ДЕРЖАВНИЙ УНІВЕРСИТЕТ ТЕЛЕКОМУНІКАЦІЙ

НАВЧАЛЬНО-НАУКОВИЙ ІНСТИТУТ ІНФОРМАЦІЙНИХ ТЕХНОЛОГІЙ

Кафедра Інженерії програмного забезпечення

Пояснювальна записка

до бакалаврської роботи
на ступінь вищої освіти бакалавр

на тему: **«РОЗРОБКА МОБІЛЬНОГО ДОДАТКУ «CHAMP» ДЛЯ ВЕДЕННЯ ЗДОРОВОГО СПОСОБУ ЖИТТЯ ЗА ДОПОМОГОЮ ФРЕЙМВОРКУ FLUTTER»**

Виконав: студент 4 курсу, групи ПД-42
спеціальності
121 Інженерія програмного забезпечення
(шифр і назва спеціальності)
Корж О.О.
(прізвище та ініціали)
Керівник Жебка В.В.
(прізвище та ініціали)
Рецензент _____
(прізвище та ініціали)

Київ – 2021

ДЕРЖАВНИЙ УНІВЕРСИТЕТ ТЕЛЕКОМУНІКАЦІЙ

НАВЧАЛЬНО-НАУКОВИЙ ІНСТИТУТ ІНФОРМАЦІЙНИХ ТЕХНОЛОГІЙ

Кафедра Інженерії програмного забезпечення
Ступінь вищої освіти - «Бакалавр»
Спеціальність - 121 «Інженерія програмного забезпечення»

ЗАТВЕРДЖУЮ
Завідувач кафедри
Інженерії програмного забезпечення
_____О.В. Негоденко

“____”_____2021 року

З А В Д А Н Н Я

НА БАКАЛАВРСЬКУ РОБОТУ СТУДЕНТУ

Коржу Олександру Олександровичу
(прізвище, ім‘я, по батькові)

1. Тема роботи: «Розробка мобільного додатку «Champ» для ведення здорового способу життя за допомогою фреймворку Flutter»

Керівник роботи д.т.н., доцент Жебка В.В. ,
(прізвище, ім’я, по батькові, науковий ступінь, вчене звання)

затверджені наказом вищого навчального закладу від “ 12.03 ” 2021року № 65 .

2. Строк подання студентом роботи 01.06.2021
3. Вихідні дані до роботи:
   3.1. Положення побудови онлайн систем;
   3.2. Методи побудови мобільних додатків;
   3.3. Розробка безсерверної архітектури;
   3.4. Науково-технічна література.
4. Зміст розрахунково-пояснювальної записки (перелік питань, які потрібно розробити):
   4.1. Загальні положення побудови безсерверних систем;

4.2. Аналіз технології і методів побудови додатків на Flutter;
4.3. Побудова онлайн додатку;
4.4. Висновки

5. Перелік графічного матеріалу:
5.1. Складники системи Champ;
5.2. Основні функції додатку;
5.3. Презентація

6. Дата видачі завдання 19.04.2021

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів бакалаврської роботи	Строк виконання етапів роботи	Примітка
1	Підбір науково-технічної літератури	19.04	Виконано
2	Дослідження положення побудови безсерверних систем	22.04	Виконано
3	Аналіз методів побудови додатків на Flutter	26.04	Виконано
4	Розробка додатку	14.05	Виконано
5	Висновки, оформлення роботи	16.05	Виконано
6	Розробка демонстраційних матеріалів	20.05	Виконано
7	Здача роботи	01.06	Виконано

Студент _____ Корж О.О.
(підпис) (прізвище та ініціали)
Керівник роботи _____ Жебка В.В.
підпис) (прізвище та ініціали

РЕФЕРАТ

Текстова частина бакалаврської роботи 56 стр., 10 рис., 3 табл., 15 джерел

РОЗРОБКА МОБІЛЬНОГО ДОДАТКУ «CHAMP» ДЛЯ ВЕДЕННЯ ЗДОРОВОГО СПОСОБУ ЖИТТЯ ЗА ДОПОМОГОЮ ФРЕЙМВОРКУ FLUTTER.

Мета: покращення процесу забезпечення ведення здорового способу життя за допомогою розробленого мобільного додатку на основі фреймворку Flutter.

Об'єкт дослідження – ведення здорового способу життя за допомогою фреймворку Flutter.

Предмет дослідження - фреймворк кроссплатформної розробки Flutter та хмарні рішення AWS і Firebase.

Методи дослідження. У науковій роботі використовувалися такі методи дослідження: аналіз, класифікація, пояснення, узагальнення. Використовуючи усі ці методи визначив чим кращий Flutter за його суперників, які хмарні рішення варто використовувати та яким чином можна створити онлайн додаток на Flutter.

Короткий опис. Було проведено аналіз існуючих хмарних рішень від найбільших постачальників. Виявлення переваг одного рішення над іншим. Огляд традиційної архітектури розробки та безсерверної. Переваги безсерверної архітектури. Дослідження переваг Flutter над фреймворком React Native. Після досліджень був проведений процес розробки онлайн додатку Champ, з використанням Flutter, AWS і Firebase. В процесі розробки було встановлено переваги та недоліки використаних методів розробки додатку.

ЗМІСТ

ВСТУП

Тема: розробка мобільного додатку «Champ» для ведення здорового способу життя за допомогою фреймворку Flutter.

Актуальність теми: в останній час світ стикнувся з великою проблемою, а саме карантин та локдаун. У цей час люди не мали можливості відвідувати звичні їм соцзаклади і в цей же перелік потрапили спортивні зали, та майданчики. На перший погляд може здатися, що це не дуже велика проблема для суспільства - закриття спортзалів. Очевидно, що це не самі пріорітетні сервіси, котрі потрібні людям в перший час. Проте карантин продовжувався достатньо довго і не відвідування спортивних закладів, чи ігнорування хоч якоїсь спортивної діяльності почало ставати значною проблемою для більшості людей. По-перше це погіршення здоров'я. Люди спочатку не зважають на це увагу, а потім через деякий час починаються проблеми зі спиною, суглобами, з серцем і т.д. По-друге інколи це навіть порушення балансу життя. Для багатьох людей це звично поєднувати роботу та своє спортивне життя. Хтось займається в офісі в перерві між роботою, якщо там є спортивна кімната, хтось полюбляє відвідувати спортивні зали після, або перед роботою. Проте коли почалися карантинні часи, цей звичний розпорядок порушився. Так, більшість продовжують якось займатися вдома, але не все так легко коли в тебе немає плану тренування та тренера, котрий тобі підскаже, що робити та як саме. Тому з'явилася ідея створити додаток для перегляду тренувань та спортивного контенту. Додаток буде створено для ОС Android та IOS на базі фреймворку Flutter з використанням безсерверної архітектури. Останнім часом сфера кроссплатформної розробки більш стрімко набуває популярності завдяки новому фреймворку Flutter. Це достатньо новий фреймворк - йому всього 3 роки, але незважаючи на своє відносно недовге існування, Flutter вже встиг довести те, що він є ефективним рішенням для сучасного бізнесу розробки мобільних додатків. Тому слід звернути увагу на цю нову технологію та дізнатися чому Flutter це майбутнє мобільної розробки і як він зможе змагатися з іншими кроссплатформними або навіть нативними рішеннями.

Окрім Flutter, проект матиме іншу цікаву особливість - безсерверне рішення. Зараз майже кожен новий додаток має своє бекенд середовище на якому зазвичай зберігаються дані користувачів та виконуються потрібні обчислення. Для реалізації традиційного серверного бекенду потрібно достатньо багато часу для налаштування самого серверу, а потім ще і на його підтримку. Тим паче, щоб налаштувати добре сервер і розгорнути на ньому свій веб сервіс вам знадобиться багато досвіду в цій справі. Але що робити якщо ви не хочете витрачати забагато часу на бекенд та швидше випустити свій додаток для користувачів зі всього світу? Для цього існують безсерверні, хмарні рішення. Вони допомагають уникнути довгих процесів налаштування власного серверу та дозволяють зосередитися саме на розробці бекенду. Зараз найбільшими провайдерами хмарних, безсерверних рішень є Google, Amazon, Microsoft та IBM. Звісно існують і інші, але це найпопулярніші та надійніші.

Рішення цих компаній дозволяють створити повноцінний бекенд для додатку за короткий час та меншими зусиллями. Вже не варто піклуватися про те, як саме буде створюватися веб сервер або де його будуть розташовувати. Достатньо лише думати про свою бізнес логіку та що буде робити бекенд. Звичайно не варто забувати і про саме головне на бекенді, а саме про базу даних. Потрібно якось зберігати дані додатку і така можливість у безсерверних рішеннях теж є. Це є причиною того, що за останні роки безсерверні додатки набирають популярність. Тому варто дослідити чим безсерверні рішення такі цікаві і чим вони можуть бути такими корисними.

Тобто *об'єктом дослідження* є ведення здорового способу життя за допомогою фреймворку Flutter. В результаті предметом дослідження будуть фреймворк Flutter, AWS і Firebase. Flutter буде використовуватися в якості інструменту розробки фронтенду, а AWS та Firebase в якості бекенду.

Метою диплому є покращення процесу забезпечення ведення здорового способу життя за допомогою розробленого мобільного додатку на основі фреймворку Flutter.

Завдання дослідження: поглибити свої знання у методах розробки кроссплатформних додатків, використовуючи фреймворк Flutter. Дослідити архітектуру клієнта, навчитися працювати із запитами до HTTP серверів та створювати інтерфейс користувача на основі певного дизайну. Окрім цього дослідити можливості створення повноцінних онлайн додатків, використовуючи хмарні сервіси. Навчитися використовувати базу даних DynamoDB для того, щоб мати можливість зберігати дані користувачів. Вивчити методи створення Lambda функцій для виконання бекенд коду та підключити їх до бази даних. Також зрозуміти як використовувати API Gateway, щоб була можливість звертатися до Lambda функцій через HTTP запит, прямо із клієнту.

1. ОГЛЯД БЕЗСЕРВЕРНИХ РІШЕНЬ

1.1 Архітектура програмних систем

Загалом архітектуру програмного забезпечення можна сприймати як план або концептуальну модель додатку. Архітектура або її відсутність можуть створити або зруйнувати ваш майбутній додаток. Правильна архітектура може допомогти масштабувати веб або мобільний додаток в рази простіше та швидше, а погана архітектура або її повна відсутність може створити великі проблеми, потребуючі великої праці для налагодження процесу.

Розуміння того яку архітектуру слід вибрати має велике значення для створення ефективних, потужних та успішних додатків.

Безсерверна розробка виходить за рамки традиційних серверних архітектур, які потребують тісної взаємодії з сервером. З цим підходом використовуються тільки хмарні сервіси, котрі будуть виконувати роль бекенд системи. За допомогою сучасних хмарних рішень для безсерверної розробки можна створювати такі системи, котрі будуть справлятися з масштабуванням або високими обчислювальними потребами без необхідності використання серверів.

Окрім цього тепер розробляти додаток стало швидше, таким чином реліз додатку компаній прискорюється, але при цьому якість програмного забезпечення та його потужність не страждають. Проте все ж таки потрібно розуміти, що «безсерверність» не означає, що бекенд система зовсім без серверу. Сервера присутні, але вони сховані від користувачів хмарних рішень компаніями провайдерами, котрі замість них пропонують свої хмарні сервіси.

Якщо подивитися на системи, на яких працюють більшість сучасних додатків, то можна побачити, що в них використовуються внутрішні сервери, виконуючі різні обчислення в якості бекенду, та клієнтські додатки, котрі забезпечують розробників інтерфейсами користувача. В стандартному веб додатку сервер приймає HTTP-запити від клієнтів і обробляє їх. Він може виконувати якісь

обчислення або інші потрібні алгоритми після чого зберігати їх в базу даних. Після цього серверна частина генерує відповідь, зазвичай у JSON форматі і відправляє її до клієнту.

Звичайно більшість систем стає більш складними, якщо враховувати такі елементи як балансування навантаження, транзакції, кластеризація, кешування та обмін повідомленнями. Для більшості цього програмного забезпечення потрібні сервера, працюючі у великих дата центрах, котрі потребують постійної підтримки та управління.

Ініціалізація, управління та налаштування серверів - це все досить складні задачі, для рішення яких зазвичай потрібні спеціальні оператори. Інфраструктура та обладнання є необхідними компонентами будь-якої IT системи, але часто вони також відволікають від того, що повинно бути в центрі уваги - рішення бізнес проблеми.

За останні часи з‘явилися такі технології, як PaaS та контейнери, котрі є потенційним рішенням проблеми несумісних інфраструктур, конфліктів та управління серверами. PaaS - це форма хмарних обчислень, яка дає користувачам платформу для запуску свого програмного забезпечення, ховаючи при цьому деяку частину базової інфраструктури. Щоб ефективно використовувати PaaS, розробникам слід писати програмне забезпечення, націлене на можливості платформи. Проте все ж таки платформа як сервіс є одним із рішень головного болю розробників для розробки сучасних додатків.

Контейнеризація - це спосіб ізолювати додаток від його власного середовища. Але все одно контейнер потрібно влаштовувати на сервері якоїсь хмари або на своєму. Це не так просто як запустити код в хмарі та виконати його.

Нарешті безсерверні хмарні рішення, а саме Lambda. За допомогою lambda можна виконувати код без додаткових складних налаштувань серверу, встановлення програмного забезпечення або влаштування контейнеру. У сучасних хмарних рішеннях всі інфраструктурні процеси зазвичай сховані від розробників і їм не варто перейматися складністю налаштування масштабування чи інших

серверних процесів. Використовуючи ці хмарні рішення розробники-користувачі можуть швидко створювати слабко зв’язані, масштабовані і ефективні архітектури.

1.1.1 Сервіс-орієнтована архітектура

Серед системної і прикладної архітектури сервіс-орієнтована архітектура (SOA) отримала широке визнання серед розробників програмного забезпечення. Це архітектура, в якій чітко сформульована ідея про те, що система може складатися з безлічі незалежних сервісів. SOA не вимагає використання якої-небудь конкретної технології. Замість цього вона заохочує архітектурний підхід, при якому розробники створюють автономні сервіси, які обмінюються даними за допомогою передачі повідомлень і часто мають схему або контракт, котрі визначають як саме сервіси мають спілкуватися між собою. Повторне використання і автономність сервісу, можливість компонування, деталізація і обслугованність - усе це важливі принципи, пов'язані з SOA. Мікросервіси та безсерверні архітектури успадкували методи сервіс-орієнтованності. Вони зберігають багато з вищезгаданих принципів і ідей, намагаючись вирішити складність застарілої сервіс-орієнтованної архітектури.

1.1.2 Мікросервіси

Останнім часом спостерігається тенденція впровадження систем з мікросервісами. Розробники схильні думати про мікросервіси як про невеликі, автономні, повністю незалежні сервіси, побудовані навколо певної бізнес-цілі [4; 15]. В ідеалі мікросервіси мають бути легко замінені, при чому кожен сервіс має бути написаний на відповідній платформі і на відповідній мові. Сам факт, що мікросервіси можуть бути написані на різних мовах загального призначення або предметно-орієнтованих мовах(DSL), є візитною карткою для багатьох розробників. Користі можна отримати, використовуючи правильну мову або спеціалізований набір бібліотек для роботи. Проте, це теж часто буває пасткою. Може бути важко підтримувати поєднання мов і фреймворків, а без суворої

дисципліни це може призвести до плутанини в майбутньому. Кожен мікросервіс може підтримувати власний стан/контекст і зберігати дані. А якщо мікросервіси правильно розділені, групи розробників можуть працювати і розгортати мікросервіси незалежно один від одного. З іншого боку, можлива узгодженість, управління транзакціями і комплексне відновлення після помилок можуть ускладнити завдання. Можна стверджувати, що безсерверна архітектура також утілює багато принципів мікросервісів. Врешті-решт, залежно від того, як розробник проектує систему, кожну обчислювальну функцію можна розглядати як окремий сервіс. Але йому не треба повністю використовувати принципи мікросервісів, якщо він цього не хочете. Безсерверні архітектури дають розробнику свободу застосовувати стільки принципів мікросервісів, скільки йому треба, не примушуючи його вибирати один шлях.

Проектування програмного забезпечення перетворилося з днів, коли код виконувався на мейнфреймах, у багаторівневі системи, в яких рівні представлення, даних і логіки займають видне місце у багатьох проектах. Усередині кожного рівня може бути декілька логічних рівнів, які мають справу з конкретними аспектами функціональності або предметної області. Існують також наскрізні компоненти, такі як системи ведення журналів або обробка виключень, які можуть охоплювати безліч рівнів. Перевага нашарування зрозуміла. Багатошаровість дозволяє розробникам розділити проблеми і мати зручніші в обслуговуванні застосування. Але може бути і зворотній ефект. Надто багато шарів може привести до неефективності. Невелика зміна часто може відбуватися каскадом і примушувати розробника змінювати кожен рівень в системі, що вимагає значних витрат часу і енергії на реалізацію та тестування. Чим більше шарів, тим складнішою і громіздкішою може стати система з часом.

Безсерверні архітектури можуть допомогти з проблемою розділення на рівні і необхідності оновлювати надто багато речей. У розробників є можливість видалити або мінімізувати багаторівневий ефект, розбивши систему на функції і дозволивши клієнтській частині безпечно взаємодіяти із службами і навіть з базою

даних безпосередньо. Усе це можна зробити організованим способом, щоб запобігти спагеті-реалізації і проблеми із залежностями, чітко визначивши межі сервісів, дозволивши функціям Lambda бути автономними і спланувавши взаємодію функцій і сервісів.

Безсерверний підхід не вирішує усіх проблем і не усуває приховані складнощі системи. Але при правильній реалізації він може надати можливості для зменшення, організації і управління складністю. Добре спланована безсерверна архітектура може спростити майбутні зміни, що є важливим чинником для будь-якого довгострокового застосування [3].

1.2 Принципи безсерверної архітектури

1.2.1 Обчислювальний сервіс

Безсерверна архітектура - природне продовження ідей, висунених в SOA. В ній увесь призначений для користувача код пишеться і виконується як ізольовані, незалежні функції, які виконуються в обчислювальному сервісі без збереження стану, такому як AWS Lambda наприклад. Розробники можуть писати функції для виконання практично будь-якого загального завдання, таких як читання і запис в джерело даних, виклик інших функцій і виконання обчислень. У складніших випадках розробники можуть налаштувати складніші процеси і організувати виклики декількох функцій. Можуть бути сценарії, коли сервер все ще потрібний для чогось. Проте таких випадків може бути дуже мало, і як розробникові слід уникати запуску і взаємодії з сервером, якщо це можливо.

1.2.2 Одноцільові функції

Як програмісту, слід розробляти свої функції з урахуванням принципу єдиної відповідальності(SRP). Функція, яка виконує тільки одну дію, більш придатна до тестів і надійна, а також призводить до меншої кількості помилок і несподіваних побічних ефектів. Складаючи і комбінуючи функції і сервіси у вільному порядку, користувачі можуть створювати складні серверні системи, які як і раніше зрозумілі

і прості в управлінні. Функція з чітко налаштованим інтерфейсом також з більшою вірогідністю буде повторно використана у безсерверній архітектурі. Код, написаний для обчислювального сервісу, такого як Lambda, має бути створений в стилі без збереження стану. Розробники не повинні розраховувати на те, що локальні ресурси або процеси виживуть після завершення сеансу.

1.2.3 Хмарні SDK на клієнті

Важливо пам'ятати, що призначений для користувача код, що виконується в Lambda, повинен виконуватися швидко. Функції, які завершуються раніше, - дешевше, тому що ціни на Lambda залежать від кількості запитів, тривалості виконання і об'єму виділеної пам'яті. Менше процесів в Lambda - дешевше. Більше того, створення багатофункціонального інтерфейсу, який може безпосередньо взаємодіяти із сторонніми сервісами, може сприяти кращій взаємодії з користувачем. Менша кількість переходів між онлайн-ресурсами і зменшення затримки приведе до кращого сприйняття продуктивності і зручності використання додатка. Іншими словами, не треба все отримувати через обчислювальний сервіс. Ваш клієнтський інтерфейс може мати можливість безпосередньо зв'язуватися з пошуковою системою, базою даних або іншим корисним API.

1.3 Провайдери хмарних рішень

1.3.1 Веб-сервіси Amazon

AWS - старий постачальник загальнодоступних хмар з щонайширшим спектром продуктів, можливостей обчислень і зберігання даних, а також керованих послуг. Вони приділяють велику увагу передовим методам безпеки і архітектури. Їх корпоративні інфраструктури, такі як Well - Architected Framework і Cloud Adoption Framework, були розроблені на основі їх досвіду роботи з великими корпоративними клієнтами. Їх ринкова доля в 33% - тому підтвердження. Проте, це не найдешевша хмара на ринку.

Коли вибирати AWS

Знаходячись на ринку майже 15 років, AWS накопила найбільший досвід роботи. Це найбільш зрілий постачальник хмарних послуг, який надає хмарні послуги мільйонам клієнтів по всьому світу, використовуючи самі різні сценарії використання. AWS безперечно є законодавцем трендів в галузі. У нього є сила впливати на ринок. AWS - відмінний вибір як для стартапів, так і для підприємств. AWS надає широкий спектр послуг, від веб-застосувань і аналітичних робочих навантажень до великомасштабних міграцій центрів обробки даних. Що стосується обчислень, AWS надає найширший вибір типів віртуальних машин. AWS також нині пропонує самі різні на ринку варіанти обчислень і сховищ. Для робочих навантажень машинного навчання і штучного інтелекту AWS також надає самі кращі конфігурації типів віртуальних машин з підтримкою графічного процесора.

Що стосується реляційної бази даних, AWS підтримує бази даних для MySQL, PostgreSQL, MariaDB, Oracle(як SE, так і EE) і MS SQL(версії Web і Enterprise) у рамках своєї пропозиції RDS. Крім того, у них є власна база даних, сумісна з MySQL і PostgreSQL, яка може похвалитися продуктивністю, подібною Oracle, за невелику вартість. Для баз даних NoSQL компанія AWS пропонує продукт DynamoDB.

AWS конкурує з GCP в хмарному машинному навчанні. У 2016 році Amazon представила довгий список нових сервісів на основі штучного інтелекту. AWS розширив свої пропозиції, щоб бути конкурентоздатним з хмарним штучним інтелектом Google. Відтоді Amazon додала такі сервіси, як SageMaker, який швидко навчає моделі машинного навчання для швидшого розгортання, і AWS DeepLens, відеокамеру з підтримкою глибокого навчання.

1.3.2 Microsoft Azure

Microsoft відставала від AWS в грі із загальнодоступними хмарами, але в першу чергу вона зосередилася на пропозиціях SaaS і PaaS, оскільки її сильні сторони полягають як в корпоративному, так і в споживчому ПЗ. Спочатку Microsoft зосередилася на послугах PaaS для Azure. Вони були зосереджені на

існуючій базі розробників Microsoft. З часом Microsoft розширила свою увагу на послуги як Linux, так і IaaS. З часом Microsoft також зробила Azure зручнішим для стартапів і створила підтримку API для своїх різних сервісів. Проте, незважаючи на широту послуг, що надаються, Microsoft істотно відстає від AWS по впровадженню на підприємствах.

Коли вибирати Azure

Azure - це хмарна платформа з великою кількістю функцій, яка може бути переважною платформою для клієнтів, які вже тим або іншим чином використовують продукти Microsoft. Хоча Azure підтримує ряд сервісів на основі продуктів з відкритим кодом, хмарне портфоліо Microsoft - це те, що відрізняє його від клієнтів. У Azure є більше 151 типу віртуальних машин з 26 сімейств віртуальних машин, які підтримують все, від невеликих веб-сервісних навантажень до робочих навантажень HPC, Oracle і SAP. У Azure є як Windows, так і декілька різновидів Linux(RHEL, CentOS, SUSE, Ubuntu). У Azure є окреме сімейство екземплярів для робочих навантажень машинного навчання і штучного інтелекту. Якщо треба виконувати високопродуктивні робочі навантаження, що вимагають до 128 віртуальних ЦП і 3,5 ТБ пам'яті, Azure - хороший вибір.

Azure також був першим хмарним гравцем, який усвідомив тенденцію до гібридної хмари, і мав одну з перших гібридних хмар. Клієнти, яким потрібний інтерфейс Azure, але хотіли запускати служби у власних центрах обробки даних, могли використати Azure Stack. Інші хмарні гравці тільки наздоганяють Azure в цій області.

Коли справа доходить до баз даних SQL і NoSQL, в Azure є досить добре продуманий набір послуг. Він надає керований MS SQL Server і сховище даних SQL. Azure також надає керовані бази даних для MySQL, PostgreSQL і MariaDB. Таблиця Azure - це кероване сховище значень ключів, тоді як CosmosDB надає багатомодельну глобально розподілену базу даних NoSQL з декількома моделями узгодженості. Він надає API, сумісний з MongoDB, Cassandra, Gremlin і сховищем таблиць Azure. Якщо треба запустити декілька моделей керованих даних,

включаючи моделі даних документу, графіку, пари "ключ-значення", таблиці і сімейства стовпців, в одній хмарі, Cosmos може бути відповідним варіантом.

1.3.3 Хмарна платформа Google

Хмарна платформа Google(GCP), що хоча і спізнюється в гру, і маючи найменшу частку ринку серед постачальників загальнодоступних хмарних сервісів в порівнянні з іншими, демонструє стійке процентне зростання. Він може похвалитися декількома функціями, які дозволяють йому випереджати конкурентів в певних областях. GCP також привертає увагу не лише нових клієнтів, які вже є частиною екосистеми Google, але і користувачів, які вже використовують хмарні технології, але прагнуть розширити своє середовище до Google у рамках мультихмарної стратегії. Google також розпочав з послуг PaaS, але постійно розширює своє продуктове портфоліо. Окрім інноваційних функцій, Google може похвалитися найнижчою ціною на інфраструктуру в порівнянні з усіма іншими постачальниками хмарних послуг. Проте звичайно, загальні витрати будь-якого підприємства залежать від використовуваних послуг і прийнятих заходів з управління витратами.

Коли вибирати GCP

За останні роки Google організував більше 2000 проектів з відкритим кодом і став видатним учасником OSS. В цьому відношенні Google поглинає Microsoft і Amazon. Google інтегрує такі продукти, як Kubernetes, Apache Beam, TensorFlow, у свою хмарну платформу Google і пропонує їх в якості керованих сервісів, якими управляють його партнери.

GCP також просувається вперед в області аналітики і машинного навчання. BigQuery і Dataflow від Google пропонують потужні можливості аналітики і обробки для підприємств, які працюють з величезними об'ємами даних, тоді як контейнерна технологія Google Kubernetes забезпечує ефективне управління кластером контейнерів і спрощує розгортання контейнерів. За допомогою Google

Cloud Machine Learning Engine і різних API машинного навчання компанії можуть миттєво використати штучний інтелект в хмарі.

Ціноутворення GCP - його найсильніша сторона. Він тарифікується щохвилини(мінімум 10 хвилин), і користувачі витрачають гроші тільки за використаний час обчислень. Більше того, GCP надає знижки на тривалі робочі навантаження без яких-небудь попередніх зобов'язань. AWS, навпаки, вимагає передоплати у вигляді "зарезервованих екземплярів", щоб мати право на знижки. А Azure пропонує тільки 5% знижку на 12-місячну передоплату.

Для реляційних баз даних GCP забезпечує підтримку керованих баз даних MySQL і PostgreSQL. Для клієнтів, яким потрібна глобально розподілена база даних, яка як і раніше підтримує узгодженість і властивості ACID, GCP створив Spanner. Spanner використовує алгоритми консенсусу і атомний годинник для синхронізації транзакцій між вузлами. Ця пропозиція є унікальною для GCP і робить Spanner дуже привабливим для великих корпоративних клієнтів, які потребують ці вимоги до свого сховища реляційних даних. З точки зору NoSQL, у GCP є продукт під назвою BigTable. BigTable - це керована база даних NoSQL зі стовпцями петабайтного масштабу, яка використовується Google у власних продуктах, таких як Gmail.

Тепер давайте проглянемо загальне порівняння цих трьох платформ:

Таблиця 1.1 – Порівняння GCP, Azure, AWS

	Azure	AWS	GCP
Років на ринку	10 (2010)	16 (2004)	12 (2008)
Володіння ринком	16.9%	32.3%	5.8%
Доступність	60 регіонів	24 регіони	24 регіони

Сервісів	Більше 200	Більше 212	Більше 90
Сховище	Azure Storage	S3, EBS, EFS, S3 Glacier	GC Storage, Nearline, Google Persistent Storage
Бази даних	Azure HDInsight, Azure table	RDS, DynamoDB	Spanner, BigTable
Безсерверні обчислення	Azure Functions	AWS Lambda	Google Cloud Functions
Ціноутворення	Кожну хвилину	Кожну годину	Кожну хвилину
Клієнти	HP, Apple, Fujifilm, Polycom	BMW, Netflix, Airbnb, Samsung, Facebook	Vodafone, Toyota, LG, Spotify

Як видно з таблиці AWS більш розвинутіша платформа. Окрім того, що вона має достатньо багато різних сервісів, ще одною перевагою є роки на ринку. Для розробників це означає, що для неї випущено більше інформації та документації, в якій можна дізнатися як варто використовувати AWS. Тому вибір для додатку Champ – це AWS.

2. МЕТОДИ РЕАЛІЗАЦІЇ ПРОЕКТУ

2.1 Веб-сервіси Amazon

2.1.1 DynamoDB

Amazon DynamoDB - це повністю керований сервіс бази даних NoSQL, яка забезпечує швидку і передбачувану продуктивність з масштабованістю. DynamoDB

дозволяє позбавитися від адміністративного навантаження, пов'язаного з експлуатацією і масштабуванням розподіленої бази даних, так що не доведеться турбуватися про виділення устаткування, налаштування, реплікацію, установку виправлень програмного забезпечення або масштабування кластера. DynamoDB також пропонує шифрування в стані спокою, що усуває операційне навантаження і складність, пов'язані із захистом конфіденційних даних.

З DynamoDB можна створювати таблиці бази даних, які можуть зберігати і витягати будь-який об'єм даних і обслуговувати будь-який рівень трафіку запитів. Також є можливість збільшувати або зменшувати пропускну спроможність таблиць без простоїв або зниження продуктивності. Консоль управління AWS можна використати для відстежування показників використання ресурсів і продуктивності. DynamoDB надає можливість резервного копіювання за запитом. Це дозволяє створювати повні резервні копії ваших таблиць для довгострокового зберігання і архівації для відповідності нормативним вимогам.

Ви можете створювати резервні копії за запитом і включати відновлення на певний момент часу для таблиць Amazon DynamoDB. Відновлення на певний момент часу допомагає захистити ваші таблиці від випадкових операцій запису або видалення. За допомогою відновлення на певний момент часу можна відновити таблицю на будь-який момент часу впродовж останніх 35 днів. DynamoDB дозволяє автоматично видаляти прострочені елементи з таблиць, щоб зменшити використання сховища і вартість зберігання даних, які більше не актуальні.

Переваги

1. Швидка і стабільна робота. Рішення Amazon DynamoDB працює стабільно і швидко в системах будь-якого масштабу у будь-якій сфері застосування. Середній час обробки запиту на сервері складає декілька мілісекунд. У міру зростання об'ємів даних і підвищення необхідної продуктивності система Amazon DynamoDB забезпечує відповідність вимогам до пропускної

спроможності і часу обробки запиту за допомогою технологій автоматичного розбиття на розділи і SSD в системах будь-якого масштабу.

2. Висока масштабованість. При створенні таблиці досить вказати, який об'єм ресурсів для обробки запитів потрібно. Якщо необхідно змінити пропускну спроможність, просто оновите параметри таблиці за допомогою консолі управління AWS або Amazon DynamoDB API. Amazon DynamoDB виконує усі операції по масштабуванню в прихованому режимі і в ході їх виконання продовжує забезпечувати відповідність встановленим вимогам до пропускної спроможності.
3. Повне управління. Amazon DynamoDB - це повністю керований хмарний сервіс баз даних NoSQL. Досить створити таблицю баз даних, настроїти пропускну спроможність, і сервіс виконає усі інші необхідні дії. Вам більше не треба турбуватися про управління базою даних, апаратним і програмним забезпеченням, налаштуваннями і конфігурацією, оновленнях ПЗ, використанні надійного розподіленого кластера бази даних або розділенні інформації на декілька блоків у міру зміни масштабів системи.
4. Програмування на основі подій. Amazon DynamoDB інтегрується з сервісом AWS Lambda, надаючи тригери, які дозволяють проектувати додатка з функцією автоматичного реагування на зміни даних.
5. Точний контроль доступу. Amazon DynamoDB інтегрується з AWS IAM, забезпечуючи точний контроль доступу користувачів у вашій організації. Можна призначити окремі права і дозволи кожному користувачеві і відстежувати його доступ до усіх сервіс
6. Гнучкість. Amazon DynamoDB підтримує роботу із структурами даних на основі як документів, так і пар "ключ-значення", завдяки чому є можливість вибрати оптимальну архітектуру з урахуванням особливостей своєї системи.

В DynamoDB таблиці, елементи і атрибути є основними компонентами, з якими розробник працює. Таблиця - це набір елементів, а кожен елемент - це набір атрибутів. DynamoDB використовує первинні ключі для унікальної ідентифікації

кожного елементу в таблиці і вторинні індекси, щоб забезпечити велику гнучкість запитів. Також використати DynamoDB Streams для запису подій зміни даних в таблицях DynamoDB.

Таблиці. Як і інші системи баз даних, DynamoDB зберігає дані в таблицях. Таблиця - це набір даних. Наприклад, можна використати для зберігання особистої контактної інформації про друзів, сім'ї або будь-яку іншу інформацію що цікавить вас. Також може бути таблиця "Машини" для зберігання інформації про автомобілі, якими управляють люди.

Елементи - кожна таблиця містить нуль або більше за елементи. Елемент - це група атрибутів, яка однозначно ідентифікується серед усіх інших елементів. У таблиці "Люди" кожен елемент представляє людину. У таблиці "Машини" кожен елемент представляє одно транспортний засіб. Елементи DynamoDB багато в чому схожі на рядки, записи або кортежі в інших системах баз даних. У DynamoDB немає обмежень на кількість елементів, які можна зберігати в таблиці.

Атрибути - кожен елемент складається з одного або декількох атрибутів. Атрибут - це фундаментальний елемент даних, який не вимагає подальшого розбиття. Наприклад, елемент в таблиці "Люди" містить атрибути з іменами PersonID, LastName, FirstName і т. Д. Для таблиці "Відділ" елемент може мати такі атрибути, як "ИД відділу", "Ім'я", "Менеджер" і так далі. Атрибути в DynamoDB багато в чому схожі на поля або стовпці в інших системах баз даних.

2.1.2 API Gateway

Amazon API Gateway - це повністю керований сервіс для розробників, призначений для створення, публікації, обслуговування, моніторингу і забезпечення безпеки API у будь-яких масштабах. Досить всього декількох клацань на панелі управління AWS, щоб створити API, який функціонуватиме як "вхідні двері" для додатків при здійсненні доступу до даних, бізнес-логіки або функцій з серверних сервісів, наприклад робочих навантажень в Amazon EC2, коду в AWS Lambda або будь-якого інтернет-застосування. Amazon API Gateway виконує будь-

які завдання, пов'язані з прийомом і обробкою сотень тисяч одночасних викликів API, включаючи управління трафіком, авторизацію і контроль доступу, моніторинг і управління версіями API.

2.1.3 Lambda функції

AWS Lambda дозволяє запускати код без виділення серверів і управління ними. Користувачі оплачують тільки витрачений обчислювальний час, а періоди, коли код не виконується, не тарифікуються. Сервіс Lambda дозволяє виконувати код практично для будь-яких типів додатків і серверних сервісів і вимагає практично нульових зусиль із адміністрування. Досить завантажити свій код, і Lambda зробить усе необхідне для запуску, масштабування і забезпечення високої доступності вашого коду. Можна налаштувати автоматичний запуск коду з інших сервісів AWS або викликати його безпосередньо з будь-якого інтернет-застосування або мобільного застосування [16].

Тепер давайте розглянемо детальніше особливості цих Lambda функцій:

Створюйте власні безсерверні сервіси

Ви можете використати AWS Lambda для створення нових безсерверних сервісів для ваших застосувань, які запускаються за запитом з використанням Lambda API або призначених для користувача API, створених за допомогою Amazon API Gateway.

Знайомі мови програмування

AWS Lambda не вимагає вивчення нових мов, інструментів або фреймворків. Також є можливість використати будь-яку сторонню бібліотеку. Окрім цього можна упакувати будь-який код(фреймворки, SDK, бібліотеки і т.д.) як Lambda Layer, а також легко управляти ним і ділитися їм за допомогою декількох функцій. Lambda підтримує код Java, Go, PowerShell, Node.js, C #, Python і Ruby і надає API, який дозволяє використати будь-які додаткові мови програмування для створення ваших функцій.

Повністю автоматизоване адміністрування

AWS Lambda управляє усією інфраструктурою для запуску вашого коду у високодоступній відмовостійкій інфраструктурі, дозволяючи зосередитися на створенні власних безсерверних сервісів. З Lambda ніколи не доведеться оновлювати ОС при випуску патча або турбуватися про зміну розміру або додавати нових серверів у міру зростання вашого використання. AWS Lambda легко розгортає ваш код, виконує усе адміністрування, обслуговування і налаштування безпеки, а також забезпечує вбудоване ведення журналу і моніторинг через Amazon CloudWatch.

Автоматичне масштабування

AWS Lambda викликає код користувача тільки при необхідності і автоматично масштабується для підтримки швидкості запитів, що приходять, не вимагаючи від користувача нічого налаштовувати. Немає обмежень на кількість запитів, які може обробити ваш код. AWS Lambda зазвичай запускає ваш код впродовж декількох мілісекунд після події, і, оскільки Lambda масштабується автоматично, продуктивність залишається стабільно високою у міру збільшення частоти подій. Оскільки ваш код не має стану, Lambda може запускати будь-яку кількість його екземплярів без тривалих затримок з розгортанням і налаштуванням.

Підключення до реляційних баз даних

Використайте Amazon RDS Proxy, щоб скористатися перевагами повністю керованих пулів з'єднань для реляційних баз даних. Проксі-сервер RDS ефективно управляє тисячами одночасних підключень до реляційних баз даних, спрощуючи створення високомасштабованих, безпечних безсерверних застосувань на основі лямбда-кода, яким необхідно взаємодіяти з реляційними базами даних. Нині RDS Proxy пропонує підтримку MySQL і Aurora. Можна використати RDS Proxy для своїх бессерверных застосувань через консоль Amazon RDS або через консоль AWS Lambda.

Підключення до загальних файлових систем

Amazon Elastic File System для AWS Lambda дозволяє безпечно читати, записувати і зберігати великі об'єми даних з малою затримкою в будь-якому масштабі. Вам не треба писати код для завантаження даних в тимчасове сховище для їх обробки. Це економить час і спрощує код, дозволяючи зосередитися на своїй бізнес-логіці. EFS для Lambda ідеально підходить для створення додатків машинного навчання або завантаження великих еталонних файлів або моделей, обробки або резервного копіювання великих об'ємів даних, розміщення веб-контенту або обміну файлами між безсерверними застосуваннями і додатками на основі екземплярів або контейнерів.

2.2 Firebase

Firebase - це бекенд як послуга(Baas). Він надає розробникам безліч інструментів і послуг, які допомагають їм розробляти якісні застосування, розширювати призначену для користувача базу і отримувати прибуток. Він побудований на інфраструктурі Google.

2.2.1 Firebase Auth

Більшості додатків необхідно знати особу користувача. Знання особи користувача дозволяє додатку безпечно зберігати призначені для користувача дані в хмарі і забезпечувати однакову роботу, що персоналізується, на усіх девайсах користувача. Firebase Authentication надає серверні служби, прості у використанні SDK і готові бібліотеки призначені для аутентифікації користувачів у вашому додатку. Він підтримує аутентифікацію з використанням паролів, номерів телефонів, популярних федеративних постачальників посвідчень, таких як Google, Facebook і Twitter, і інших. Firebase Authentication тісно інтегрується з іншими сервісами Firebase і використовує галузеві стандарти, такі як OAuth 2.0 і OpenID

Connect, тому його можна легко інтегрувати з вашим призначеним для користувача сервером.

Щоб створити аккаунт користувача у своєму додатку, спочатку слід отримати від користувача облікові дані для аутентифікації. Ці облікові дані можуть бути адресою електронної пошти і паролем користувача або токеном OAuth від федеративного постачальника посвідчень. Потім передати ці облікові дані в Firebase Authentication SDK. Після, серверні служби Google Firebase перевірять ці облікові дані і повернуть відповідь клієнтові. Після успішного входу можна отримати доступ до основної інформації профілю користувача і контролювати доступ користувача до даних, що зберігаються в інших продуктах Firebase. Також є можливість використати наданий токен аутентифікації для перевірки особи користувачів у власних серверних службах.

2.2.2 Firebase Storage

Хмарне сховище Firebase створене для розробників додатків, яким необхідно зберігати і обслуговувати призначений для користувача контент, наприклад фотографії або відео. Хмарне сховище Firebase - це потужний, простий і економічний сервіс зберігання об'єктів. SDK Firebase для хмарного сховища додають безпеку Google до завантаження і скачування файлів для ваших додатків Firebase незалежно від якості мережі.

SDK Firebase для хмарного сховища виконує вивантаження і скачування незалежно від якості мережі. Завантаження і скачування є надійними, тобто вони перезапускаються з того місця, де були зупинені, що економить час і пропускну спроможність користувачів. Також SDK Firebase для хмарного сховища інтегруються з Firebase Authentication, щоб забезпечити просту і інтуїтивно зрозумілу аутентифікацію для розробників. Можна використати декларативну модель безпеки, щоб дозволити доступ на основі імені файлу, розміру, типу вмісту і інших метаданих. Хмарне сховище розраховане на эксабайтный масштаб, коли застосування стає популярним. Легко можна перейти від прототипу до

комерційного застосування, використовуючи ту ж інфраструктуру, що і Spotify або Google Фото.

Розробники використовують SDK Firebase для хмарного сховища для завантаження і скачування файлів безпосередньо з клієнтів. Cloud Storage Firebase зберігає файли в кошику Google Cloud Storage, що робить їх доступними як через Firebase, так і через Google Cloud. Це дозволяє гнучко завантажувати і викачувати файли з мобільних клієнтів через Firebase SDK для хмарного сховища. Крім того, можна виконувати обробку на стороні сервера, таку як фільтрація зображень або перекодування відео, за допомогою API Google Cloud Storage.

2.3 Dart

Dart - це оптимізована для клієнтів мова для розробки швидких додатків на будь-якій платформі. Його мета - бути найбільш продуктивною мовою програмування для багатоплатформної розробки.

Мови програмування визначаються їх технічними можливостями. Dart розроблений з технічними можливостями, які особливо підходять для розробки клієнтів, приділяючи пріоритетну увагу як розробці (миттєве перезавантаження з відстеженням стану), так і високоякісним виробничим можливостям для широкого спектра цілей компіляції (веб, мобільні пристрої та настільні комп'ютери).

Dart безпечний по типу; він використовує перевірку статичного типу, щоб гарантувати, що значення змінної завжди відповідає статичному типу змінної. Хоча типи є обов'язковими, анотації типів не є обов'язковими через виведення типу. Система типізації Dart також є гнучкою, дозволяючи використовувати динамічний тип в поєднанні з перевірками під час виконання, що може бути корисно під час експериментів або для коду, який повинен бути особливо динамічним.

Dart забезпечує надійну null безпеку, що означає, що значення не можуть бути null, якщо розробник сам не напише, що вони можуть бути такими. Володіючи

надійної null безпекою, Dart може захистити розробника від null винятків під час виконання з допомогою статичного аналізу коду.

Dart також становить основу Flutter. Dart надає мову і середу виконання, які використовуються в додатках Flutter, але Dart також підтримує багато основних завдань розробника, такі як форматування, аналіз і тестування коду.

На початковому етапі розробки команда Flutter вивчила безліч мов і середовищ виконання і в кінцевому підсумку прийняла Dart для фреймворка і віджетів. Flutter використовував чотири основних параметри для оцінки і враховував потреби авторів, розробників і кінцевих користувачів фреймворка. Вони виявили, що багато мов відповідають деяким вимогам, але Dart отримав високі бали за всіма параметрами оцінки і відповідав всім вимогам і критеріям команди Flutter. Середовище виконання і компілятори Dart підтримують комбінацію двох важливих функцій для Flutter: швидкий цикл розробки на основі JIT, який дозволяє змінювати форму і миттєве перезавантаження зі збереженням стану на мові з типами, а також випереджаючий компілятор, який генерує ефективний код ARM для швидкого запуску.

Одне з основних ціннісних пропозицій Flutter полягає в тому, що він економить інженерні ресурси, дозволяючи розробникам створювати додатки для iOS і Android з однієї і тієї ж кодової базою. Використання високопродуктивної мови прискорює роботу розробників і робить Flutter більш привабливим. Велика частина Flutter побудована на тій же мові, не жертвуючи доступністю або можливістю читання фреймворка і віджетів для розробників.

Для Flutter потрібна мова, яка підходить для проблемної області Flutter: створення візуальної взаємодії з користувачем. В галузі накопичено багаторічний досвід створення фреймворків призначеного для користувача інтерфейсу на об'єктно-орієнтованих мовах. Переважна більшість розробників мають досвід об'єктно-орієнтованої розробки, що спрощує навчання розробці за допомогою Flutter.

Flutter команда хотіла дати розробникам можливість створювати швидкі і гнучкі призначені для користувача інтерфейси. Для цього потрібно мати можливість запускати значний обсяг коду кінцевого розробника під час кожного кадру анімації. Це означає, що потрібна мова, яка забезпечує високу продуктивність і передбачувану продуктивність без періодичних пауз, які можуть викликати пропадання кадрів.

Dart підтримує як своєчасну (JIT) компіляцію, так і попередню (AOT) компіляцію:

- Компілятор AOT перетворює Dart в ефективний власний код. Це робить Flutter швидким (перемога для користувача і розробника), але також означає, що (майже) весь фреймворк написаний на Dart. Для розробника, це означає, що можна налаштувати практично все.
- Додаткова JIT-компіляція Dart дозволяє виконувати миттєве перезавантаження. Швидка розробка і ітерація - ключ до щастя використання Flutter.

Dart - продуктивна, передбачувана мова. Його легко вивчити, і він здається знайомим. Незалежно від того, чи використовувати динамічний або статичний стиль, можна легко приступити до роботи. Також наявність Dart у Google - це перевага. За останні кілька років Dart домігся великих успіхів, щоб стати хорошим мовою спеціально для написання сучасного інтерфейсу користувача. Система типів в Dart 5 і об'єктна орієнтація спрощують створення багаторазових компонентів для користувача інтерфейсу. А Dart включає в себе кілька функцій функціонального програмування, які спрощують перетворення ваших даних в елементи призначеного для користувача інтерфейсу. Також, функції асинхронного потокового програмування - першокласні функції в Dart. Ці функції широко використовуються в реактивному програмуванні, яке є парадигмою сьогоднішнього дня.

2.4 Flutter

Flutter поєднує в собі простоту розробки з продуктивністю, аналогічній нативній продуктивності, при збереженні візуальної відповідності між платформами. Що найбільш важливе, Flutter повністю безкоштовний і має відкритий код. На даний момент Flutter користується однаковою популярністю з React Native як на GitHub, так і на Stack Overflow [12].

Щоб уникнути проблем з продуктивністю, пов'язаних з використанням скомпільованої мови програмування як міст JavaScript, Flutter використовує Dart. Він заздалегідь компілює Dart (AOT) в власний код для декількох платформ. Таким чином, Flutter може легко взаємодіяти з платформою без необхідності використання моста JavaScript, який включає перемикання контексту між областю JavaScript і власної областю. Компіляція в машинний код також збільшує час запуску програми.

Основна ідея Flutter - використання віджетів. Комбінуючи різні віджети, розробники можуть побудувати весь призначений для користувача інтерфейс. Кожен з цих віджетів визначає структурний елемент (наприклад, кнопку або меню), стилістичний елемент (шрифт або колірну схему), аспект макета (наприклад, відступи) і багато інших. Зверніть увагу, що Flutter не використовує OEM-віджети, а надає розробникам власні готові віджети, які виглядають рідними для додатків Android або iOS (відповідно до Material Design або Cupertino). Звичайно, що розробники також можуть створювати свої власні віджети [12].

StatelessWidget

Різниця між StatefulWidget і StatelessWidget прямо в назві. StatefulWidget відстежує власний внутрішній стан. Віджет StatelessWidget не має внутрішнього стану, який змінюється впродовж існування віджета. Його не хвилює його конфігурація або дані, які він відображає. Йому може бути передана конфігурація від свого батька, або конфігурація може бути визначена у віджеті, але він не може

змінити свою власну конфігурацію. Віджет без стану - незмінний. Зазвичай використовується для статичних, незмінних елементів [9].

StatefulWidget

StatefulWidget має внутрішній стан і може управляти цим станом. Усе віджети з відстежуванням стану мають відповідні об'єкти стану. Його варто використовувати для елементів з анімацією або елементів які будуть змінювати свій вигляд відносно даних [8; 9].

2.5 Порівняння Flutter та React Native

Flutter достатньо новий фреймворк, проте він вже зараз може позмагатися з існуючим найбільш відомим кроссплатформ-рішенням, а саме - React Native. Давайте розглянемо можливості Flutter та React, щоб одразу було зрозуміло на якому етапі знаходиться Flutter і чи дійсно його варто почати використовувати.

2.5.1 Порівняння загальної ефективності Dart і JavaScript

Команда Debian поділилася докладним порівняльним аналізом JS і Dart. Вони протестували кожну мову програмування в 10 різних обчислювальних завданнях і виявили варіанти використання, які ілюструють сильні і слабкі сторони цих мов програмування. Час виконання демонструє продуктивність і споживання пам'яті - ефективність ресурсів.

Таблиця 2.1 – Порівняння Dart та Javascript [1]

Задачі для тестування	Час виконання (секунди)		Споживання пам’яті (KB)	
	JS	Dart	JS	Dart
Генерація цифр Пі з арифметикою довільної точності	6	3	67	150

Управління послідовностями ДНК, використовуючи ті ж прості алгоритми регулярних виразів	5	5	1160	884
Вичислити координати планет в імітованій сонячній системі	9	9	35	125
Прочитати послідовності ДНК і записати їх зворотне значення	2	42	1525	6600
Індексований доступ до дуже маленької цілочисельної послідовності	12	60	63	116
Вирішити набір з 10 завдань чисельного аналізу	2	6	64	124
Зображувати множину Мандельброта і написати вихідний файл в спеціальному форматі	4	12	93	228
Створення і запис випадкових послідовностей ДНК	2	6	71	159
Накопичення лічильників і оновлення їх значення за допомогою хеша	16	24	393	526
Виділити і звільнити багато двійкових дерев	7	10	1310	694

Як видно з наведеної вище таблиці, в 7 з 10 випадків JavaScript виконує задачі швидше при меншому споживанні пам'яті. Таким чином, у порівнянні з Dart, JS є

більш потужним і ресурсо-ефективнішим для десктоп, серверного програмування та інших середовищ.

2.5.2 Мобільна продуктивність

Спільнота Startup на Medium поділилася цікавим дослідженням ефективності нативних і кросплатформних підходів в розробці мобільних додатків. Вони провели кілька тестів на реальних смартфонах Android та iOS і порівняли час виконання розрахунків числа Pi, реалізованих за допомогою фреймворків Flutter, React Native і Native (Swift / Obj-C для Apple, Java / Kotlin для Android). Давайте подивимося на результати в таблиці нижче.

Таблиця 2.2 – Порівняння Flutter, React Native та Нативного коду [1]

Задачі для тесту	IOS (міллісекунди)			Android (міллісекунди)		
	React Native	Flutter	Native	React Native	Flutter	Native
CPU тест із задачею алгоритму Борвейна	582	180	26	822	285	144
Тест з інтенсивним використанням пам’яті із алгоритмом Гаусса-Лежандра	2992	189	173	3289	273	223

Безсумнівно, найбільшою перевагою нативной мобільної розробки є найвища продуктивність в будь-якій задачі. Однак з кроссплатформними фреймворками все не так просто.

Flutter продемонстрував зниження продуктивності всього на 9-22% в тестах з інтенсивним використанням пам'яті в порівнянні з Native. Хоча в ресурсномістких завданнях він в 2-6 разів повільніше, ніж Native.

React Native показав найгіршу продуктивність у всіх тестах. Він в 2-15 разів повільніше, ніж Flutter, і в 5-21 разів повільніше, ніж Native.

Виконання кінцевого коду на React Native вимагає «міст» від коду JavaScript до власного середовища пристрою. Цей міст дозволяє цим двом середовищам зв'язуватися і обмінюватися даними. Простими словами, міст - це інтерпретатор, який переводить код JS на нативну мову програмування пристрою і навпаки. На жаль, міст є додатковим рівнем в цьому системному ланцюжку, який вимагає додаткового часу та ресурсів для обробки.

На відміну від цього, Flutter компілюється у власні бібліотеки x86 і ARM без будь-яких додаткових прошарків. Таким чином, він працює швидше і споживає менше ресурсів для виконання коду. Це призводить до миттєвого запуску додатка, більш високої продуктивності і меншого навантаження на пристрій. Крім того, Flutter має вбудований двигун рендерингу графіки C++, який транслює зображення прямо на екран. Завдяки цьому анімація виконується швидше і плавніше, і потрібно менше коду. Це не означає, що React Native не може обробляти складні алгоритми або складну графіку. Просто Flutter може робити це швидше і ефективніше. Тому нехай Dart і менш швидкий, аніж Javascript, проте в двобої продуктивності між фреймворками, Flutter перемагає.

2.5.3 Веб-продуктивність

React - король розробки веб-сайтів. На ньому написано багато сайтів. JavaScript - один з основних мов інтерфейсу поряд з HTML і CSS. Отже, продуктивність React Native у вебі чудова. Dart не дуже популярний в розробці веб-сайтів. У нього немає гідних переваг у веб-розробці, готовій інфраструктурі і рішеннях в порівнянні з JS, PHP, Java або Ruby. Розробники можуть створити веб-сайт за допомогою Flutter дуже швидко, і він буде працювати, ймовірно, навіть

швидше, ніж мобільний додаток. Однак, якщо продуктивність сайту має вирішальне значення, краще подумати про те, щоб зробити його за допомогою іншої технології, спеціально призначеної для вебу. Таким чином, можливість створити веб-сайт з Flutter - це скоріше корисна функція, яка значно зменшує час виведення на ринок.

2.5.4 Можливості дизайну та графіки

React Native та Flutter мають чудові графічні методи, хоча вони використовують абсолютно різні підходи при рендерингу інтерфейсу. Важка графіка та складні анімації можуть бути створені на обох фреймворках.

React Native має власні візуальні елементи та зовнішній вигляд: кнопки, навігація, меню і т.д. Ці елементи виглядають природньо для користувачів і пропонують хороший “user experience”.

У разі оновлень операційної системи елементи програми будуть відповідно оновлюватися і зберігати відчуття природності і подібності з іншими додатками. Однак слід мати на увазі, що додаток завжди буде виглядати трохи по-різному на пристроях iOS і Android, а також в різних версіях прошивки.

З Flutter додаток буде виглядати однаково незалежно від версії ОС або моделі пристрою. Навігація і елементи залишаться незмінними, якщо розробник-програміст навмисно не змінить їх. Загальний зовнішній вигляд програми більше нагадує гібридні фреймворки (Xamarin, Ionik, Cordova), а не нативні. Приклавши додаткові зусилля, можна досягти персоналізованого нативного зовнішнього вигляду. Однак набагато простіше реалізувати загальний нейтральний стиль для всіх пристроїв і заощадити на обсязі проекту.

2.5.5 Сумісність, функціонал платформ і CI / CD

Flutter підтримує різні права доступу на різних пристроях з Android 4.1+ або iOS 8+. React Native підтримує Android 4.1+ і iOS 10+. У деяких випадках може знадобитися додаткова нативна розробка як для Flutter, так і для React Native зі

складними функціями. Написання нативних частин може вирішити деякі проблеми, але може зменшити переваги використання кроссплатформного рішення в обслуговуванні. Віджети в Flutter і готові бібліотеки в React Native також часто містять ін'єкції нативного коду. Це забезпечує майже нативную продуктивність зі складними функціями і обміном даними з обладнанням пристроїв.

Геолокація і картографія

У Flutter є безліч офіційних плагінів, створених командою Google. React Native добре працює з відстеженням локації, однак при безперервному відстеженні можуть виникнути деякі проблеми, які можна вирішити, написавши деякі частини на нативних мовах програмування. У Flutter з цим проблем немає.

Камера пристрою

У простих сценаріях використання камери, у Flutter не виявили ніяких проблем. У React Native виникають деякі проблеми з використанням камери пристрою. Це вимагає додаткового часу та ресурсів для вирішення і досягнення бажаної продуктивності.

Відеочати і потокова передача

У Flutter, зоб створити для користувача відеочат можна використовувати зовнішній плагін, але для вибору найбільш підходящого потрібні пошукові дослідження. Також є кілька готових платних рішень. З React Native це можна було реалізувати на старті.

Аналітика

Обидві технології підтримують популярні сторонні аналітичні рішення, такі як Google Analytics, Firebase, AppsFlyer і Adjust. У Flutter не було проблем з продуктивністю при використанні інструментів аналітики. У рідких випадках на продуктивність програми React Native може вплинути велика кількість аналітичних подій.

CI / CD

З Flutter можна розгорнути свій додаток за допомогою інтерфейсу командної строки, однак CI / CD для iOS App Store трохи складний і вимагає додаткових

зусиль. React Native не має вбудованого інструменту або офіційної документації з цього приводу, але підтримує багато популярних сторонніх служб CI / CD, такі як Fastlane, Gitlab CI / CD, Github Actions і Microsoft AppCenter.

2.5.6 Висновок

Тож загалом можна сказати, що Flutter точно не поступається місцем і не програє своєму супернику. Навпаки він стрімко рухається в гору до нових досягнень. В цьому йому допомагає професійна команда з Google, що означає, що Flutter в надійних руках і надалі продукт буде тільки вдосконалюватися. Тому вже зараз мобільним розробникам слід задуматися про вивчення Flutter, так як його все частіше використовують різні компанї і сам фреймворк стає кращим [1].

3. ОПИС ПРОЦЕСУ РОЗРОБКИ ПРОЕКТУ

3.1 Огляд аналогів та порівняння

Зараз на просторі мобільних додатків дуже багато різних програм на тему спорту і підтримку здорового образу життя. Існують системи, котрі аналізують раціон та виводять кількість калорій, які людина з‘їла. Є багато додатків, які мають багато записів відео-тренувань від певних тренерів. Також можна знайти програму яка буде таймером тренування, виводити час та контролювати коли слід зупинитися або перейти до наступної вправи. Проте все ж таки аналогами додатку *Champ* є інформативні програми, котрі містять в собі відео записи тренувань та інформацію про заняття. Одними із таких додатків є системи *BetterMe* та *Adidas*. Тепер варто розглянути більш детально кожен з них.

BetterMe - додаток який містить в собі анімаційні відеозаписи тренувань і являється інформаційним ресурсом для користувачів, які шукають вправи для себе.

Окрім цього програма містить в собі календар занять, щоб можна було розуміти скільки днів займаєтеся користувач та який у нього успіх.

Adidas додаток має в собі набір тренувань, які діляться на короткі відеозаписи з обмеженим часом. Тобто додаток налаштований таким чином, що користувач може виконувати вправи в реальному часі, повторюючи вправи за тренером на відео. Також в додатку присутні деякі статті на тему правильного харчування та тренувань. Також є можливість вибирати тренувальні тижні і таким чином користувач матиме розклад тренувань на весь тиждень.

Усі ці додатки мають деякі власні та неповторні особливості, але загалом вони дуже схожі між собою. Перш за все - це базовий, власний набір тренерів додатку, які показують вправи на відео. Загалом два або три тренера ведуть усі тренування які є в додатку. Окрім цього такого виду програми мають в собі обмежену кількість категорій тренувань. Зазвичай категорії в усіх аналогічних додатках є такими: м‘язовий набір, тренування, щоб схуднути та тренування на витривалість. Тобто знайти якісь комплекси вправ або технік для специфічних видів спорту, наприклад футболу або змішаних єдиноборств - зазвичай неможливо. Виходить, що ці аналоги не є повноцінними для усіх спортсменів. Часто більшість із вправ не пасують для тренувань більш професійних спортсменів та також рідко оновлюються.

Також головна задача усіх подібних додатків - надавати вправи для спортивного тренування у вигляді коротких відеозаписів. На початку може здатися, що так краще займатися, ти можеш слідкувати за вправами, в тебе є певний час на розігрів м‘яз та час на виконання самої вправи. Проте перший недолік багатьох таких додатків, що потім стає помітним те, що усі відеозаписи зациклені, тобто тренера на відео не виконують вправи стільки разів скільки вказано в описі тренування. Із за цього зникає відчуття того, що користувач слідкує за тренером в прямому часі. Це недоліки інтерактивності з глядачем. Тобто між користувачем та тренером на відео не має контакту. Це просто перегляд можливих вправ, які можна виконувати самотужки. Також великою проблемою зв‘язку між глядачем та

тренером є відсутність опису завдань самим тренером. Людина, яка показує вправи на відео не пояснює як їх виконувати, замість тренера - це робить текст опису вправи, який з‘являється на відео. Тож слід мати можливість завантажувати повноцінне відео-тренування, де тренер пояснює детально усі вправи так, як гадає за потрібним.

Тому варто створити додаток, в якому перш за все тренером може бути будь-хто та завантажувати ті тренування, які захоче. Це дасть більш обширний асортимент відео-записів та категорій і додаток зможуть використовувати усі можливі спортсмени. Тож *Champ* буде спортивною соціальною мережою. Окрім відео-записів у користувачів буде можливість створювати пости, в яких вони зможуть ділитися цікавою інформацією по своєму виду спорту або просто розказати про свої досягнення. Це допоможе створити внутрішнє спортивне суспільство, де люди зможуть обмінюватися своїм досвідом. Тож основними особливостями додатку *Champ* будуть:

- Створення власного відео-тренування
- Створення власного посту
- Перегляд відео-тренувань
- Перегляд постів
- Можливість коментувати пости та тренування
- Додавання власних тегів для тренувань

У цьому розділі було проведено огляд аналогів додатку *Champ*. Виявлено їх переваги та недоліки, які варто застосувати для створення власного додатку. Також після огляду був створений список основних, головних особливостей додатку *Champ*.

3.2 Архітектура системи

Проект створено на основі безсерверної архітектури. Це означає, що в програмній системі немає шару з внутрішнім сервером і всі обчислення виконуються в обчислювальних сервісах хмар Google та Amazon. В моєму випадку зручно створювати проект без налаштування власних серверів або серверної інфраструктури в якійсь хмарі. Замість цих всіх маніпуляцій достатньо створити аккаунт на AWS та Google Firebase і почати використовувати їх хмарні сервіси.

Додаток клієнту створено на фреймворку Flutter. За допомогою нього є можливість створити програми котрі можуть виконуватися на IOS, Android, WEB та навіть Desktop платформах. Проте звичайно це все залишається клієнтом і ще потрібно створити бекенд частину.

Бекенд частина в системі буде складатися із Lambda функцій AWS та прямих контактів клієнту з сервісами Firebase, використовуючи Firebase SDK. Звичайно не можна викликати Lambda функції напряму, тому потрібен якийсь спосіб звернутися до них. Це можна робити через HTTP запити до API Gateway. За допомогою API Gateway можна створити контакт між клієнтом та Lambda. Це схоже на міст. Створивши ендпоінти для моїх Lambda функцій можу тепер до них звертатися за допомогою пакету Flutter Dio. В якості параметрів передачі буде використовуватися JSON структура. В ній будемо передавати та отримувати дані користувачів. На бекенд частині будемо використовувати дані для обробки та збереження їх в базу даних, а на стороні клієнту будуть відображатися отримані дані. Це стандартна модель і вона добре пасує для більшості програмних систем [10; 14; 15].

В нас є такі основні елементи:

- Клієнт
- Firebase SDK
- API Gateway
- Lambda функції

- DynamoDB

Давайте розглянемо детальніше роботу кожного з елементів.

Клієнт це додаток користувача, який містить в собі можливо клієнтську логіку та створює потрібний інтерфейс користувача. На клієнті можуть зберігатися якісь локальні дані певного користувача, проте, щоб додаток був онлайн - основні дані клієнт отримує від бекенд частини. Клієнт зазвичай різний під різні операційні системи. Це зумовлено тим, що різні ОС потребують власних інструментів для розробки. Проте використовується Flutter і таким чином можна створити клієнт одночасно для декількох ОС. Кодова база мого клієнту може використовуватися для створення додатку і для IOS, і для Android, але є одна проблема з побудуванням додатку для IOS. Щоб створити програму котра буде виконуватися для IOS, необхідно мати MacOS на якій буде налаштовано Xcode. Тільки за допомогою Xcode можна будувати додатки для IOS.

Firebase SDK дозволяє клієнту працювати з сервісами Firebase Storage та Auth безпосередньо напряму без додаткових запитів. Таким чином є можливість зберігати різні файли прямо з клієнту, що спрощує процес взаємодії. Також в проекті використовується авторизація в додатку і для цього теж буду використовувати Firebase SDK. З ним простіше додати авторизацію від різних соціальних мереж. Я використовував авторизацію за допомогою Google. В наш час 99% користувачів в Інтернеті мають google аккаунт, особливо ті хто використовують Android ОС. Тому це буде найкраща можливість авторизації користувача в моєму додатку. Окрім цього авторизація з Google набагато швидша, аніж інші традиційні методи.

API Gateway використовується як міст між Lambda функціями і клієнтом. Я налаштував власний API для кожної функції. Тепер в мене є можливість звертатися до функцій за допомогою HTTP запитів. Насправді у Flutter pub вже додали нову бібліотеку AWS amplify, котра дає можливість використовувати сервіси напряму аналогічно Firebase SDK. Проте там є свої нюанси налаштування і ця бібліотека

достатньо нова і недосконала. Тому вирішив не додавати зайвої залежності і використовувати сервіси AWS через HTTP запити.

Lambda функції. Основа моєї бекенд системи. Тут ховається основна обробка даних користувача та взаємодія з базою даних. Lambda функція дуже зручна річ для починаючих розробників. Тут не потрібно вчити багато зайвих термінів та інструментів для налаштування власної бекенд системи. Вам достатньо створити аккаунт, а після чого створити нову функцію. Для редагування коду функції та її тестування AWS має достатньо зручну консоль. В ній можна створювати потрібні файли вашого коду, переглядати аналітику та створювати тести.

Для написання власної функції можна вибрати такі програмні мови: Java, Go, Javascript, Python, C# та Ruby. Для свого проекту вибрав Python. Тому всі обчислення та взаємодії з даними будуть написані на мові Python. Щоб використовувати сервіси AWS за допомогою Python слід додати таку бібліотеку, як Boto3. Вона дозволить отримувати клієнти всіх потрібних вам сервісів. Я використав бібліотеку Boto3 для того, щоб отримати клієнт DynamoDB для взаємодії з таблицями бази даних. Але це не все, що слід зробити для роботи з сервісами AWS. Окрім бібліотеки з клієнтами, потрібно ще підключити право доступу. Права доступу використовуються для того, щоб обмежити можливі дії функції над сервісами AWS та зробити вашу бекенд систему більш безпечною. Тому також додав потрібні права доступу для взаємодії з DynamoDB із моєї функції.

DynamoDB це NoSQL база даних, котра зберігає усі дані ваших користувачів. В мене система не дуже велика проте все д таки окремі моделі даних існують. Такими моделями є:

- користувач - User
- пост - Post
- тренування - Training
- топік - Topic

В стандартній реляційній моделі мені слід би було створити окремі таблиці для кожної моделі даних. Проте використовується NoSQL база даних і тут існують свої правила створення таблиць. Перш за все потрібно розуміти, що чим менше таблиць у вашій DynamoDB, тим більш вона ефективніша та продуктивніша. Це зумовлено іншою архітектурою БД. Проте як можна вмістити усі дані в одну таблицю, якщо нам потрібні окремі стовпчики для кожної моделі. Це неможливо зробити з реляційною базою даних, проте DynamoDB не реляційна і таким чином можна створювати абсолютно різні елементи в одній вашій таблиці. Дані в таблицях DynamoDB зберігаються у вигляді ключ-значення і подібна структурі JSON. Тому тим хто працював з JSON даними, буде легко зрозуміти DynamoDB. Але все ж таки окрім збереження даних, ми повині якось їх отримувати по певним полям. Для цього ми можемо правильно налаштувати наші первинні ключі та ключі сортування. У нас є чотири основні моделі: користувач, пост, тренування та топік. В цих моделях обов'язково повинне бути поле, котре буде ідентифікатором унікальності кожного окремого екземпляру моделі. В DynamoDB може існувати лише одне поле первинного ключа, тому ми не можемо створювати в одній таблиці такі поля, як "post-Id" та "user-id". Але та сама задача залишається. Нам слід якось вмістити усі наші моделі в одній таблиці і вони повинні мати унікальний ідентифікатор. Це можна зробити достатньо цікавим рішенням - ми просто додаємо до нашої строки ідентифікатора потрібний нам префікс. Наприклад, у нас є користувач і у нього є ідентифікатор "www123", то в таблиці DynamoDB ми будемо зберігати його з ідентифікатором "USER_www123". Потім коли нам потрібно буде дістати користувача з певним ідентифікатором ми можемо зробити перевірку, що ID в таблиці повинен починатися з такої строки "USER_" і містити в собі певний залишок, котрий в нашому випадку є ідентифікатором користувача. Окрім префікса моделі даних, таким чином ми можемо додавати будь-які потрібні нам дані і потім фільтрувати по ним наші дані. Всі ці маніпуляції вигідні нам, бо ми використовуємо первинний ключ нашої таблиці, а це достатньо швидкий процес, але якщо ми

будемо намагатися фільтрувати по іншим полям, то з великою кількістю елементів таблиці це буде займати достатньо багато часу [2; 5; 6].

Загальний вигляд моєї системи:

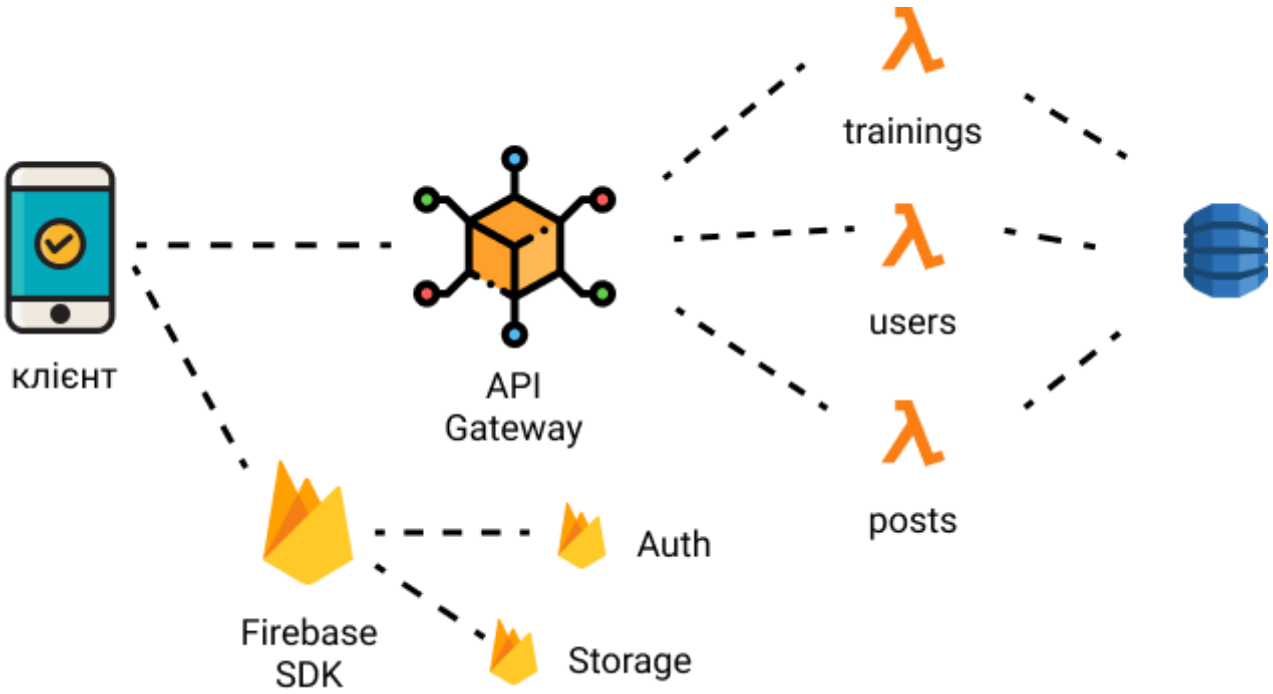


Рисунок 4.1 – Складові частини системи Champ

Загалом це всі елементи моєї системи, котрі дійсно варті уваги. Це основа мого проекту. Всі ці елементи як конструктор в результаті створюють повноцінний онлайн додаток, в якому можна створювати аккаунт, отримувати та додавати пости або тренування і переглядати відео та фото контент. Саме головне, що все це створено без налаштування серврної інфраструктури, тільки за допомогою безсерверних обчислювальних та інших сервісів Amazon і Firebase.

3.3 Архітектура клієнту

Мій клієнтський додаток буде складатися з трьохрівневої архітектури. Ця архітектура зазвичай містить в собі три рівні, котрі ділять додаток на логічні окремі частини і можуть бути незалежними до змін. Перш за все ця архітектура допомагає розробникам логічно відділити та відсортувати різні процеси додатку. Таким чином

створювати структуру проекту і вже простіше шукати та блукати по проекту. Окрім цього великою перевагою такої архітектури є можливість до масштабування додатку, тобто можна накопичувати нові функціональності без зайвих зусиль та потреб рефакторингу. Це спричинено тим, що структура ділиться на окремі модулі, котрі мають власний предметний контекст. За відсутності архітектури, було б складно створювати нові функціональності та розширювати додаток. Можна було б заплуталися в коді додатку та в решті решт застрягли б в нескінченному рефакторингу. Окрім цього наявність гарно продуманої архітектури дозволяє простіше тестувати додаток та відстежувати окремі баги. Не потрібно шукати по всьому коду, що зламалося. Із правильною структурою у є можливість одразу визначити який модуль зламався і де варто шукати помилку.

Архітектура має такі рівні: презентаційний, предметний та рівень даних або репозиторний. Кожен рівень має свої межі і не можна їх переплутати між собою. Рівні використовують інтерфейси для спілкування між собою і зазвичай це виглядає таким чином: презентаційний рівень використовує предметний рівень, а предметний рівень за допомогою інтерфейсу використовує репозиторний рівень [13].

Презентаційний рівень потрібен для того щоб обробляти логіку вашого інтерфейсу користувача. Логіка інтерфейсу може бути такою: натискання кнопки, перехід в інше вікно, зміна кольору при певних умовах, перехід в темний режим і т.д. Це не є логікою додатку, а просто логіка обробки дій користувача з вашим графічним інтерфейсом. Також презентаційний шар використовується для того, зоб взяти якісь дані безпосередньо від користувача. Наприклад користувач написав свій логін для того, щоб увійти у свій аккаунт і натиснув на кнопку «логін». Після цього ми повинні якось передати той логін що написав користувач. Ми знаємо, що з графічним інтерфейсом працює презентаційний рівень і саме він буде збирати дані від користувачів, і передавати далі по системі.

В моєму випадку презентаційний шар буде включати в себе код екранів мого додатку, окремі віджети котрі використовуються декілька разів в певних екранах,

та контролери інтерфейсу. Контролери інтерфейсу будуть відповідати за оновлення стану графічного інтерфейсу, збирати дані з інтерфейсу та передавати їх далі по системі в інші рівні. Ми вже знаємо, що розробка на Flutter включає в собі такий процес, як менеджмент станом інтерфейсу. Для цього зазвичай і існують контролери, але як ми вже дізналися Flutter має різні можливості управлінням стану. Я буду використовувати для цього таку бібліотеку як GetX. Вона дуже легка у використанні і використовує набагато менше коду для управління станом, аніж інші бібліотеки [7].

Після того як ми отримали дані від інтерфейсу користувача нам потрібно передавати їх далі по системі. Наступний рівень котрий отримує дані - предметний. На цьому рівні зазвичай виконується різна бізнес логіка додатку. В мене додаток достатньо простий і на цьому рівні ніякої додаткової логіки не має. Зазвичай для сучасних простих соціальних додатків, котрі достатньо схожі між собою, додаткової логіки на клієнті багато не потрібно, тому часто предметний шар в ваших додатках буде просто посередником між презентаційним та репозиторним рівнем. В моєму клієнті так само. Вся загальна логіка буде знаходитися на бекенд частині. Проте деякі задачі будуть виконуватися не на бекенді, а з клієнту за допомогою потрібних SDK. В додатку Champ такими задачами будуть:

- Збереження медіа в Firebase Storage
- Створення токену авторизації за допомогою Firebase Auth

Інші задачі виконуються у обчислювальних системах Lambda. Для того щоб використовувати ці сервіси, щоб якось до низ звернутися нам потрібно перейти до іншого рівня - репозиторного. Предметний рівень передає дані взяті із презентаційного рівня до репозиторного. Репозиторний рівень відповідає за виконання HTTP запитів та створення Dart об‘єктів із відповіді бекенду. Мій проект не має внутрішнього серверу котрий приймає HTTP запити, але мені потрібно якось звертатися до Lambda функцій. Для цього налаштував власний API Gateway, котрий має ендпоінти до яких прикріплені Lambda функції.

На репозиторному рівні створено класи котрі виконують HTTP запити до API Gateway за допомогою бібліотеки Dio. Dio - це бібліотека, котра дозволяє детально налаштувати запити до бекенду. Є можливість додати потрібні HTTP заголовки або створити кешування ваших запитів, щоб менше турбувати вашу бекенд систему.

Окрім взаємодії з бекенд системою репозиторний рівень має класи-мапери. Класи-мапери - це класи які дозволяють перетворювати JSON значення на об‘єкти Dart. Такі класи необхідні, тому що зазвичай у відповіді від бекенду буде JSON і для простішої взаємодії на клієнті слід зберігати дані в об‘єктах Dart.

Тепер давайте розглянемо структуру архітектури клієнту. Перший рівень презентаційний містить у собі такі елементи:

- Віджети. Окремі маленькі частини інтерфейсу, котрі використовуються в екранах.
- Окремі екрани інтерфейсу. Основні частини графічного інтерфейсу, котрі містять в собі інші віджети.
- Контролери. Відповідають за логіку інтерфейсів та отримання даних безпосередньо від графічного інтерфейсу.
- Класи-зв‘язувачі. Класи котрі міксують в собі залежності класів контролерів та інших шарів.

Другий рівень - предметний. Містить в собі такі елементи:

- Usecase-класи. Відповідають за бізнес логіку клієнту. Якщо існує бізнес логіка, то вона міститься саме в цих класах.
- Інтерфейс репозиторію. Власне інтерфейс для взаємодії з реалізацією репозиторію із рівня даних.
- Entity-класи. Моделі даних додатку. Це класи, котрі будуть створюватися із JSON даних та використовуватися для відображення цих даних.

Третій рівень - репозиторний/рівень даних. В ньому є такі елементи:

- Реалізація інтерфейсу репозиторію. Це клас, котрий використовує інтерфейс предметного шару та реалізує його.
- Класи-мапери. Класи котрі перетворюють JSON дані на об‘єкти Dart.

- Datasource-класи. Це класи, які використовуються для повернення даних із серверної частини або для того, щоб зробити макет даних із бекенду.

4. ОГЛЯД РОЗРОБЛЕНОГО ДОДАТКУ

4.1 Авторизація

Авторизацію в додатку намагався мінімізувати, так як по дослідженням UX-експертів можна побачити, що багато користувачів покидають огляд додатка із за складної регістрації [10]. Тобто для того, щоб більше користувачів почали використовувати ваш додаток, слід спростити процес регістрації до мінімуму. Тому в додатку зроблено авторизацію не через окрему сторінку, а через не велике діалогове вікно знизу додатка. Виглядає це так:

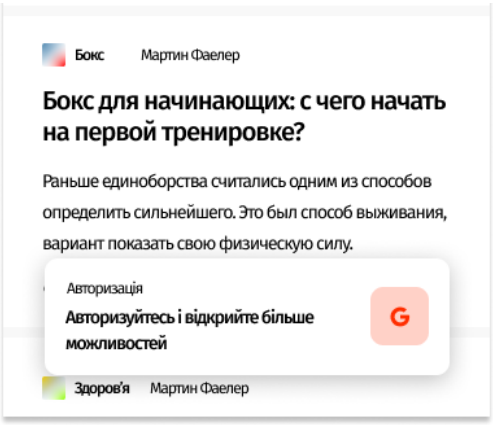


Рисунок 5.1 – Вікно авторизації

Після того як користувач авторизувався, у нього є можливість перейти до свого профілю та також додалися функції зберігання та вподобання постів або тренувань. Профіль додатку виглядає таким чином:

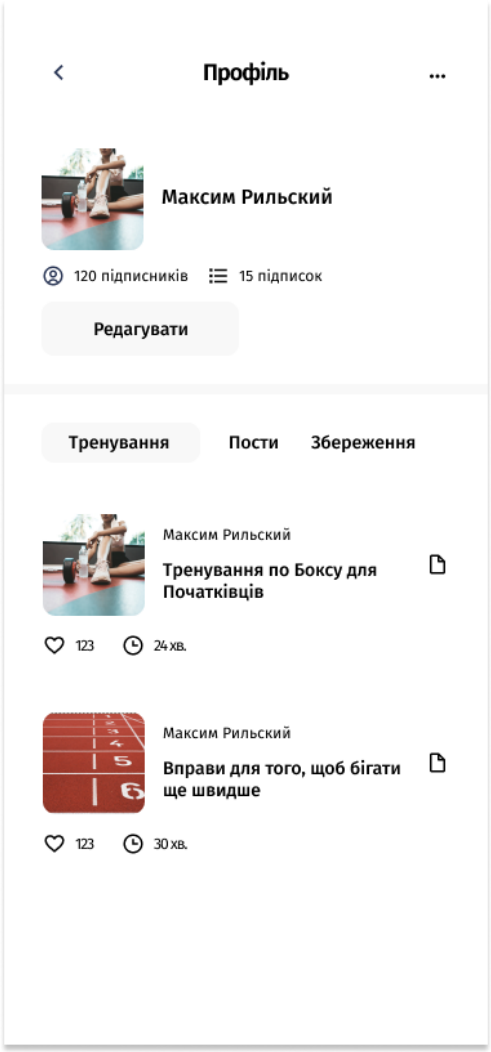


Рисунок 5.2 – Профіль користувача

Тут він побачить усі свої дані, якщо це його профіль або дані іншого користувача. Також зможе підписатися на іншого користувача.

4.2 Перегляд постів

Переглядати пости можна одразу зайшовши до додатку. Тут користувача зустрічає сторінка «моя стрічка». В ній можна побачити усі пости відносно топіків користувача або якщо ж він не авторизований, то тут будуть відображатися пости з усіх рекомендованих топіків. На цій сторінці можна побачити заголовки та короткі описи постів та також у користувача є можливість поставити вподобання або зберегти пост на потім. «Моя стрічка» виглядає таким чином:

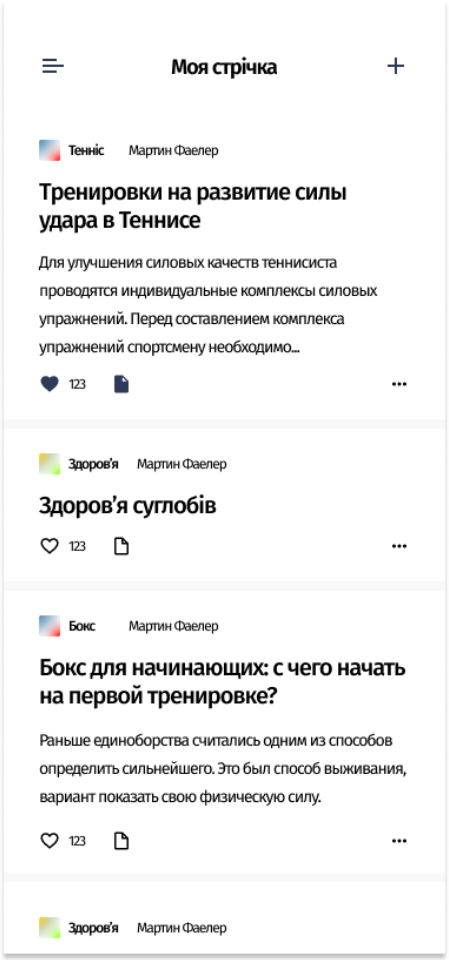


Рисунок 5.3 – Сторінка всіх постів

Окрім цього, якщо натиснути на область посту, то буде можливість перейти до сторінки яка відображає повністю пост з його складовою частиною. Детальна сторінка посту виглядає так:

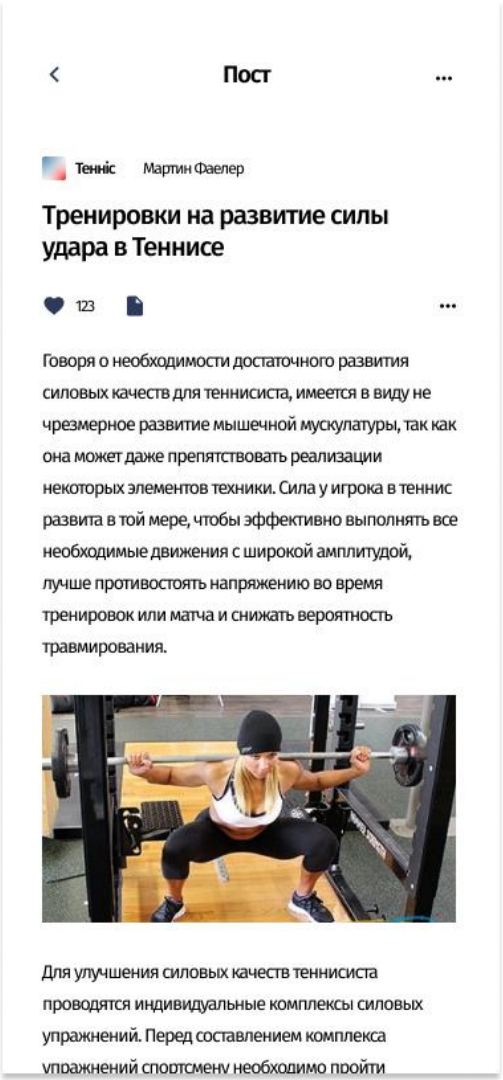


Рисунок 5.4 – Сторінка певного посту

На цій сторінці також можна поставити вподобання або зберегти пост та також перечитати його коментарі.

4.3 Створення посту

Щоб створити пост можна натиснути на плюсик в верхній правій частині сторінки «моя стрічка» або зайти в меню та вибрати пункт створити. Після цього користувач перейде в екран редактора, в якому він може писати потрібний текст. Окрім написання просто тексту він може додавати посилання та фото контент, також можна додати різні заголовки. Ще якщо користувач захоче, щоб пост

відображався у якомусь певному топіку, то можна вибрати топік. Екран створення посту виглядає так:

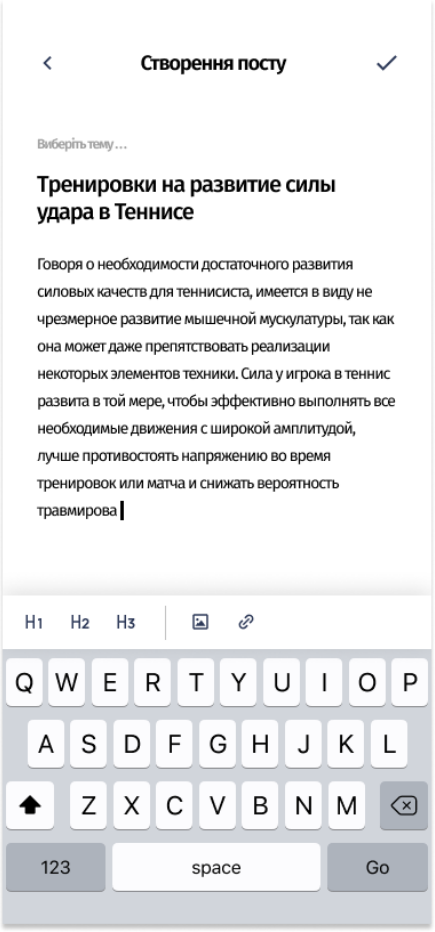


Рисунок 5.5 – Сторінка створення посту

Коли користувач вибирає тему, то буде відкриватися таке вікно [11]:

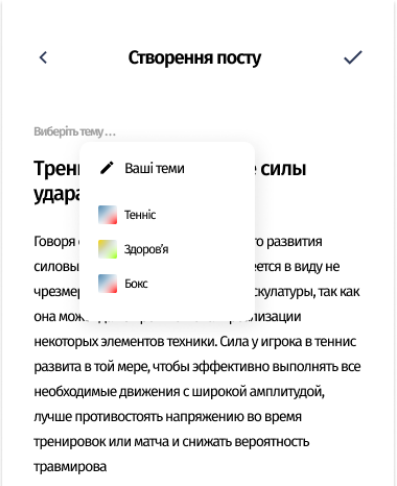


Рисунок 5.6 – Вибір теми посту

В списку топіків будуть тільки ті, на котрі він підписаний. Після закінчення процесу написання посту можна натиснути на кнопку у верхньому правому кутку для створення посту. На цьому етапі пост збережеться в базі даних і його зможуть переглядати інші користувачі.

4.4 Перегляд тренувань

Переглядати тренування можна, якщо перейдете в меню та натиснути на пункт «тренування». Тут можна побачити усі тренування, проте першими будуть йти ті котрі сходяться із вподобаними тегами користувача. Екран всіх тренувань виглядає так:

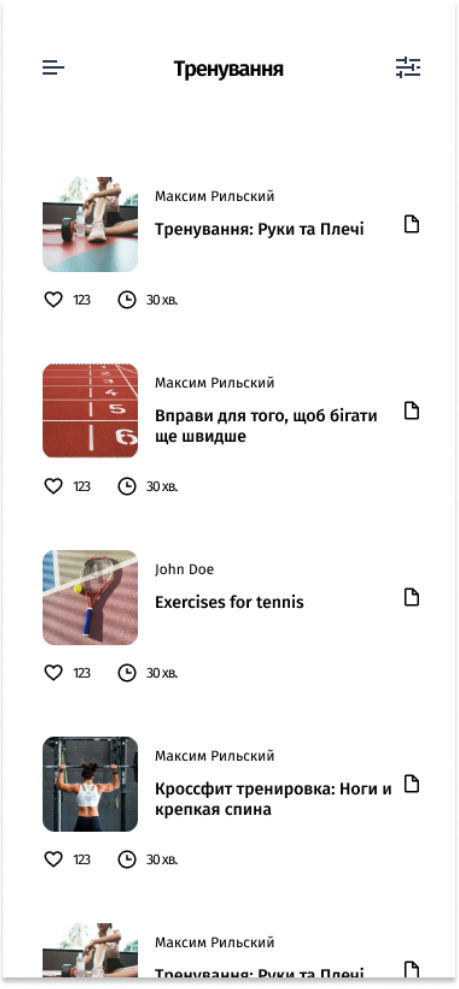


Рисунок 5.7 – Сторінка всіх тренувань

Якщо сподобався заголовок певного тренування і користувач хоче його передивитися, то для цього варто натиснути на саме тренування і перейти на сторінку певного тренування. Дизайн сторінки певного тренування:

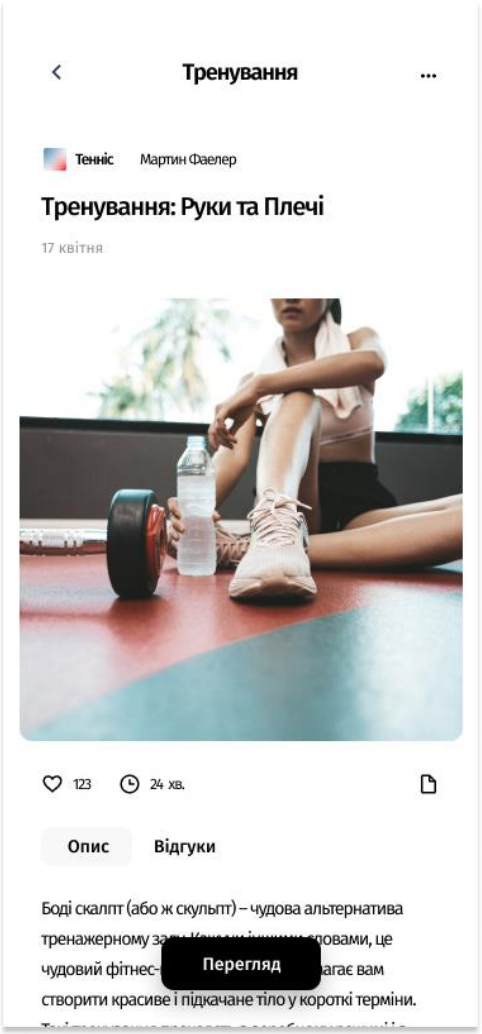


Рисунок 5.8 – Сторінка певного тренування

Тут він побачить опис та відгуки тренування і також зможе передивитися його. Для цього достатньо натиснути на кнопку «Перегляд».

4.5 Створення тренування

Щоб створити тренування потрібно перейти в меню та вибрати пункт створити. У вікні яке з‘явиться слід вибрати пункт «створити тренування».

Сам екран має такий вигляд:

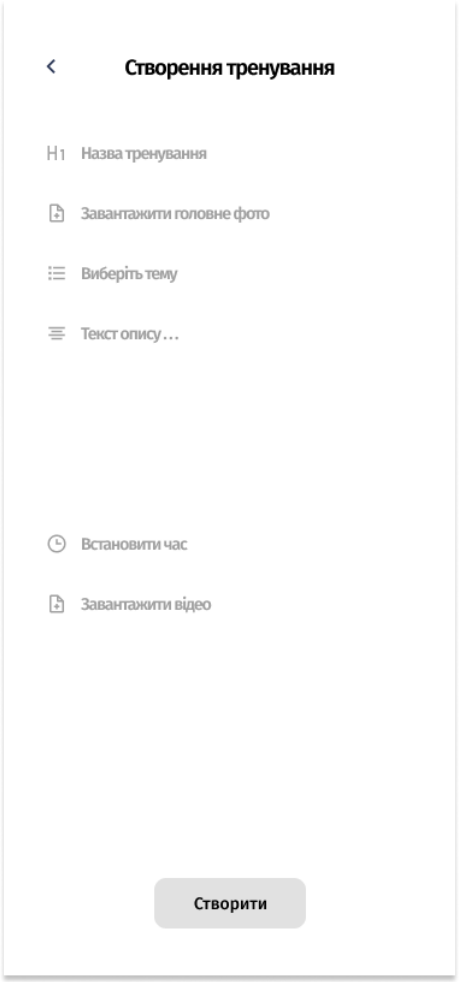


Рисунок 5.9 – Створення тренування

На сторінці створення можна побачити основні пункти, котрі потрібні для створення нового тренування. Є такі пункти:

- Назва
- Завантажити фото
- Завантажити відео
- Додати опис
- Встановити час

Після того як все це додати, кнопка створити активується і можна буде додати нове тренування.

ВИСНОВКИ

У даній роботі було розроблено проект, використовуючи фреймворк для кроссплатформної розробки Flutter у якості інструменту розробки клієнта та хмарні обчислювальні сервіси Amazon у якості бекенд системи.

Flutter використовувався для того, щоб була можливість створити додаток для декількох платформ, таких як Android та IOS. Для цього варто створити стандартний Flutter проект і налаштувати усі залежності вашого проекту під певні платформи. Наприклад плагін локальних та пуш повідомлень в Android та IOS виконується по різному і щоб мати можливість виконувати Flutter код для повідомлень слід на нативній стороні платформ прописати потрібний код. Так з багатьма залежностями, котрі виконуються по різному на цих платформах.

З розробкою інтерфейсу користувача все набагато простіше. Можна писати код без додаткового налаштування і він буде виконуватися під різні платформи. Проте якщо хочеться налаштувати різний вигляд певного додатку на різних платформах можна використовувати перевірку платформ. Логіка достатньо проста: якщо Android - вивести інтерфейс для Android, якщо IOS, то вивести інший інтерфейс. Також є багато способів реалізувати адаптивний інтерфейс. Можна зробити додаток, котрий буде перестроюватися під кожен екран по певній схемі. Також можна налаштувати адаптивність у розмірах. Наприклад в дизайні кнопка має ширину 250 пікселів, то на різних пристроях ця кнопка буде одного розміру - 250. Таким чином інколи на деяких девайсах кнопка може виходити за рамки екрану, чи зачіпати інші елементи інтерфейсу. З певною бібліотекою у Flutter можна зробити так, що ця кнопка буде адаптивно змінювати свою ширину, щоб виглядати гармонічно, з правильним масштабом.

Щодо логічних елементів додатку, то Flutter має достатньо бібліотек вже зараз, щоб реалізувати безпечний, сучасний, онлайн додаток, котрий можна розгортати в різних маркетах. Flutter має чудовий Firebase SDK, котрий можна використовувати для різних сервісів Firebase. Таким чином було налаштовано

авторизацію додатку та створена можливість зберігати медіа-файли. Окрім цього додаток отримав бібліотеку Dio, котра дозволяє створювати запити до бекенду і дає багато можливостей налаштувань запитів. Якщо потрібна оптимізація, можна використовувати ізоляти Dart. Це реалізація багатопоточності в Dart. З ними можна менше використовувати ресурсів головного потоку і тому додаток буде більш швидшим та оптимізованим. Загалом додаткові ізоляти варто використовувати при складних обчисленнях або при запитах до бекенду. В Champ було використано ізоляти для запитів до AWS.

На бекенді використовувались безсерверні рішення Amazon. Створено власні функції, котрі виконують код написаний на Python та використовують DynamoDB. Так тепер додаток може зберігати дані користувачів та стає повноцінним онлайн додатком.

Загалом Flutter достатньо зручний фреймворк і щоб почати його використовувати не треба багато зусиль. Якщо є досвід хоч якоїсь мобільної розробки, особливо комерційної, то швидко можна перейти на Flutter. Він достатньо простий і високорівневий. Тобто не потрібно буде багато втручатися в складні налаштування платформ. У розробника з’являється можливість сконцентруватися на розробці додатку без зайвих відволікань. Звісно Flutter потребує деяких налаштувань під різні платформи, проте це не займає занадто багато часу і можна зробити це на першому етапі розробки без подальшої маніпуляції платформами.

Самим найвдалішим суперником Flutter є React Native. React Native достатньо довго на ринку, але так і продовжує помірно популяризуватися та розвиватися. Flutter розвивається та удосконалюється набагато швидше. Він вже зараз стає кращим рішенням у кроссплатформній розробці хоча він набагато молодший за React Native. Тому очевидно, що у 2021 році слід переходити на Flutter. Можна бути впевненими у тому, що цей фреймворк буде нарощувати функціонал і ставати кращим. Вже в цьому році Flutter команда анонсувала підтримку веб клієнту та

створення десктопних клієнтів. Тому Flutter - це дійсно майбутнє кроссплатформної розробки.

Щодо безсерверної розробки можна сказати, що це остаточно чудове рішення для розробників, котрі хочуть створити власні додатки і не мають багато можливостей та часу для повноцінної серверної розробки. Є можливість створити повноцінний бекенд за малий час і бути впевненим у безпеці свого бекенду, використовуючи сервіси Amazon, Google або Microsoft. Проте це не означає, що крупним підприємствам не варто використовувати хмарні сервіси. Навпаки, якщо немає ідеї створити власний дата центр і тримати свої сервери, то безсерверні рішення можуть допомогти. Проте слід розуміти, що чим більше використовуєш сервіси, тим більше платиш, тому варто бути усвідомленим при налаштуванні сервісів та аналізувати постійно як часто ви їх використовуєте.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. React Native vs. Flutter: What is Better for App Development in 2021 [Електронний ресурс] – Режим доступу: https://nix-united.com/blog/flutter-vs-react-native/

2. NoSQL Data modeling techniques [Електронний ресурс] – Режим доступу: https://highlyscalable.wordpress.com/2012/03/01/nosql-data-modeling-techniques/

3. S. Teller, Serverless Handbook for Frontend Engineers, 2021.

4. M. Stigler, Beginning Serverless Computing, 2018.

5. Особенности проектирования модели данных для NoSQL [Електронний ресурс] – Режим доступу: https://habr.com/ru/post/504166/

6. NoSQL базы данных: понимаем суть [Електронний ресурс] – Режим доступу: https://habr.com/ru/post/152477/

7. GetX. State management [Електронний ресурс] – Режим доступу: https://github.com/jonataslaw/getx/blob/master/documentation/en_US/state_management.md#conditions-to-rebuild

8. E. Windmill, Flutter in Action, 2020.

9. A. Biessek, Flutter for Beginners, 2019.

10. Как сделать идеальное мобильное приложение для интернет-магазина [Електронний ресурс] – Режим доступу: https://vc.ru/dev/95855-kak-sdelat-idealnoe-mobilnoe-prilozhenie-dlya-internet-magazina

11. PopupMenuButton in Flutter [Електронний ресурс] – Режим доступу: https://medium.com/flutterdevs/popupmenubutton-in-flutter-bde21c708018

12. What is Flutter and Why Use Flutter for App Development [Електронний ресурс] – Режим доступу: https://nix-united.com/blog/the-pros-and-cons-of-flutter-in-mobile-application-development/

13. The Comprehensive Guide to Application Architecture in Mobile Development [Електронний ресурс] – Режим доступу: https://nix-united.com/blog/the-comprehensive-guide-to-mobile-application-architecture/

14. M. Fuller, A Guide to Serverless Microservices, 2020.

15. Peter Sbarski, Yan Cui, Ajay Nair, Serverless Architectures on AWS, Second Edition MEAP V06, 2020.

Додаток А

Код usecase'у для модулю постів

```
import 'dart:convert';

import 'package:champ/core/error/failure.dart';
import 'package:champ/core/util/storage/local_storage.dart';
import 'package:champ/modules/post_feed/domain/entities/post.dart';
import 'package:champ/modules/post_feed/domain/entities/post_element.dart';
import 'package:champ/modules/post_feed/domain/entities/post_page.dart';
import 'package:champ/modules/post_feed/domain/entities/topic.dart';
import 'package:champ/modules/post_feed/domain/repositories/post_repository.dart';
import 'package:dartz/dartz.dart';
import 'package:flutter/foundation.dart';

class PostUsecase {
  final PostRepository _postRepository;
  final LocalStorage _localStorage;

  PostUsecase({
    @required PostRepository postRepository,
    @required LocalStorage localStorage,
  })
  : _postRepository = postRepository,
    _localStorage = localStorage,
    super();

  Future<Either<Failure, PostPage>> getPostsByUserTopics(List<TopicInfo> topics,
String lastEvaluatedKey) async {
    return _postRepository.getPostPageByTopics(topics, lastEvaluatedKey);
  }

  Future<Either<Failure, Post>> createPost(Post post, List<PostElementFile>
postElementFiles) async {
    postElementFiles.forEach((pef) async {
      final uploadResult = await _postRepository.uploadFile(pef.file);
      uploadResult.fold(
        (failure) => Left(failure),
        (url) {
          post.postElements
          .firstWhere((pe) => pe.id == pef.postElementId)
```

```
          .data = url;
        }
      );
    });

    return _postRepository.createPost(post);
  }

  Future<Either<Failure, int>> createPostLike(String postId, String userId, int
likesCount, Map<String, String> likedPostsIds) async {
    _localStorage.saveLikedPosts(userId, json.encode(likedPostsIds));
    final likeId = likedPostsIds[postId];
    return _postRepository.createPostLike(postId, userId, likesCount, likeId);
  }

  Future<Either<Failure, int>> deletePostLike(String postId, String userId, int
likesCount, String likeId) async {
    return _postRepository.deletePostLike(postId, userId, likesCount, likeId);
  }

  Future<Map<String, String>> getLikedPostsByUserId(String userId) async {
    return json.decode(_localStorage.fetchLikedPosts(userId));
  }
}
```

Код контролера інтерфейсу для модулю постів

```
import 'package:champ/core/error/failure.dart';
import 'package:champ/modules/auth/domain/entities/user.dart';
import 'package:champ/modules/auth/presentation/controllers/auth_controller.dart';
import 'package:champ/modules/post_feed/domain/entities/post.dart';
import 'package:champ/modules/post_feed/domain/entities/topic.dart';
import 'package:champ/modules/post_feed/domain/usecases/post_usecase.dart';
import 'package:flutter_screenutil/flutter_screenutil.dart';
import 'package:get/get.dart';
import 'package:pull_to_refresh/pull_to_refresh.dart';

class FeedController extends GetxController {
  final PostUsecase _postUsecase;

  FeedController(this._postUsecase);
```

```
  final screenUtil = ScreenUtil();
  final authController = Get.find<AuthController>();
  final refreshController = RefreshController(initialRefresh: true);
  final lastPage = Rx<bool>(false);
  final currentLastEvaluatedKey = Rx<String>();
  final userTopics = Rx<List<TopicInfo>>([]);
  final posts = RxList<Post>([]);

  @override
  void onInit() {
    ever(authController.currentUser, initFeed);
    print('FeedController onInit');
    super.onInit();
  }

  Future<void> fetchPostPage() async {
    if (!lastPage.value) {
      final result = await _postUsecase.getPostsByUserTopics(userTopics.value,
currentLastEvaluatedKey.value);
      print('FeedController: fetch post page');
      result.fold(
        (failure) {
          print('FeedController: result: Failure');
          showErrorSnackbar(failure);
        },
        (postPage) {
          print('FeedController: result: Success');
          if (currentLastEvaluatedKey.value == null) {
            posts.clear();
          }
          if (postPage.lastEvaluatedKey == null) {
            lastPage.value = true;
          }
          currentLastEvaluatedKey.value = postPage.lastEvaluatedKey;
          posts.addAll(postPage.items);
        }
      );
    }
  }

  void initFeed(User user) {
```

```
    currentLastEvaluatedKey.value = null;
    if (user == null) {
      print('FeedController: user == null');
      userTopics.value = [];
    } else {
      print('FeedController: user != null');
      userTopics.value = user.topics;
    }
  }

  Future<void> onRefreshFeed() async {
    currentLastEvaluatedKey.value = null;
    lastPage.value = false;
    await fetchPostPage();
    refreshController.refreshCompleted();
  }

  Future<void> onLoadNewPage() async {
    await fetchPostPage();
    refreshController.loadComplete();
  }
```

Код інтерфейсу «Моя стрічка»

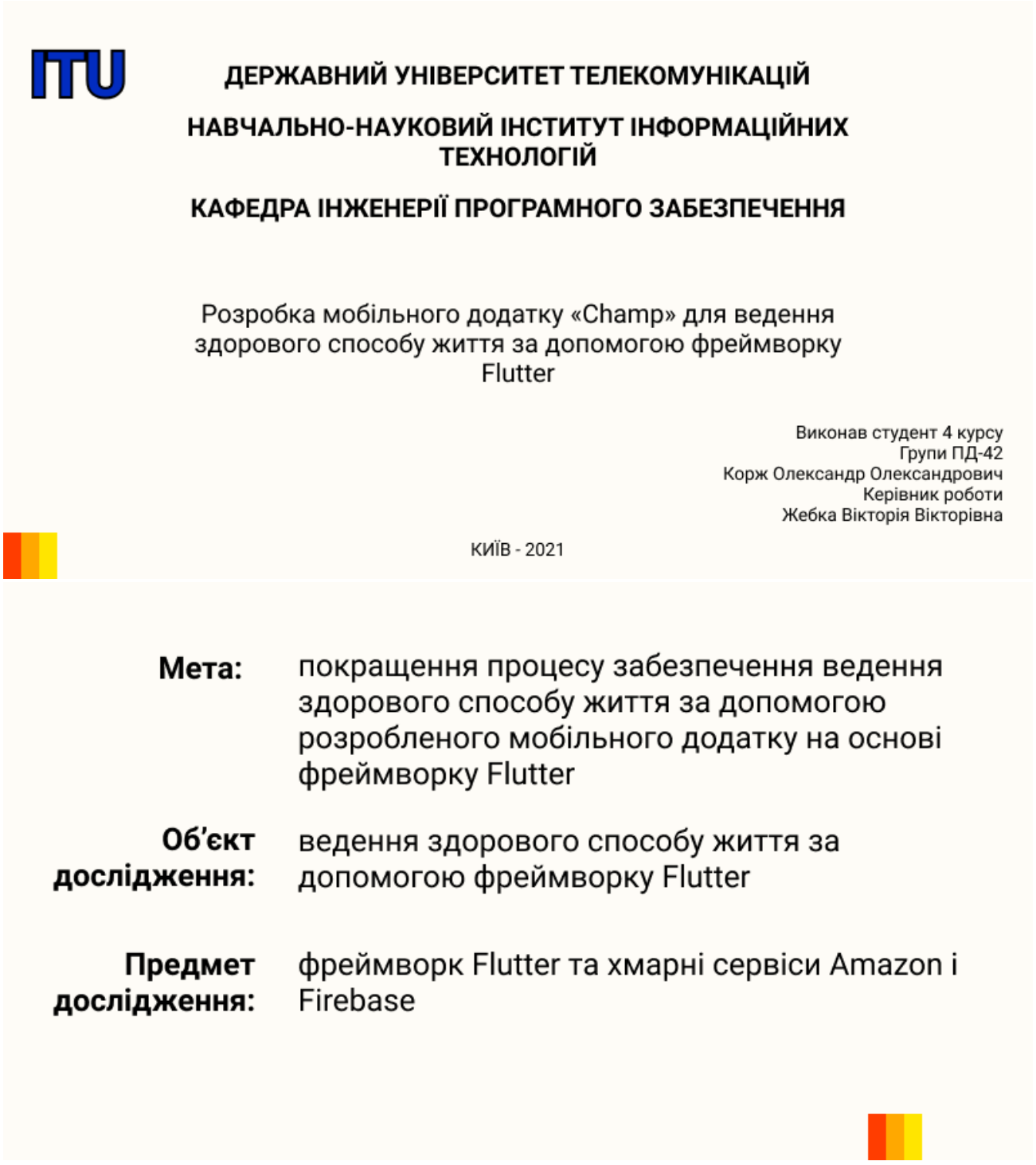
```
import 'package:champ/modules/home/home_controller.dart';
import 'package:champ/core/util/appColors.dart';
import 'package:champ/core/util/no_glow_scroll_behavior.dart';
import 'package:champ/core/util/textStyles.dart';
import
'package:champ/modules/post_feed/presentation/controllers/feed_controller.dart';
import 'package:champ/modules/post_feed/presentation/widgets/post_list_item.dart';
import 'package:flutter/cupertino.dart';
import 'package:flutter/material.dart';
import 'package:get/get.dart';
import 'package:pull_to_refresh/pull_to_refresh.dart';


class FeedPage extends StatelessWidget {
  final FeedController feedController = Get.find();
  final HomeController homeController = Get.find();
```

```
@override
Widget build(BuildContext context) {
 return Scaffold(
  backgroundColor: AppColors.white,
  body: Obx(() => ScrollConfiguration(
   behavior: NoGlowScrollBehavior(),
   child: SmartRefresher(
    controller: feedController.refreshController,
    onRefresh: feedController.onRefreshFeed,
    onLoading: feedController.onLoadNewPage,
    enablePullDown: true,
    enablePullUp: true,
    footer: getCustomFooter(),
    header: getCustomHeader(),
    child: ListView.builder(
     itemBuilder: (context, index) {
      final post = feedController.posts[index];
      final notLast = index < feedController.posts.length;
      return Column(
       children: [
        PostListItem(post: post),
        notLast ?
         Obx(() => AnimatedContainer(
          duration: Duration(milliseconds: 300),
          height: feedController.screenUtil.setHeight(8),
          margin: homeController.isCollapsed.value ?
           EdgeInsets.only(left: feedController.screenUtil.setWidth(30))
           :
           EdgeInsets.zero,
          color: AppColors.grey,
         ))
        :
         Container()
       ],
      );
     },
     itemCount: feedController.posts.length,
    ),
   ),
  )),
 );
```

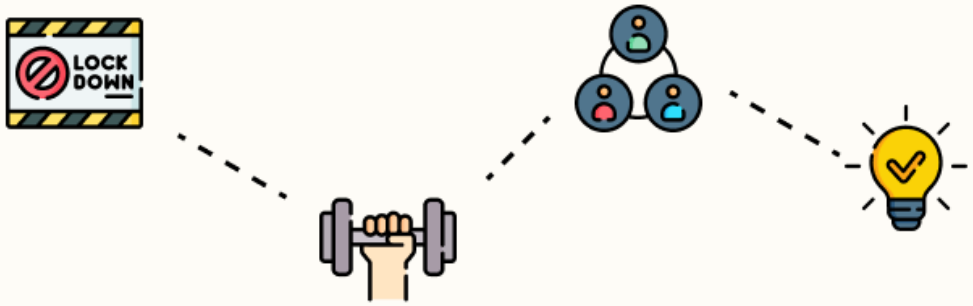
```
}

Widget getCustomHeader() {
  return CustomHeader(
    refreshStyle: RefreshStyle.Follow,
    builder: (BuildContext context, RefreshStatus status){
      Widget text;
      switch (status) {
        case RefreshStatus.canRefresh:
        case RefreshStatus.idle:
          text =  Text('Update', style: TextStyles.weight400px14());
          break;
        case RefreshStatus.refreshing:
          text = Text('Loading', style: TextStyles.weight400px14());
          break;
        case RefreshStatus.completed:
          text =  Text('Completed', style: TextStyles.weight400px14());
          break;
        default:
          text = Text('Error', style: TextStyles.weight400px14());
      }

      return Container(
        height: 55.0,
        child: Center(
          child: Row(
            mainAxisAlignment: MainAxisAlignment.center,
            children: [
              CupertinoActivityIndicator(),
              SizedBox(width: 10),
              text
            ],
          )
        ),
      );
    },
  );
}

Widget getCustomFooter() {
  return CustomFooter(
```

```
      loadStyle: LoadStyle.ShowWhenLoading,
      builder: (BuildContext context, LoadStatus status){
        Widget text;
        switch (status) {
          case LoadStatus.canLoading:
          case LoadStatus.idle:
            text =  Text('Loading', style: TextStyles.weight400px14());
            break;
          case LoadStatus.noMore:
            text =  Text('End of Page.(', style: TextStyles.weight400px14());
            break;
          default:
            text = Text('Error', style: TextStyles.weight400px14());
        }

        return Container(
          height: 55.0,
          child: Center(
            child: Row(
              mainAxisAlignment: MainAxisAlignment.center,
              children: [
                CupertinoActivityIndicator(),
                SizedBox(width: 10),
                text
              ],
            )
          ),
        );
      },
    );
  }
}
```

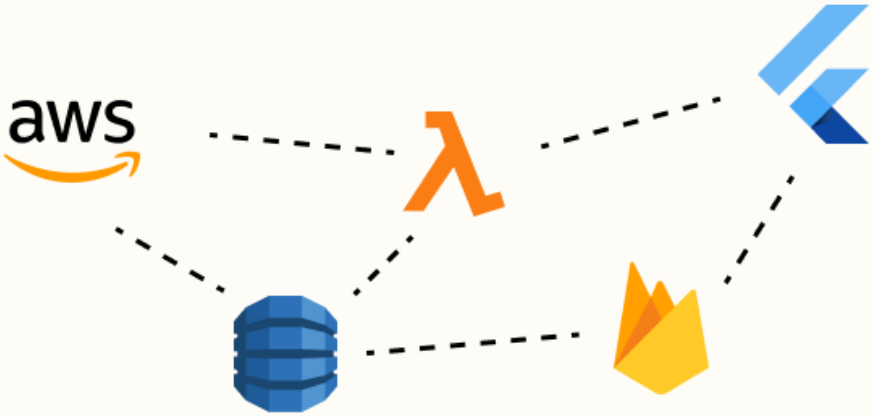
Додаток Б


ДЕРЖАВНИЙ УНІВЕРСИТЕТ ТЕЛЕКОМУНІКАЦІЙ
НАВЧАЛЬНО-НАУКОВИЙ ІНСТИТУТ ІНФОРМАЦІЙНИХ ТЕХНОЛОГІЙ
КАФЕДРА ІНЖЕНЕРІЇ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ
Розробка мобільного додатку «Champ» для ведення здорового способу життя за допомогою фреймворку Flutter
Виконав студент 4 курсу
Групи ПД-42
Корж Олександр Олександрович
Керівник роботи
Жебка Вікторія Вікторівна
КИЇВ - 2021
Мета:
покращення процесу забезпечення ведення здорового способу життя за допомогою розробленого мобільного додатку на основі фреймворку Flutter
Об'єкт дослідження:
ведення здорового способу життя за допомогою фреймворку Flutter
Предмет дослідження:
фреймворк Flutter та хмарні сервіси Amazon і Firebase

Актуальність спорту в онлайн середовищі



Актуальність дослідження технологій проекту



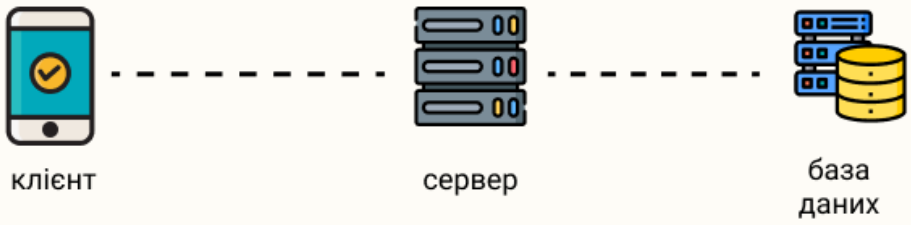
Що таке Flutter?

Компанія Google представила Flutter як технологію з відкритим кодом для створення власних додатків для Android і iOS.

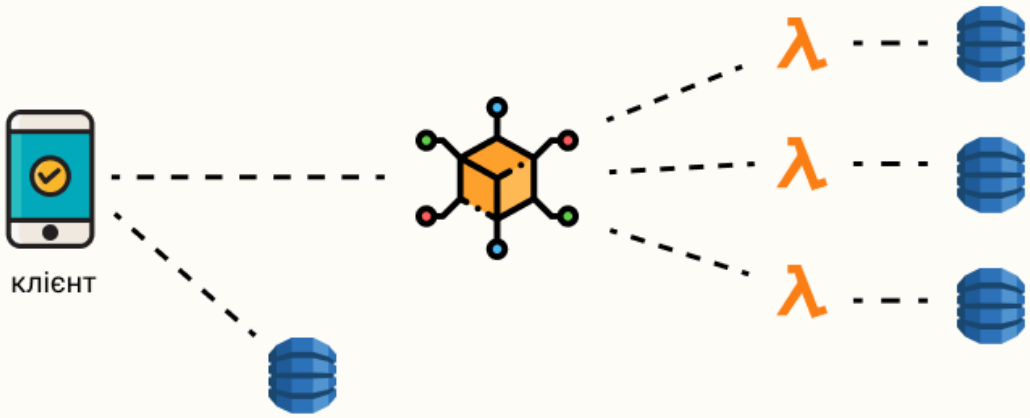
Flutter поєднує в собі простоту розробки з продуктивністю, аналогічній нативній продуктивності, при збереженні візуальної відповідності між платформами.

Flutter vs React Native vs Native

Традиційний бекенд



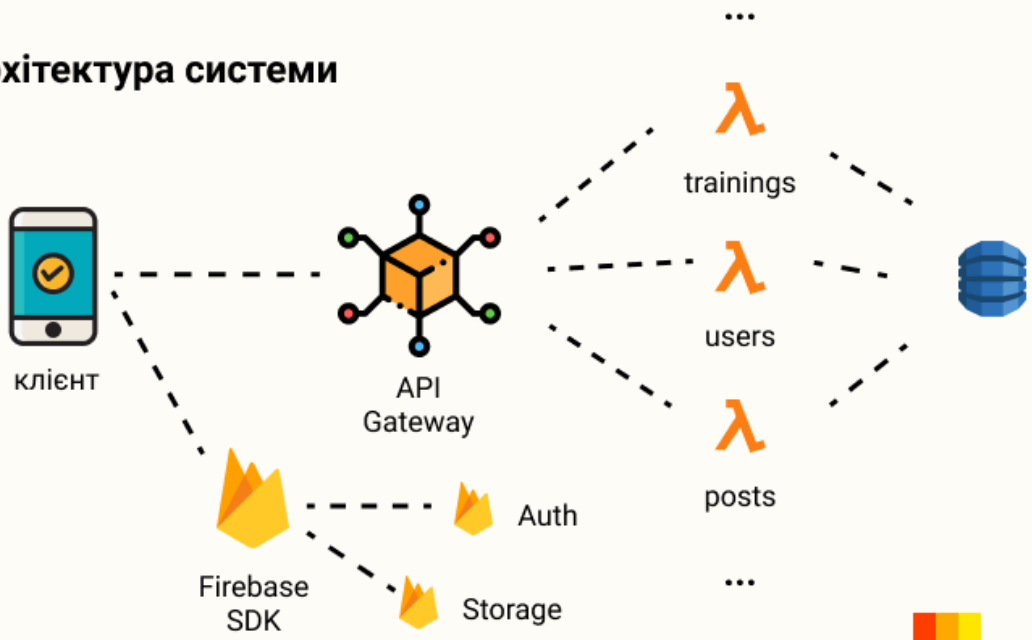
Безсерверна архітектура

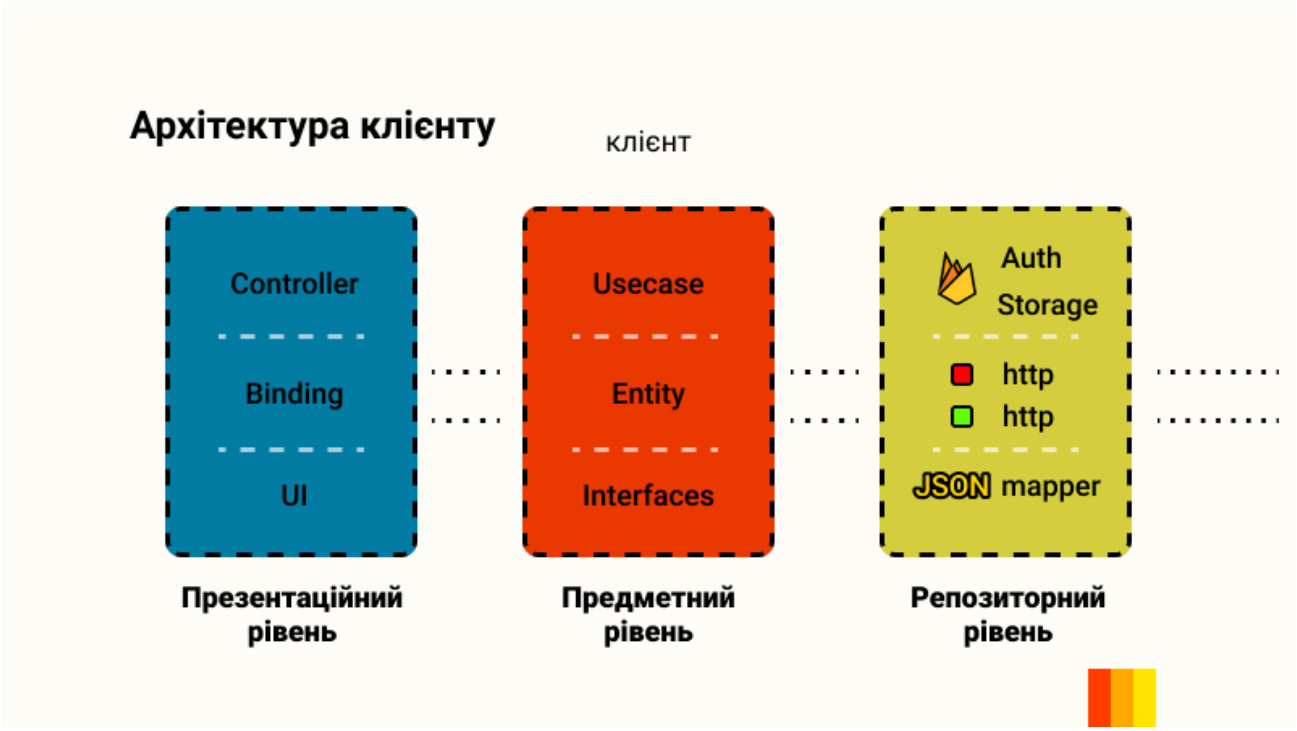


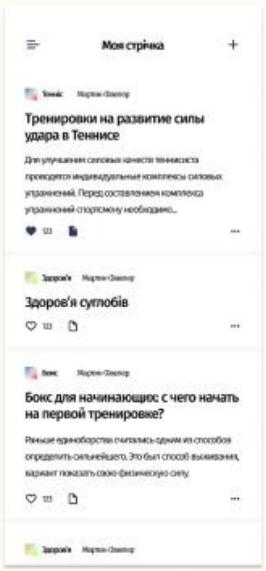
Аналоги

	Користувач - тренер	Розклад занять	Створення постів	Багатохвилинні тренування	Реальний тренер
Champ	+	-	+	+	+
Adidas	-	+	-	+	+
BetterMe	-	+	-	-	-

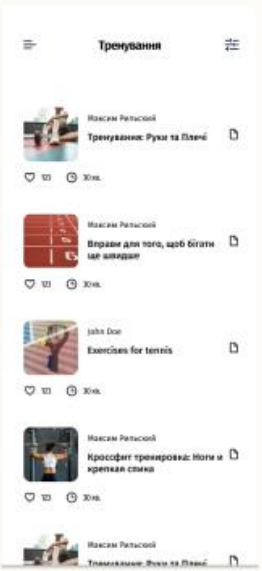
Архітектура системи



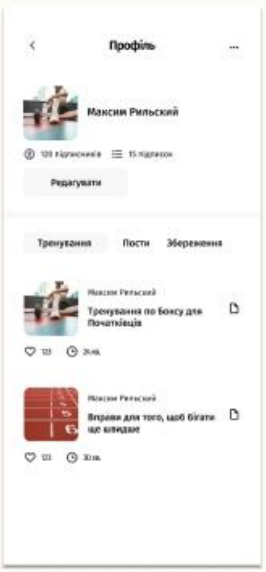




моя стрічка

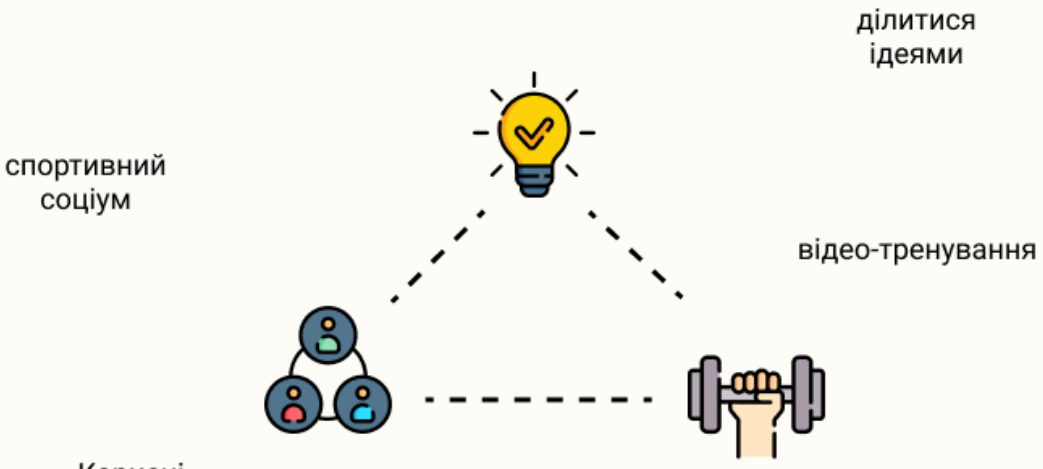


відео-тренування

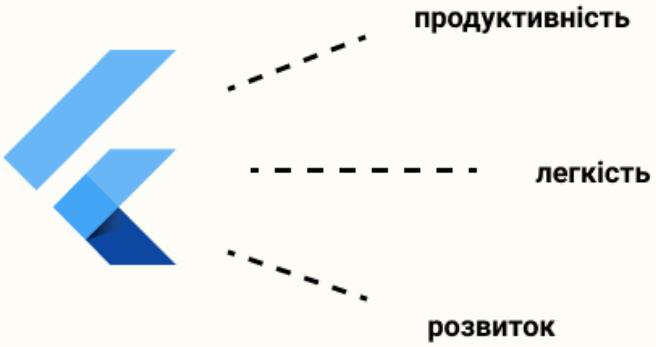


профіль

Висновки: Додаток



Висновки: Flutter



Висновки: Безсерверність

Дякую за увагу