. Далі вхідний файл декодується та парситься до вигляду синтаксичного дерева за допомогою парсера бібліотеки SwiftSyntax. Після цього синтаксичне дерево вихідного файлу передається в кожен відвідувач та аналізується по окремості. Залишається лише вивести попередження у консоль при запуску з терміналу або підсвітити його у середовищі розробки.

Наступним кроком було імпортування аналізатора в реальні застосунки. Був вибір між Build Phase та Swift Plugin. Все ж таки, було вирішено зупинитись на новішому Swift Plugin завдяки простоті роботі з ним.

Для того, щоб мати можливість використовувати Swift Package у формі плагіну, в Package.swift файлі, потрібно додати декілька рядків коду, що будуть давати можливість використовувати плагін з поточного проекту аналізатора. Задача полягає в створенні окремого продукту, що буде відповідати саме за плагін нашого Swift Package. Саме .executable було використано для можливості запуску аналізатору з терміналу, а для інтеграції аналізатору в наявний проект використовується .plugin. В ньому ми вказуємо назву плагіну та таргет, який він використовує у вигляді залежності.

Для імплементації самого методу використання плагіну аналогічним чином модифікується файл Package.swift, де вказується capability плагіну та його залежності. В такому випадку, плагін залежить від основного таргету, що проводить аналіз коду. Capability відповідає за тип плагіну: є можливість обрати чи це буде command або buildTool.

```
targets: [
    .executableTarget(
        name: "network-analyzer",
        dependencies: [
            .product(name: "SwiftSyntax", package: "swift-syntax"),
            .product(name: "SwiftParser", package: "swift-syntax"),
            .product(name: "ArgumentParser", package: "swift-argument-parser")
        ]
    ),
    .plugin(
        name: "NetworkAnalyzerPlugin",
        capability: .buildTool(),
        dependencies: [
            .target(name: "network-analyzer")
        ]
    )
]
```

Лістинг. 3.3 Налаштування деталей плагіну

Залишається лише додати плагін в наявний проєкт через Add Dependency, як і будь-яку іншу залежність, вказавши посилання на репозиторій, де зберігається потрібний інструмент. Далі треба вказати вимогу у виконанні в налаштування проекту, Build Phases та вказати потрібний плагін, з тих, що був підключений.

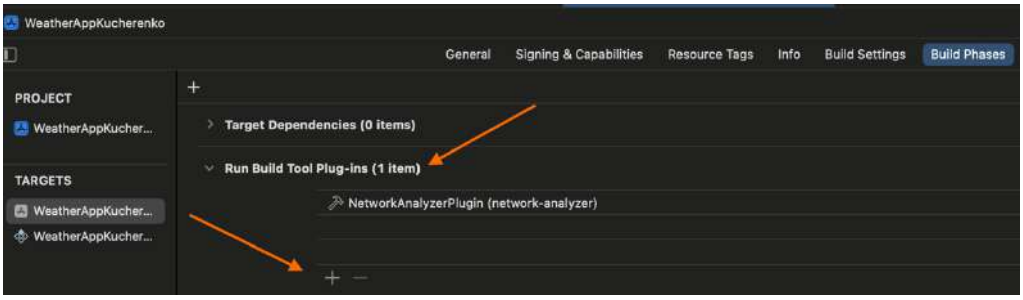

Рис. 3.1 Спосіб додавання плагіну в проект

Після цього, при кожній збірці проекту, плагін буде виконуватись та видавати попередження для користувача, де йому варто звернути увагу на написаний код.

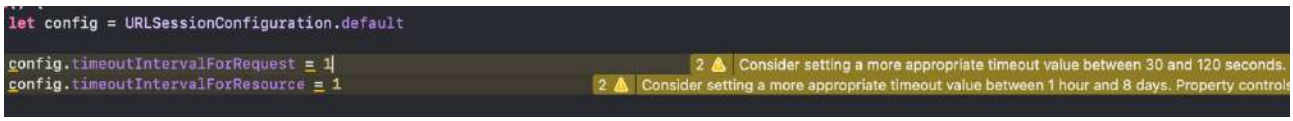

Рис. 3.2 Приклад роботи плагіну в сторонньому проекті

Після виправлення антипатернів в коді, при повторній збірці проекту, рекомендації щодо коду користувача пропадуть.

3.3. Build Phases vs Swift Plugin

У Xcode існує два варіанти використовувати додаткові залежності для їх запуску під час збірки проєкту: Swift Build Phases та Swift Plugins.

Swift Build Phases це дещо застарілий метод запуску скриптів, що містять набір інструкцій, під час збірки проєкту. Вони можуть виконувати будь-які задачі.

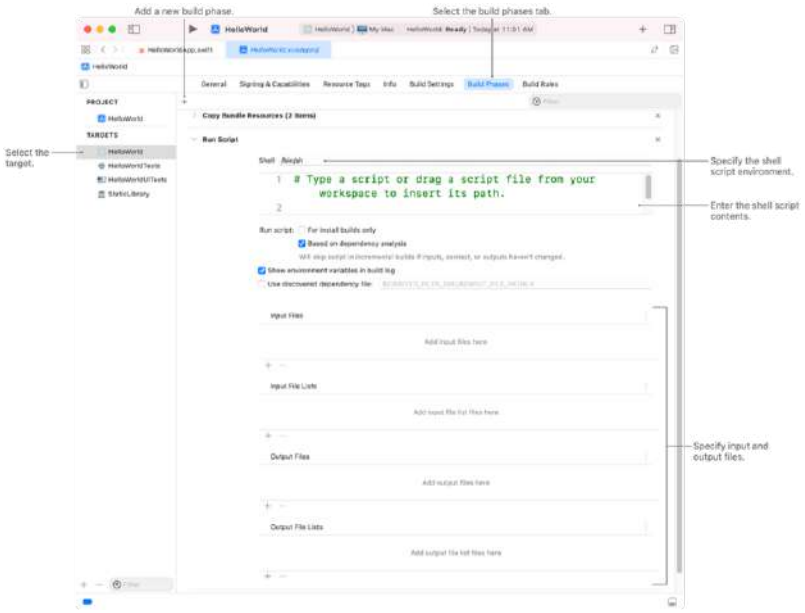

Рис. 3.3 Приклад створення Build Script

Труднощі можуть виникати під час створення таких скриптів, адже при масштабуванні проекту, їх кількість зростає, а з цим зростає і складність. Також існує проблема, що Build Scripts не можуть виконуватись паралельно. Через це при великій кількості таких скриптів зростатиме час збірки проекту, що є досить важливою метрикою для великих проектів.

З варіантів розв'язання такої проблеми є такі: винесення подібних скриптів на CI або технічна зміна в проекті. З простих рішень виникає паралізація виконання скриптів збірки проекту. Паралелізація bash скриптів не є легкою задачею, тому від цього плавно відмовляються. Якщо ж проект є багатомодульним, то гарним варіантом є виклик build script на кожен модуль проекту, що теж покращить показники збірки проекту, але не зможе багаторазово зменшити час збірки.

Ще одним рішенням є використання Swift Plugins. Swift Plugin – це Swift скрипт, який може запускатись до збірки проекту або мануально. У 2022-му році Apple представила Swift Package Plugins. Swift Plugin є нативним рішенням від Apple та використовує API від Swift Package Manager. Це рішення працює в Xcode, терміналі або Swift Package.

Swift Plugins поділяються на два типи:

- **Command Plugin:** це плагіни, що можна запускати вручну. Підтримується запуск через термінал або в інтерфейсі Xcode.

Рис. 3.4 Приклад виклику Command Plugin

- **Build tool Plugin:** плагін, що виконується при збірці проєкту

Перевагою Swift Plugins є легке поширення між іншими розробниками і зручна інтеграція в проєкт. Достатньо просто поділитися своїм кодом на GitHub, а потім додати залежність до цього репозиторію та додати цей плагін в Build Tool Plugins у Xcode для автоматичного запуску.

До недоліків можна віднести відсутність можливості додавати параметри для запуску таких плагінів, тому для цього потрібно використовувати додаткові конфігураційні файли.

3.4. Висновок до розділу 3

Розроблений аналізатор роботи з трафіком є зручним інструментом для виявлення та усунення проблем у коді, що впливають на використання мережі. Реалізація менеджерів і класів Visitor забезпечує модульність та легке масштабування для аналізу інших типів антипатернів.

Використання Swift Plugin забезпечує зручну інтеграцію в наявні проєкти. Окрім цього використання Swift Plugin обдумане завдяки нативності цього рішення та легшій можливості поширювати код між іншими розробниками.

Висновки

Дослідження, проведене в рамках кваліфікаційної роботи, дало змогу провести детальний аналіз антипатернів використання мережевого трафіку в мобільних застосунках та розробити інструмент для їхнього виявлення та усунення. У процесі написання роботи було детально вивчено теоретичні та практичні аспекти оцінки ефективності мережевих операцій.

Для вирішення розглянутих антипатернів було розроблено аналізатор на базі Swift Package, який використовує бібліотеку SwiftSyntax для статичного аналізу абстрактного синтаксичного дерева вихідного коду. Аналізатор автоматично виявляє ділянки коду, що містять визначені антипатерни, формулює рекомендації щодо їх виправлення, тим самим, допомагаючи розробникам оптимізувати використання трафіку.

Результати дослідження мають важливе значення для розробників екосистеми Apple, адже дозволяються підвищувати якість коду, зменшувати споживання мережевих ресурсів і покращувати користувацький досвід. Впровадження аналізатора в проєкти допомагає створювати більш стабільні, економні та користувацько-орієнтовані мобільні застосунки.

Список використаних джерел

1. Firzouq A. Apple App Store Statistics 2025: Key Insights and Trends. Tekrevol. 25.03.2025. URL: https:////www.tekrevol.com/blogs/apple-app-store-statistics/
2. An analysis of mobile application network behavior / N. Wipawee та ін. *AINTEC '16: Proceedings of the 12th Asian Internet Engineering Conference*. 2016. С. 9–16. URL: https://dl.acm.org/doi/abs/10.1145/3012695.3012697.
3. Simon K. DIGITAL 2025: MOBILE DATA CONSUMPTION TRENDS. *Dateportal*. 05.02.2025. URL: https://datareportal.com/reports/digital-2025-sub-section-mobile-data-consumption.
4. K. Lee et al. Mobile data offloading: how much can wifi deliver? In Proceedings of the 6th International COnference. *ACM*, 2010. URL: https://dl.acm.org/doi/abs/10.1145/1921168.1921203
5. Smartphone Market Share. *IDC*. 14.05.2025. URL: https://www.idc.com/promo/smartphone-market-share/.
6. Analyzing HTTP traffic with Instruments. *Apple Developer*. URL: https://developer.apple.com/documentation/foundation/analyzing-http-traffic-with-instruments.
7. Swift Syntax Documentation. *Github*. URL: https://github.com/swiftlang/swift-syntax.
8. Previewing your apps interface in Xcode. *Apple Developer*. URL: https://developer.apple.com/documentation/Xcode/previewing-your-apps-interface-in-xcode.
9. Crudu A. A Deep Dive into How HTTP Headers Influence Security and Performance in Web Applications. *MoldStud*. 13.02.2025. URL: https://moldstud.com/articles/p-a-deep-dive-into-how-http-headers-influence-security-and-performance-in-web-applications.
10. L P. D. GZIP file format specification version 4.3. *RFC 1952*. 16.05.1996. URL: https://datatracker.ietf.org/doc/html/rfc1952.

11. Jyrki A., Zoltan S. Brotli Compressed Data Format. *DataTracker*. 13.07.2016. URL: https://datatracker.ietf.org/doc/html/rfc7932.
12. Jaswanth R. V. Integrating Web Security Headers into the Secure Software Development Lifecycle: Effective Strategies and Key Considerations. *ResearchGate*. 19.12.2024. URL: https://www.researchgate.net/publication/389175603_Integrating_Web_Security_Headers_into_the_Secure_Software_Development_Lifecycle_Effective_Strategies_and_Key_Considerations.
13. Lavrenovs, A. and Melón, F.J.R. (2018). "HTTP Security Headers Analysis of Top One Million Websites." *International Conference on Cyber Conflict, CYCON*, Tallinn, 29 May-1 June 2018, pp. 345-370. doi: 10.23919/CYCON.2018.8405025.
14. URLSession. *Apple Developer*. URL: https://developer.apple.com/documentation/foundation/urlsession.
15. URLSessionConfiguration. *Apple Developer*. URL: https://developer.apple.com/documentation/foundation/urlsessionconfiguration
16. httpCookieAcceptPolicy. *Apple Developer*. URL: https://developer.apple.com/documentation/foundation/urlsessionconfiguration/httpcookieacceptpolicy
17. httpShouldSetCookies. *Apple Developer*. URL: https://developer.apple.com/documentation/foundation/urlsessionconfiguration/httpshouldsetcookies
18. httpAdditionalHeaders. *Apple Developer*. URL: https://developer.apple.com/documentation/foundation/urlsessionconfiguration/httpadditionalheaders
19. Matlock W. Efficient Network Caching in Swift with URLCache. *Medium*. URL: https://medium.com/@wesleymatlock/efficient-network-caching-in-swift-with-urlcache-9655a1248181.
20. Steinberger P. Caching file downloads with URLCache in Swift. *Nutrient.io*. 11.11.2020. URL: https://www.nutrient.io/blog/downloading-large-files-with-urlsession/.

21. Borama A. Deep dive into Apple's URLRequest Cache Policies. *Medium*. 23.08.2020. URL: https://medium.nextlevelswift.com/urlrequest-cache-policy-f7c30a96b698.
22. useProtocolCachePolicy. *Apple Developer*. URL: https://developer.apple.com/documentation/foundation/nsurlrequest/cachepolicy-swift.enum/useprotocolcachepolicy.
23. returnCacheDataElseLoad. *Apple Developer*. URL: https://developer.apple.com/documentation/foundation/nsurlrequest/cachepolicy-swift.enum/returncachedataelseload
24. timeoutIntervalForRequest. *Apple Developer*. URL: https://developer.apple.com/documentation/foundation/urlsessionconfiguration/timeoutintervalforrequest
25. Van Der Lee A. Optimizing your app for Network Reachability. *SwiftLee*. 25.08.2023. URL: https://www.avanderlee.com/swift/optimizing-network-reachability/.
26. iOS HTTP cache analysis for abusing APIs and forensics. *SilentSignal*. 06.05.2016. URL: https://blog.silentsignal.eu/2016/05/06/ios-http-cache-analysis-for-abusing-apis-and-forensics/.
27. Worldwide comparison of Internet prices in 2023 – how much does Internet cost in the United States?. *Picodi*. 15.11.2023. URL: https://www.picodi.com/us/bargain-hunting/internet-prices-2023.
28. Cellular access and discretionary downloads. *Frank van Boheemen*. URL: https://frankvanboheemen.nl/blog/cellular-and-discretionary-downloads.
29. httpMaximumConnectionsPerHost. *Apple Developer*. URL: https://developer.apple.com/documentation/foundation/urlsessionconfiguration/httpmaximumconnectionsperhost.
30. allowsExpensiveNetworkAccess. *Apple Developer*. URL: https://developer.apple.com/documentation/foundation/urlsessionconfiguration/allowsexpensivenetworkaccess
31. Supporting Low Data Mode in your app. *DonnyWals*. 23.09.2019. URL: https://www.donnywals.com/supporting-low-data-mode-in-your-app/.

32. Networking Overview. *Apple Developer*. *Documentation Archive*. URL: https://developer.apple.com/library/archive/documentation/NetworkingInternetWeb/Conceptual/NetworkingOverview/Introduction/Introduction.html.
33. Why does recursive Dispatch.main.async take longer and longer?. *Reddit*. 15.05.2022. URL: https://www.reddit.com/r/swift/comments/uq8ckv/why_does_recursive_dispatchmainasync_take_longer/?rdt=55694.
34. Yuan S., Md Gapar M. J., Jacquline T. Performance Evaluation of Software in Large Data Environments Utilizing Time-Managed Computation Tree Logic. *DLINEJOURNALS*. URL: https://www.dline.info/fpaper/jdim/v23i1/jdimv23i1_3.pdf.
35. A comparative analysis of certificate pinning in Android & iOS. *IMC '22: Proceedings of the 22nd ACM Internet Measurement Conference*. C. 605–618. URL: https://dl.acm.org/doi/10.1145/3517745.3561439.
36. SwiftParser. *SwiftInit*. URL: https://swiftinit.org/docs/swift-syntax/swiftparser/parser.
37. Meet Swift Package plugins. *Apple Developer*. URL: https://developer.apple.com/videos/play/wwdc2022/110359/.