

Детектор Плагиата v. 2961 - Отчёт оригинальности: 14.06.2026 11:51:22

Проанализированный документ: Диплом_Науменко.docx Лицензия: ВОЛОДИМИР МАТІЄВСЬКИЙ

? Тип поиска: Поиск переписанного ? Язык: Uk

? Тип проверки: Интернет

ТЕЕ и кодировка: DocX n/a

Детальный анализ тела документа:

? Диаграмма соотношения частей:

Плагат 0% Оригинал 96.89%
Кавычки 0.31% ИИ 2.81%



? Граф распределения зон:



? Источники плагиата: 0

? Детали обработанных ресурсов: 181 - ОК / 66 - Ошибка

? Важные замечания:

Википедия:



[не обнаружено]

Google Книги:



[не обнаружено]

Сервисы платных работ:



[не обнаружено]

Античит:



**Обнаружено
сокрытие!**

? Античит-отчет UACE:

1. Статус: Анализатор **Включен** Нормализатор **Включен** сходство символов установлено на **100%**
2. Обнаруженный процент загрязнения UniCode: **20,5%** с лимитом: 4%
3. Процент нераспознанных символов после нормализации: **9,4%**
4. Все подозрительные символы будут отмечены фиолетовым цветом: **Abcd...**
5. Найдены невидимые символы: 0

Рекомендации по оценке:

Особое внимание следует уделить анализу этого отчета! Предполагается, что этот документ содержит значительное количество символов, чуждых языку документа. Это прямое указание на то, что автор документа использовал специальное программное обеспечение/онлайн-веб-сервис, чтобы эффективно скрыть текст в попытке избежать обнаружения потенциального плагиата. Настоятельно рекомендуется передать это дело на более высокий уровень! В случае сомнений обращайтесь: в службу поддержки Детектора плагиата!

Алфавитная статистика и анализ символов:

 Активные ссылки (URL-адреса, извлеченные из документа):

URL не найдены

 Исключённые ресурсы:

URL не найдены

 Включённые ресурсы:

URL не найдены

Детальний аналіз документа:

Міністерство освіти і науки України ДЗ

Цитування: 0,01%

id: 1

«Луганський Національний університет імені Тараса Шевченка»

Факультет технологій та інформаційних систем (назва факультету, інституту)
Інформаційних технологій та систем (назва кафедри) Пояснювальна записка до
дипломного проєкту (роботи) БАКАЛАВРА (освітньо-кваліфікаційний рівень) на тему:
МОБІЛЬНИЙ ДОДАТОК ДЛЯ ФОРМУВАННЯ СТАРТАП-КОМАНДИ Виконав: студент 4 курсу
Спеціальності 121

Цитування: 0%

id: 2

«Інженерія програмного забезпечення»_

(шифр і назва напряму підготовки, спеціальності) Науменко О.В.
(прізвище та ініціали) Керівник _____ Донченко В.Ю. (прізвище та ініціали) Рецензент _____
Козуб Ю.Г. (прізвище та ініціали) Лубни 2026 ЗМІСТ ВСТУП11 РОЗДІЛ 1. АНАЛІЗ ПРЕДМЕТНОЇ
ОБЛАСТІ ТА ПОСТАНОВКА ЗАДАЧІ13 1.1. Поняття стартап-команд та особливості їх
формування13 1.2. Аналіз існуючих мобільних сервісів для пошуку партнерів15 1.2.1.
Сервіси знайомств15 1.2.2. Професійні соціальні мережі15 1.2.3. Платформи для спільнот
та подій16 1.2.4. Узагальнений аналіз існуючих рішень16 1.3. Проблеми та недоліки
наявних рішень17 1.4. Обґрунтування вибору концепції застосунку Synergy Hub19 1.5.
Формування вимог до програмного продукту20 Висновки до розділу22 РОЗДІЛ 2.
ПРОЄКТУВАННЯ МОБІЛЬНОГО ЗАСТОСУНКУ SYNERGY HUB23 2.1. Вибір мови програмування
та технологій23 2.2. Загальна архітектура програмного забезпечення25 2.3. Використання
Clean Architecture (Domain, Data, Presentation)27 2.4. Проєктування бази даних на основі
Firebase Firestore29 2.5. Моделювання взаємодії користувачів у системі31 2.6. Проєктування
інтерфейсу користувача34 2.7. Розробка онбордингу та збору інформації про
користувача35 2.8. Проєктування механізму пошуку та свайп-системи37 2.9. Проєктування
системи чатів та повідомлень40 Висновки до розділу41 РОЗДІЛ 3. РЕАЛІЗАЦІЯ МОБІЛЬНОГО
ДОДАТКА44 3.1. Архітектура реалізації клієнтської частини на Flutter та Dart44 3.2.
Використання патерну Repository, Unit of Work та Dependency Injection46 3.3. Реалізація
анімованого фону з використанням GLSL-шейдерів48 3.4. Реалізація системи
автентифікації користувачів49 3.5. Реалізація Discover-екрану та механізму свайпів51 3.6.
Реалізація чатів та збереження повідомлень53 3.7. Реалізація бізнес-логіки на основі
патерну Use Case55 3.8. Інтеграція з Firebase Firestore та Firestore Security Rules56 3.9.
Реалізація серверної логіки на основі Firebase Cloud Functions (TypeScript)58 3.9.1. Функція
матчингу користувачів у транзакції (matchUser)58 3.9.2. Обробка нових повідомлень та
розсилка push-сповіщень (onNewMessage)60 3.9.3. Синхронізація профілю користувача в
чатах (onUserUpdate)61 3.10. Інтеграція Firebase Cloud Messaging для push-сповіщень62
Висновки до розділу64 ВИСНОВКИ65 СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ68 ДОДАТОК А71
ДОДАТОК Б73 ДОДАТОК В74 ВСТУП У сучасних умовах розвитку цифрових технологій та
глобалізації економіки стартап-проєкти відіграють важливу роль у формуванні
інноваційного середовища та створенні конкурентоспроможних продуктів. Успішність
стартапу значною мірою залежить не лише від якості ідей, але й від професійного рівня,
мотивації та узгодженості дій членів команди. Процес формування стартап-команди є
складним та багатофакторним завданням, яке передбачає пошук фахівців з необхідними
компетенціями, оцінку їх професійних і особистісних якостей, а також забезпечення
ефективної комунікації між учасниками проєкту.

AI generated text detected: ChatGPT 5.3, : 71,18%

id: 3

Традиційні методи пошуку партнерів, зокрема через особисті знайомства, соціальні
мережі або професійні платформи, не завжди забезпечують швидкий та якісний підбір
команди. З розвитком мобільних технологій з'являється можливість створення
спеціалізованих програмних засобів, орієнтованих на підтримку процесу формування
стартап-команд. Мобільні застосунки дозволяють забезпечити доступність сервісу,
зручність взаємодії користувачів та оперативний обмін інформацією незалежно від їх
місцезнаходження. Даний дипломний проєкт націлений на розробку ефективного
інструменту для пошуку односторонніх, співзасновників та учасників стартап-проєктів із
використанням сучасних інформаційних технологій. Створення мобільного застосунку для
формування стартап-команд сприятиме розвитку підприємницької діяльності,

підвищенню рівня інноваційності та оптимізації процесів командної взаємодії. Об'єкт дослідження: мобільний додаток для формування стартап-команди.

Предмет дослідження: Методи, засоби та програмні технології розробки мобільних застосунків для пошуку та взаємодії учасників в стартап-проекті. Мета дослідження: проектування та реалізація мобільного застосунку для формування стартап-команд. Методи дослідження: порівняння та аналіз. Для досягнення поставленої мети необхідно вирішити наступні завдання: – проаналізувати предметну область та існуючі програмні рішення; – визначити вимоги до функціональних можливостей застосунку; – спроектувати архітектуру програмного забезпечення; – розробити клієнтську частину мобільного застосунку; – реалізувати механізми автентифікації та зберігання даних; – створити систему пошуку та взаємодії користувачів; – реалізувати серверну логіку на основі [Firebase Cloud Functions](#); Практичною цінністю роботи є розробка мобільного застосунку для формування стартап-команд. В першому розділі проаналізовано предметну область формування стартап-команд та виконано порівняльний аналіз існуючих мобільних сервісів ([Tinder](#), [Bumble](#), [LinkedIn](#), [Meetup](#)). Обґрунтовано концепцію застосунку [Synergy Hub](#) та сформовано функціональні й нефункціональні вимоги до програмного продукту. В другому розділі виконано проектування мобільного застосунку. Обґрунтовано вибір технологічного стеку, описано загальну архітектуру, спроектовано базу даних та змодельовано взаємодію користувачів у системі матчингу. Розроблено проекти інтерфейсу, онбордингу, свайп-системи та чатів. У третьому розділі детально описано реалізацію основних модулів застосунку клієнтської частини на [Flutter](#) із застосуванням патернів; серверної логіки; системи автентифікації; [push](#)-сповіщень та [GLSL](#)-шейдерів для анімованого фону. Наведено специфікацію функцій [matchUser](#), [onNewMessage](#) та [onUserUpdate](#).

РОЗДІЛ 1. АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ ТА ПОСТАНОВКА ЗАДАЧ

1.1. Поняття стартап-команд та особливості їх формування

Стартап-команда — це група осіб, об'єднаних спільною ідеєю створення інноваційного продукту або сервісу з високим потенціалом розвитку та масштабування. Основною метою такої команди є реалізація бізнес-ідеї в умовах обмежених ресурсів, високої конкуренції та невизначеності. Формування стартап-команди є одним із ключових факторів успішності проекту, оскільки саме людський капітал визначає якість прийняття рішень, швидкість розвитку та адаптацію до змін ринку. До основних характеристик ефективної стартап-команди належать: – комплементарність навичок — поєднання технічних, управлінських, маркетингових та фінансових компетенцій; – спільність цінностей і бачення — розуміння цілей проекту та готовність працювати на результат; – гнучкість та адаптивність — здатність швидко реагувати на зміни; – високий рівень мотивації — зацікавленість у розвитку продукту; – комунікаційна ефективність — вміння взаємодіяти та вирішувати конфлікти. Процес формування стартап-команди зазвичай включає такі етапи: – визначення ідеї та цілей проекту; – аналіз необхідних компетенцій; – пошук потенційних учасників; – оцінка професійних та особистісних якостей; – формування ролей у команді; – налагодження внутрішніх процесів взаємодії. У сучасних умовах пошук учасників команди все частіше відбувається за допомогою цифрових платформ, соціальних мереж та мобільних застосунків. Це дозволяє залучати фахівців з різних регіонів та галузей, проте ускладнює процес оцінювання їхньої сумісності. Саме тому актуальним є створення спеціалізованих програмних рішень, орієнтованих не лише на пошук людей, а й на формування ефективних стартап-команд з урахуванням професійних і особистісних характеристик. За даними досліджень, проведених [CB Insights](#), серед основних причин невдач стартапів 23% припадають на невідповідний склад команди. Це підтверджує критичну важливість правильного підбору учасників на початкових етапах розвитку проекту. Стартап-команди відрізняються від традиційних робочих груп кількома суттєвими ознаками. По-перше, вони функціонують в умовах високої невизначеності, коли бізнес-модель ще не валідована, а ринок не до кінця визначений. По-друге, учасники стартап-команди часто поєднують кілька ролей, що вимагає широкого спектру компетенцій. По-третє, мотивація учасників базується не лише на фінансовій винагороді, але й на вірі в ідею та бажанні створити інноваційний продукт. Типова стартап-команда включає такі ключові ролі: технічний співзасновник ([CTO](#)), який відповідає за технічну реалізацію продукту; продуктове керівництво, який визначає стратегію розвитку; дизайнер, який забезпечує користувацький досвід; маркетолог, який відповідає за просування. Залежно від специфіки проекту, склад команди може варіюватися. Окремою проблемою є оцінка сумісності потенційних учасників. Традиційні методи, такі як співбесіди та рекомендації,

мають обмежену ефективність у контексті стартапів, оскільки не враховують специфіку роботи в умовах невизначеності. Необхідні нові інструменти, які дозволять оцінювати не лише професійні навички, але й співпадіння бачення, цілей та робочого стилю.

1.2. Аналіз існуючих мобільних сервісів для пошуку партнерів

На сьогодні існує значна кількість мобільних сервісів, які опосередковано використовуються для пошуку партнерів, співзасновників та однодумців. Проте більшість із них не орієнтована безпосередньо на формування стартап-команд.

1.2.1. Сервіси знайомств

До цієї групи належать застосунки, спочатку створені для особистих знайомств, але які іноді використовуються з професійною метою. Прикладами є [Tinder](#) та [Bumble](#). Основні особливості: – механізм свайпів для вибору користувачів; – мінімальна інформація про професійні навички; – орієнтація на особисті інтереси. Недоліком таких сервісів є відсутність інструментів для оцінки професійної сумісності та командної взаємодії. Варто зазначити, що сервіс [Bumble](#) має окремий режим [Bumble Bizz](#), спрямований на професійні знайомства. Проте цей режим зберігає загальну концепцію додатку і не надає спеціалізованих інструментів для оцінки технічних навичок або формування проектних команд. Профілі у [Bumble Bizz](#) містять лише загальну інформацію про посаду та компанію, без деталізації технологічного стеку чи рівня досвіду. Окремо слід виділити додаток [CoFoundersLab](#), який позиціонується як платформа для пошуку співзасновників. Проте цей сервіс доступний переважно у веб-форматі, має обмежену мобільну версію та не надає автоматизованого алгоритму матчингу, покладаючись на ручний пошук та фільтрацію.

1.2.2. Професійні соціальні мережі

Професійні платформи дозволяють знаходити фахівців різного профілю. Найбільш поширеним прикладом є [LinkedIn](#). Переваги: – детальний опис досвіду та навичок; – можливість налагодження ділових контактів; – доступ до широкої аудиторії спеціалістів. Недоліки: – відсутність механізмів швидкого підбору команди; – складність пошуку людей саме для стартап-проектів; – орієнтація на класичне працевлаштування.

1.2.3. Платформи для спільнот та подій

До цієї категорії належать сервіси для організації зустрічей та заходів, наприклад [Meetup](#). Основні можливості: – пошук спільнот за інтересами; – участь у тематичних подіях; – живе спілкування. Недоліки: – відсутність системи персонального підбору партнерів; – обмежена підтримка онлайн-взаємодії; – відсутність інтегрованих чатів для проектної роботи.

1.2.4. Узагальнений аналіз існуючих рішень

Аналіз наявних мобільних сервісів дозволяє зробити такі висновки: – більшість платформ не спеціалізується на стартап-командах; – відсутні алгоритми оцінки сумісності учасників; – недостатньо інструментів для довготривалої співпраці; – переважає орієнтація на знайомства або працевлаштування; – обмежена підтримка командної комунікації. Таким чином, наявні сервіси лише частково задовольняють потреби стартап-спільноти та не забезпечують комплексного підходу до формування команд. Це обґрунтовує доцільність розробки спеціалізованого мобільного застосунку [Synergy Hub](#), орієнтованого на професійний підбір учасників, підтримку комунікації та розвиток стартап-проектів. Для більш наочного порівняння основних характеристик існуючих сервісів та запропонованого рішення [Synergy Hub](#) складено порівняльну таблицю (таблиця 1.1). Аналіз показує, що [Synergy Hub](#) є єдиним рішенням, яке поєднує всі ключові функціональні можливості, необхідні для формування стартап-команд: інтуїтивний свайп-механізм, детальний професійний профіль, автоматизований алгоритм оцінки сумісності та вбудовану систему комунікації.

Критерій	Tinder	Bumble	LinkedIn	Meetup	Synergy Hub
Свайп-механізм	Так	Ні	Ні	Ні	Так
Професійний профіль	Ні	Так	Так	Ні	Так
Частково	Так	Так	Так	Так	Так
Алгоритм сумісності	Ні	Ні	Ні	Ні	Так
Вбудований чат	Так	Так	Так	Так	Так
Орієнтація на стартапи	Ні	Частково	Частково	Так	Push-сповіщення

1.3. Проблеми та недоліки наявних рішень

На підставі проведеного аналізу можна виділити ключові проблеми існуючих програмних рішень, які ускладнюють процес формування стартап-команд: Відсутність спеціалізації. Жоден із розглянутих сервісів не орієнтований безпосередньо на формування стартап-команд. Сервіси знайомств не враховують професійні характеристики, а професійні мережі фокусуються на класичному працевлаштуванні. Відсутність алгоритмів оцінки сумісності. Жодна з проаналізованих платформ не пропонує автоматизованої оцінки сумісності учасників за такими параметрами, як рівень досвіду, технічні навички та бачення проекту. Підбір здійснюється виключно вручну. Обмеженість комунікаційних інструментів. Більшість платформ не надає інтегрованих засобів для проектної комунікації. Навіть за наявності чатів, вони не адаптовані для потреб командної співпраці. Відсутність онбордингу. Існуючі сервіси не збирають детальну інформацію про професійний профіль користувача на етапі реєстрації, що ускладнює подальший підбір партнерів. Обмежена масштабованість. Більшість рішень

не передбачає гнучкої системи фільтрації та персоналізованого пошуку за спеціалізацією, роллю чи рівнем досвіду. Відсутність автоматичного створення робочого простору. Після знаходження потенційного партнера користувачі змушені переходити на зовнішні платформи для комунікації ([Telegram](#), [Slack](#), [Discord](#)), що ускладнює процес та збільшує ймовірність втрати контакту. Відсутність зворотного зв'язку щодо якості матчів. Жодна з платформ не надає інформації про ступінь сумісності між учасниками, що змушує користувачів самотійно оцінювати потенційних партнерів без будь-яких кількісних показників. Таким чином, існуючі програмні рішення мають системні недоліки, які не дозволяють ефективно використовувати їх для формування стартап-команд. Це підтверджує необхідність створення спеціалізованого інструменту, орієнтованого саме на цю задачу.

1.4. Обґрунтування вибору концепції застосунку [Synergy Hub](#)

На основі проведеного аналізу існуючих рішень та визначених проблем було сформовано концепцію мобільного застосунку [Synergy Hub](#), який поєднує переваги різних типів платформ та усуває їх ключові недоліки. Основні принципи концепції: – свайп-механізм для швидкого та інтуїтивного перегляду профілів потенційних партнерів, запозичений з сервісів знайомств, але адаптований під професійний контекст; – детальний професійний профіль з інформацією про спеціалізацію, рівень досвіду, навички та бачення проєкту, подібно до [LinkedIn](#), але зі спрощеним введенням; – алгоритм матчингу з тривимірною оцінкою сумісності (досвід, навички, бачення), який автоматично визначає ступінь відповідності між користувачами; – інтегрована система чатів для комунікації між учасниками, яка створюється автоматично після взаємного матчу; – багатоетапний онбординг для збору інформації про користувача, що забезпечує якісний підбір команди. Назва застосунку — [Synergy Hub](#) — відображає його основну місію: створення середовища, де синергія між учасниками стартап-команди досягається завдяки ефективному підбору та комунікації. Вибір свайп-механізму як основного способу взаємодії обумовлений кількома факторами. По-перше, дослідження у сфері UX-дизайну демонструють, що свайп-жести є найбільш інтуїтивними для мобільних пристроїв. По-друге, бінарне рішення (лайк/дизлайк) знижує когнітивне навантаження на користувача, що особливо важливо при перегляді великої кількості профілів. По-третє, взаємне підтвердження інтересу створює більш якісну базу для початку комунікації, ніж одностороннє звернення. Тривимірна модель оцінки сумісності є ключовою інновацією [Synergy Hub](#). На відміну від існуючих рішень, які покладаються виключно на ручний підбір, алгоритм автоматично розраховує три незалежні показники: співвідношення рівнів досвіду, перетин технічних навичок та співпадіння бачення проєкту. Кожен показник оцінюється за шкалою від 0 до 10, що дозволяє користувачам швидко оцінити потенціал співпраці. Архітектурно застосунок розроблено з урахуванням можливості масштабування. Використання [Firebase](#) як [Backend-as-a-Service](#) забезпечує автоматичне масштабування серверної частини, а фреймворк [Flutter](#) дозволяє підтримувати множину платформ ([Android](#), [iOS](#), [Web](#)) з єдиної кодової бази. Це суттєво знижує витрати на розробку та підтримку. Ключовими відмінностями [Synergy Hub](#) від існуючих рішень є: – спеціалізація виключно на стартап-командах; – автоматизований алгоритм оцінки сумісності; – тривимірна модель матчингу (досвід, навички, бачення); – інтеграція пошуку та комунікації в єдиному інтерфейсі; – кросплатформність завдяки використанню [Flutter](#).

1.5. Формування вимог до програмного продукту

На основі аналізу предметної області та обраної концепції було сформовано перелік функціональних та нефункціональних вимог до мобільного застосунку [Synergy Hub](#).

Функціональні вимоги:

- реєстрація та автентифікація користувачів ([email/password](#), [Google OAuth](#));
- багатоетапний онбординг для збору інформації про спеціалізацію, досвід, навички та роль;
- перегляд профілів інших користувачів у форматі свайп-карток;
- фіксація рішення (лайк/дизлайк) та збереження історії взаємодії;
- автоматичне створення чату після взаємного матчу;
- обмін текстовими повідомленнями у реальному часі;
- push-сповіщення про нові матчі та повідомлення;
- редагування профілю користувача;
- фільтрація користувачів за спеціалізацією та досвідом.

Нефункціональні вимоги:

- кросплатформність ([Android](#), [iOS](#), [Web](#));
- відповідність стандартам [Material Design](#);
- час відгуку інтерфейсу не більше 2 секунд;
- масштабованість серверної частини;
- безпека зберігання та передачі даних;
- підтримка роботи в офлайн-режимі (кешування);
- автоматичний деплой через [CI/CD](#).

Таблиця 1.2. Функціональні вимоги до застосунку [Synergy Hub](#)

Вимога	Пріоритет
Реєстрація через email/password	Високий FR-01
Автентифікація через Google OAuth	Високий FR-02
Багатоетапний онбординг	Високий FR-03
Свайп-перегляд профілів	Високий FR-04
Фіксація лайку/дизлайку	Високий FR-05
Автоматичне створення чату після матчу	Високий FR-06
Обмін повідомленнями у реальному	Високий FR-07

часі Високий [FR-08 Push](#)-сповіщення Середній [FR-09](#) Редагування профілю Середній [FR-10](#) Фільтрація за спеціалізацією Середній Додатково сформовано вимоги до інфраструктури та процесу розробки: – використання системи контролю версій [Git](#); – автоматизація збірки та деплою через [CI/CD](#); – збереження секретних даних ([API-ключі](#), токени) у захищеному середовищі; – підтримка [Firebase Emulator](#) для локального тестування; – покриття критичної бізнес-логіки юніт-тестами. Сформовані вимоги є основою для подальшого проєктування архітектури та реалізації функціональних модулів застосунку, що описано у наступних розділах.

 AI generated text detected: ChatGPT 5.3, : 61,53%

id: 4

Висновки до розділу У даному розділі було проведено аналіз предметної області формування стартап-команд та досліджено особливості підбору учасників для реалізації інноваційних проєктів. Встановлено, що успішність стартапу значною мірою залежить від правильного формування команди, рівня професійної сумісності учасників, їхніх навичок, досвіду та спільного бачення розвитку проєкту. Було виконано аналіз існуючих програмних рішень для пошуку партнерів та командування, зокрема сервісів [Tinder](#), [Bumble](#), [LinkedIn](#), [Meetup](#) та інших платформ. Результати дослідження показали, що наявні рішення лише частково задовольняють потреби стартап-спільноти, оскільки не забезпечують комплексного підходу до формування команд, не містять ефективних механізмів оцінювання сумісності учасників та мають обмежені можливості для подальшої командної взаємодії. На основі виявлених недоліків було обґрунтовано доцільність створення спеціалізованого мобільного застосунку [Synergy Hub](#), орієнтованого на пошук потенційних учасників стартап-команд, оцінювання їхньої сумісності та підтримку комунікації між ними. Сформовано концепцію застосунку, визначено його основні функціональні можливості та переваги порівняно з існуючими аналогами.

Також було сформовано функціональні та нефункціональні вимоги до програмного продукту, які визначають основні характеристики майбутньої системи та слугують основою для її подальшого проєктування й реалізації. Отримані результати аналізу підтверджують актуальність розробки мобільного застосунку [Synergy Hub](#) та визначають напрями подальшого проєктування архітектури, моделей даних і програмних компонентів системи.

РОЗДІЛ 2. ПРОЄКТУВАННЯ МОБІЛЬНОГО ЗАСТОСУНКУ [SYNERGY HUB](#)

2.1. Вибір мови програмування та технологій

Для реалізації мобільного застосунку [Synergy Hub](#) було обрано набір сучасних технологій та інструментів, що забезпечують ефективну розробку, тестування та розгортання продукту. [Flutter 3.9](#) та [Dart](#) — основний фреймворк та мова програмування для створення кросплатформного клієнтського застосунку. [Flutter](#) забезпечує компіляцію в нативний код для [Android](#), [iOS](#) та [Web](#) з єдиної кодової бази. [TypeScript](#) — мова програмування для реалізації серверної логіки [Firebase Cloud Functions](#). [TypeScript](#) забезпечує строгу типізацію та покращує підтримуваність коду. [Firebase](#) — платформа [Backend-as-a-Service \(BaaS\)](#) від [Google](#), що включає: – [Firebase Auth](#) — автентифікація користувачів; – [Firebase Firestore](#) — хмарна [NoSQL](#) база даних; – [Firebase Cloud Functions](#) — серверна логіка; – [Firebase Cloud Messaging](#) — [push](#)-сповіщення; – [Firebase Storage](#) — зберігання файлів; – [Firebase Hosting](#) — хостинг веб-версії. Інструменти кодогенерації. Для зменшення обсягу шаблонного коду ([boilerplate](#)) використовуються інструменти кодогенерації: [Freezed](#) для створення імутабельних [data](#)-класів з підтримкою [copyWith](#), [equality](#) та [sealed class pattern matching](#); [JsonSerializable](#) для автоматичної серіалізації/десеріалізації [JSON](#); [go_router_builder](#) для типобезпечної генерації маршрутів; [flutter_gen](#) для генерації констант для ресурсів (зображення, шрифти). Функціональне програмування. Бібліотека [dartx](#) надає функціональні типи [Either](#), [Left](#), [Right](#) та [Option T](#), що дозволяють явно моделювати успішні та помилкові результати без використання [exceptions](#). Це покращує надійність коду та робить обробку помилок більш передбачуваною. Реактивне програмування. Бібліотека [RxDart](#) розширює стандартні [Stream API Dart](#) додатковими операторами ([debounce](#), [switchMap](#), [combineLatest](#)), що спрощує роботу з асинхронними потоками даних, особливо при реалізації пошуку та фільтрації. Літінг та якість коду. Для забезпечення якості коду використовується файл [analysis_options.yaml](#) з правилами: [avoid_print](#) (забороняє [debug](#)-виведення), [prefer_single_quotes](#) (єдиний стиль лапок), [require_trailing_commas](#) (покращення [git diff](#)), [prefer_relative_imports](#) (чистіші імпорти). На сервері використовується [ESLint](#) з конфігурацією [Google style guide](#).

Бібліотека	Версія	Призначення
flutter_bloc	9.1.1	Керування станом (BLoC)
go_router	17.1.0	Типобезпечна маршрутизація
get_it	9.2.1	Впровадження залежностей
freezed	3.2.3	

Генерація імутабельних моделей [firebase_core](#) 4.1.1 Ініціалізація [Firebase cloud_firestore](#) 6.1.1 База даних [Firestore firebase_auth](#) 6.1.0 Автентифікація [firebase_messaging](#) 16.0.2 Push-сповіщення [flutter_shaders](#) 0.1.3 GLSL-шейдери [dartx](#) 0.10.1 Функціональне програмування ([Either](#)) Таблиця 2.2. Залежності серверної частини Бібліотека Версія Призначення [firebase-admin](#) 12.6.0 Серверний [Firebase SDK firebase-functions](#) 6.0.1 [Cloud Functions runtime typescript](#) 5.9.3 Мова програмування [jest](#) 30.3.0 Фреймворк тестування [ts-jest](#) 29.4.6 [TypeScript](#) для [jest](#) 2.2. Загальна архітектура програмного забезпечення

Загальна архітектура програмного забезпечення мобільного застосунку [Synergy Hub](#) побудована з урахуванням принципів модульності, масштабованості та підтримованості. Під час проєктування системи було обрано підхід [Clean Architecture](#), який передбачає чітке розмежування відповідальностей між окремими компонентами програми. Основною метою використання такої архітектури є забезпечення незалежності бізнес-логіки від інтерфейсу користувача, зовнішніх сервісів та конкретних технологічних реалізацій. Це дозволяє спростити процес супроводу програмного продукту, його тестування та подальшого розвитку. Система складається з двох основних компонентів: – клієнтська частина — мобільний застосунок на [Flutter \(Dart\)](#), що реалізує інтерфейс користувача, бізнес-логіку та взаємодію з [Firebase](#); – серверна частина — набір [Firebase Cloud Functions \(TypeScript\)](#), що забезпечує матчинг, сповіщення та синхронізацію даних. Клієнтська частина побудована за принципом модульної архітектури, де кожна функціональна область (автентифікація, онбординг, матчинг, чати, профіль, сповіщення) є ізольованим модулем з власною структурою [Domain-Data-Presentation](#). Така модульність забезпечує можливість незалежної розробки та тестування окремих функціональних областей. Серверна частина реалізована у вигляді [Firebase Cloud Functions](#) — безсерверних ([serverless](#)) функцій, які виконуються у хмарному середовищі [Google Cloud](#). Використання безсерверної архітектури забезпечує автоматичне масштабування при зростанні навантаження, оплату лише за фактичне використання ресурсів та відсутність необхідності управління серверною інфраструктурою. Взаємодія між клієнтською та серверною частинами здійснюється двома способами: – пряме звернення до [Firestore](#) — для операцій читання/запису даних (профіль, чати, повідомлення). [Firestore SDK](#) забезпечує кешування, офлайн-підтримку та оновлення в реальному часі; – виклик [Cloud Functions](#) — для складних операцій, що вимагають серверної валідації або атомарності (матчинг, розрахунок сумісності). [Callable Functions](#) забезпечують автоматичну передачу токена автентифікації. Третім каналом комунікації є [Firebase Cloud Messaging \(FCM\)](#), який забезпечує push-сповіщення. [Cloud Functions](#) відправляють сповіщення через [Firebase Admin SDK](#), а клієнтський застосунок отримує їх через [FCM SDK](#).

Рис. 2.1. Загальна архітектура системи [Synergy Hub](#)

2.3. Використання [Clean Architecture \(Domain, Data, Presentation\)](#)

Архітектура застосунку складається з трьох основних рівнів: – [Presentation Layer](#) (рівень представлення); – [Domain Layer](#) (рівень бізнес-логіки); – [Data Layer](#) (рівень доступу до даних). Кожен із зазначених рівнів виконує чітко визначені функції та взаємодіє з іншими рівнями через абстракції. Рівень представлення ([Presentation Layer](#)) відповідає за взаємодію з користувачем та відображення інформації в інтерфейсі. До його основних функцій належать: формування графічного інтерфейсу за допомогою [Flutter](#); обробка дій користувача (натискання, свайпи, введення даних); відображення станів завантаження, помилок та результатів операцій; взаємодія з бізнес-логікою через менеджер станів [BLoC](#). Рівень бізнес-логіки ([Domain Layer](#)) є центральною частиною архітектури та не залежить від зовнішніх бібліотек і платформ. До його складу входять: бізнес-моделі ([UserData](#), [Chat](#), [ChatMessage](#), [Matches](#) тощо); інтерфейси репозиторіїв; сценарії використання ([Use Case](#)); правила обробки даних. Рівень доступу до даних ([Data Layer](#)) відповідає за отримання, збереження та обробку інформації з різних джерел. Основними компонентами даного рівня є: реалізації репозиторіїв; сервіси роботи з [Firebase Firestore](#); [Data Transfer Objects \(DTO\)](#) для серіалізації/десеріалізації. Взаємодія між рівнями відбувається відповідно до принципу інверсії залежностей: [Presentation Layer](#) звертається до [Domain Layer](#) через [Use Case](#); [Domain Layer](#) працює з [Data Layer](#) через інтерфейси репозиторіїв; [Data Layer](#) реалізує доступ до конкретних джерел даних. У процесі розробки застосовуються такі архітектурні патерни: [Clean Architecture](#) — для організації структури проєкту; [Repository Pattern](#) — для абстрагування доступу до даних; [BLoC Pattern](#) — для керування станами; [Dependency Injection](#) — для керування залежностями; [Unit of Work](#) — для транзакційних операцій на сервері. Принцип інверсії залежностей ([Dependency Inversion Principle](#)) є фундаментальним для обраної архітектури. Він передбачає, що модулі верхнього рівня не повинні залежати від модулів нижнього рівня. Обидва рівні повинні залежати від абстракції. У контексті [Synergy Hub](#) це означає,

що BLoC-компоненти залежать від інтерфейсів репозиторіїв ([IAuthRepository](#), [IChatRepository](#)), а не від конкретних реалізацій [Firebase](#). Така архітектура забезпечує можливість замінити джерела даних (наприклад, перехід з [Firebase](#) на власний бекенд) без модифікації бізнес-логіки та інтерфейсу. Кожен рівень може тестуватися ізольовано: [Domain Layer](#) — через юніт-тести з моками репозиторіїв, [Presentation Layer](#) — через [bloc_test](#) з моками [Use Cases](#), [Data Layer](#) — через інтеграційні тести з [Firebase Emulator](#). Організація коду за фічерним принципом ([feature-based structure](#)) забезпечує модульність проєкту. Кожна функціональна область ([auth](#), [matches](#), [chat](#), [user](#), [onboard](#), [survey](#), [notifications](#)) містить власні [domain](#), [data](#) та [presentation](#) рівні. Це дозволяє розробникам працювати над окремими фічами незалежно та мінімізує конфлікти при паралельній розробці. Файлова структура фічерного модуля має такий вигляд: [features/ matches/ data/ models/ // DTO](#) для серіалізації [repositories/ // Firebase](#) реалізації [domain/ models/ // Бізнес-моделі \(Freezed\) repositories/ // Інтерфейси usecases/ // Сценарії використання presentation/ bloc/ // BLoC/Cubit](#) компоненти [pages/ // Екрани та віджети di/ matches injection.dart // DI](#) налаштування Рис. 2.2. Діаграма залежностей між рівнями [Clean Architecture](#) 2.4.

Проектування бази даних на основі [Firebase Firestore](#) Для зберігання та обробки даних у мобільному застосунку [Synergy Hub](#) використовується хмарна [NoSQL](#)-база даних [Firebase Firestore](#). Дане рішення забезпечує високу масштабованість, синхронізацію в реальному часі та інтеграцію з мобільними платформами. [Firestore](#) використовує ієрархічну модель зберігання даних, що базується на колекціях, документах та вкладених підколекціях. На кореневому рівні бази даних розміщено такі основні колекції: – [users](#) — зберігання інформації про користувачів ([uid](#), [firstName](#), [lastName](#), [age](#), [avatarUrl](#), [aboutMe](#), [profiler](#)); – [chats](#) — зберігання чатів з підколекцією [messages](#) для повідомлень; – [matches](#) — зберігання інформації про матчі між користувачами. Документ користувача містить вкладену модель [Profiler](#), яка описує професійну спеціалізацію: – рівень досвіду ([junior](#), [middle](#), [senior](#), [lead](#)/[expert](#)); – роль у проєкті; – основна спеціалізація ([frontend](#), [backend](#), [mobile](#), [devops](#) тощо); – перелік навичок ([skills](#)); – напрямки зацікавленості ([searchingSpecializations](#)). У межах кожного документа користувача створено набір вкладених підколекцій: – [onboardStatus](#) — зберігає стан проходження онбордингу; – [notificationConfig](#) — зберігає [FCM](#)-токени та налаштування сповіщень; – [notifications](#) — зберігає історію сповіщень користувача; – [chats](#) — посилання на чати, в яких бере участь користувач. Документ матчу ідентифікується за ключем, що формується сортованою конкатенацією [UID](#) обох користувачів (наприклад, [uid1_uid2](#)). Це гарантує унікальність запису незалежно від напрямку ініціалізації матчу. Детальна структура бази даних наведена у Додатку Б. Вибір [Firebase Firestore](#) як основної бази даних обумовлений кількома факторами. По-перше, [Firestore](#) забезпечує реальночасову синхронізацію даних через [Snapshot Listeners](#), що є критичним для системи чатів. По-друге, [Firestore](#) має вбудований офлайн-режим з автоматичною синхронізацією при відновленні з'єднання. По-третє, [Firestore](#) масштабується автоматично і не вимагає ручного налаштування серверної інфраструктури. Для забезпечення консистентності даних при операціях матчингу використовуються [Firestore Transactions](#). Транзакція гарантує, що всі операції (читання стану матчу, створення чату, оновлення статусу) виконуються атомарно. Якщо будь-яка операція в транзакції завершується помилкою, всі зміни відкочуються. Індексация даних у [Firestore](#) налаштована для оптимізації основних запитів: пошук чатів за учасником ([array-contains](#) на полі [participants](#)), пошук матчів за статусом та учасником, пошук чатів за ключовими словами ([searchKeywords](#)). Складені індекси створюються автоматично при першому запиті через [Firebase Console](#). Рис. 2.3. Даталогічна модель БД ([NoSQL Firebase](#)) 2.5. Моделювання взаємодії користувачів у системі Взаємодія користувачів у системі [Synergy Hub](#) побудована на принципі двостороннього підтвердження ([mutual matching](#)). Процес взаємодії включає кілька послідовних етапів: Етап 1: Перегляд профілю. Користувач переглядає картку іншого учасника на [Discover](#)-екрані, де відображається ім'я, спеціалізація, рівень досвіду, навички та опис. Етап 2: Прийняття рішення. Користувач виконує свайп вправо (лайк) або вліво (дизлайк). Рішення фіксується через виклик [Cloud Function matchUser](#). Етап 3: Обробка матчу. [Cloud Function](#) перевіряє наявність зустрічної реакції. Якщо обидва користувачі виразили зацікавленість, відбувається двосторонній матч ([twoWayMatch](#)). Етап 4: Створення чату. Після двостороннього матчу автоматично створюється чат між учасниками. Обидва користувачі отримують [push](#)-сповіщення. Етап 5: Комунікація. Користувачі обмінюються повідомленнями у реальному часі через [Firebase Firestore](#). Кожне нове повідомлення генерує [push](#)-сповіщення для отримувача. При двосторонньому матчі розраховуються три показники сумісності: – [Experience Score](#) — оцінка співвідношення

рівнів досвіду (формула: $\max(10 - \text{diff} \times 3, 2)$); - **Skills Score** — оцінка перетину технічних навичок ($\text{intersection} / \max(\text{skills_a}, \text{skills_b}) \times 10$); - **Vision Score** — оцінка співпадіння бачення на основі текстового аналізу полів **aboutMe**. Рис. 2.4. Діаграма взаємодії користувачів у системі матчингу Стани матчу визначаються перелічувальним типом **CompletenessStatus**: - **none** — один з учасників виконав дизлайк, матч не відбувся; - **oneWayMatch** — лише один учасник виразив зацікавленість (лайк), очікується рішення другого; - **twoWayMatch** — обидва учасники виразили зацікавленість, створюється чат. Важливим архітектурним рішенням є генерація унікального ідентифікатора матчу через сортовану конкатенацію **UID** обох користувачів. Наприклад, для користувачів з **UID**

” Цитирования: 0% id: 5

“abc123”

та

” Цитирования: 0% id: 6

“xyz789”

ідентифікатор матчу буде

” Цитирования: 0% id: 7

“abc123_xyz789”

(незалежно від того, хто першим виконав свайп). Це забезпечує відсутність дублікатів і дозволяє обом учасникам знайти запис матчу за єдиним ключем. Після створення чату система автоматично генерує пошукові ключові слова (**searchKeywords**) на основі імен учасників. Алгоритм генерації створює посимвольні префікси для підтримки автодоповнення при пошуку чатів. Наприклад, для імені

” Цитирования: 0% id: 8

“Oleksandr”

будуть створені ключові слова:

” Цитирования: 0% id: 9

“o”,

” Цитирования: 0% id: 10

“ol”,

” Цитирования: 0% id: 11

“ole”,

” Цитирования: 0% id: 12

“olek”

і так далі. Система сповіщень інтегрована у процес матчингу на двох рівнях. При односторонньому лайку відправляється сповіщення з текстом

” Цитирования: 0,01% id: 13

“A potential co-founder liked your profile”.

При двосторонньому матчі обидва учасники отримують сповіщення

” Цитирования: 0,01% id: 14

“It's a Match!”

з даними для навігації до екрану матчу. 2.6. Проектування інтерфейсу користувача Інтерфейс користувача **Synergy Hub** спроектовано з урахуванням принципів **Material Design** та сучасних підходів до мобільного **UX**-дизайну. Навігація побудована на основі бібліотеки **GoRouter** з типобезпечною маршрутизацією. Структура навігації включає: - екран авторизації — головний екран з вибором методу входу (**email/password** або **Google**); - онбординг — багатоетапна форма збору інформації про користувача; - **Discovery** — основний екран з картками для свайпів; - **Matches** — список матчів та активних чатів; - **Chat** — екран обміну повідомленнями; - **Profile** — перегляд та редагування профілю; - **Notifications** — список сповіщень. Маршрутизація реалізована за допомогою вкладених **ShellRoute** та **StatefulShellRoute**, що забезпечує збереження стану між вкладками та плавні анімації переходів (**FadeTransition**). Ієрархія маршрутів: **Root Shell** (**RootPageRoute**)

[Authentication Shell](#) / - [AuthorizationPageRoute](#) /[sign-up](#) - [RegisterPageRoute](#) /[sign-in](#) - [LoginPageRoute](#) [Onboarding Shell](#) /[onboard](#) - [OnboardSurveyPageRoute](#) /[onboard-about](#) - [OnboardAboutYouPageRoute](#) [Home Stateful Shell](#) /[discovery](#) - [DiscoveryPageRoute](#) /[matches](#) - [MatchesPageRoute](#) /[chat](#) - [ChatPageRoute](#) (?[chatId](#)) /[profile](#) - [ProfilePageRoute](#) /[notification](#) - [NotificationPageRoute](#) [StatefulShellRoute](#) забезпечує збереження стану вкладок при навігації між ними. Це означає, що при переході з [Discovery](#) на [Matches](#) і назад позиція скролу та завантажені дані зберігаються. Для забезпечення адаптивності інтерфейсу використовується [ResponsiveFramework](#) з такими breakpoints: [MOBILE](#) (0-450px), [TABLET](#) (451-900px), [DESKTOP](#) (901-1920px), [4K](#) (1921px+). Це дозволяє коректно відображати інтерфейс як на мобільних пристроях, так і у веб-версії. Кольорова палітра базується на відтінках [Coral/Peach](#) (#F27361, #FA9E73, #FFBF8C), що відображає фірмовий стиль [Synergy Hub](#) та використовується у [GLSL](#)-шейдерах для анімованого фону.

2.7. Розробка онбордингу та збору інформації про користувача Онбординг у [Synergy Hub](#) є ключовим елементом, що забезпечує збір необхідної інформації для роботи алгоритму матчингу. Він складається з двох основних етапів: Етап 1 — [About You](#). Користувач заповнює основну персональну інформацію: ім'я, прізвище, вік, фото профілю та короткий опис ([aboutMe](#)). Ці дані використовуються для відображення на картці профілю та для розрахунку [Vision Score](#). Етап 2 — [Professional Survey](#). Багатоетапне опитування, що включає чотири стадії: - [Specialization Stage](#) — вибір основної спеціалізації ([Frontend Developer](#), [Backend Developer](#), [Mobile Developer](#), [DevOps Engineer](#), [Data Scientist](#), [UI/UX Designer](#), [Product Manager](#) тощо); - [Experience Stage](#) — визначення рівня досвіду ([Junior](#), [Middle](#), [Senior](#), [Lead/Expert](#)); - [Role Stage](#) — вибір ролі у стартап-проєкті ([Technical Development](#), [Product Manager](#), [Design](#), [Business/Marketing](#)); - [Goal Stage](#) — визначення пошукових переваг (які спеціалізації шукає користувач для своєї команди).

Рис. 2.5. Онбординг: Вибір спеціалізації Стан онбордингу зберігається у підколекції [onboardStatus](#) документа користувача у [Firestore](#). Це дозволяє відновлювати процес онбордингу після виходу з додатку. Онбординг включає проміжні мотиваційні екрани ([overlays](#)): [PerfectChoiceOverlay](#) — після вибору спеціалізації з підтримуючим повідомленням; [ActiveCommunityOverlay](#) — інформація про активну спільноту; [ProcessingOverlay](#) — анімація обробки даних перед переходом до основного інтерфейсу. Прогрес онбордингу відображається через [StepProgressIndicator](#) — візуальний індикатор поточного етапу. Це допомагає користувачу зрозуміти, скільки етапів залишилося, і зменшує ймовірність передчасного виходу. Дані, зібрані під час онбордингу, використовуються у трьох контекстах: для відображення на картці профілю (ім'я, фото, спеціалізація); для алгоритму матчингу (досвід, навички, [aboutMe](#)); для фільтрації кандидатів ([searchingSpecializations](#)).

Рис. 2.6. Етапи онбордингу [Synergy Hub](#)

2.8. Проєктування механізму пошуку та свайп-системи Свайп-система є центральним елементом взаємодії користувача з додатком. Механізм пошуку побудовано на принципі послідовного перегляду карток профілів з можливістю прийняття рішення за допомогою жестів. Проєктні рішення: - стек карток — одночасно відображається поточна картка та попередня для створення ефекту глибини; - жести взаємодії — обробка [drag](#)-жестів з розрахунком зміщення та кута повороту картки; - порогове значення — для фіксації рішення зміщення має перевищити 75 пікселів по горизонталі; - візуальний відгук — індикація лайку або дизлайку на картці під час свайпу; - фільтрація — виключення вже переглянутих профілів з подальших показів.

Рис. 2.7. [Discover](#)-екран зі свайп-картками Керування станом свайп-картки реалізовано через [TinderCardCubit](#), який обробляє події [onPanStart](#), [onPanUpdate](#) та [onPanEnd](#) для відстеження позиції та визначення статусу ([liked](#), [reject](#), [none](#)). Анімація картки включає два компоненти: горизонтальне зміщення ([translate](#)) та обертання ([rotate](#)). Кут обертання розраховується пропорційно горизонтальному зміщенню з максимумом 45 градусів. При відпусканні картки, якщо зміщення не досягло порогового значення, картка плавно повертається на початкову позицію з використанням пружинної анімації. Для оптимізації завантаження кандидатів використовується пагінація. [PeopleDataBloc](#) завантажує кандидатів пакетами та запитує наступний пакет, коли поточний наближається до кінця. [GetExcludedMatchesUseCase](#) забезпечує виключення вже переглянутих профілів з наступних запитів.

Рис. 2.8. Макет свайп-екрану з індикацією рішення

2.9. Проєктування системи чатів та повідомлень Система чатів спроектована для забезпечення обміну повідомленнями між користувачами, що здійснили взаємний матч. Архітектура чатів базується на реальночасовій синхронізації через [Firebase Firestore](#). Модель чату включає такі поля: - [chatId](#) — унікальний ідентифікатор чату; - [participants](#) — масив [UID](#) учасників; - [participantInfo](#) — мапа інформації про учасників (ім'я, аватар, спеціалізація); - [lastMessage](#), [lastMessageSent](#) — інформація про останнє повідомлення; -

[typingUsers](#) — масив [UID](#) користувачів, які зараз набирають повідомлення; — [searchKeywords](#) — ключові слова для пошуку чату; — [config](#) — налаштування (архівовані, улюблені чати). Рис. 2.9. Список чатів та матчів. Повідомлення зберігаються у вкладеній підколекції [messages](#) кожного чату. Підписка на оновлення реалізується через [Firestore Snapshot Listeners](#), що забезпечує миттєву доставку нових повідомлень без необхідності політінгу. Синхронізація профілю у чатах забезпечується [Cloud Function onUserUpdate](#), яка автоматично оновлює [participantInfo](#) у всіх чатах користувача при зміні його імені, аватару чи спеціалізації. Архітектурно система чатів розроблена з урахуванням можливості майбутнього розширення. Поточна реалізація підтримує лише текстові повідомлення, проте структура моделі [ChatMessage](#) дозволяє додати підтримку зображень, файлів та голосових повідомлень без зміни архітектури. Для забезпечення консистентності даних при одночасному оновленні чату декількома учасниками використовуються атомарні операції [Firestore: arrayUnion](#) та [arrayRemove](#) для масивів ([typingUsers](#)), [FieldValue.serverTimestamp](#) для серверного часу ([lastMessageSent](#)). Рис. 2.10. Діаграма послідовності обміну повідомленнями. Висновки до розділу У даному розділі було здійснено проектування мобільного застосунку [Synergy Hub](#), спрямованого на ефективне формування стартап-команд та забезпечення взаємодії між користувачами. На основі аналізу було обґрунтовано вибір сучасного технологічного стеку, зокрема [Flutter](#) та [Dart](#) для клієнтської частини, [Firebase](#) як хмарної платформи [Backend-as-a-Service](#), а також [TypeScript](#) для реалізації серверної логіки у [Cloud Functions](#). Було визначено основні бібліотеки та інструменти, які забезпечують стабільність, масштабованість та підтримуваність системи, включаючи засоби кодогенерації, функціонального та реактивного програмування, а також механізми контролю якості коду. Це дозволило зменшити обсяг шаблонного коду та підвищити надійність програмного продукту. Розроблено загальну архітектуру системи, яка базується на підході [Clean Architecture](#) та принципах модульності. Система поділена на клієнтську та серверну частини, між якими організовано взаємодію через [Firestore](#), [Cloud Functions](#) та [Firebase Cloud Messaging](#). Такий підхід забезпечує масштабованість, гнучкість та легкість подальшого розвитку системи. Детально описано реалізацію багаторівневої архітектури ([Presentation, Domain, Data](#)), яка дозволяє чітко розмежувати відповідальності компонентів та забезпечує незалежність бізнес-логіки від зовнішніх технологічних залежностей. Використання патернів [Repository](#), [BLoC](#) та [Dependency Injection](#) підвищує тестованість і підтримуваність системи. Було спроектовано структуру бази даних на основі [Firebase Firestore](#), визначено основні колекції, підколекції та принципи зберігання даних. Особливу увагу приділено механізмам забезпечення цілісності даних, використанню транзакцій, індексації та [real-time](#) синхронізації. Описано модель взаємодії користувачів у системі, яка базується на принципі взаємного матчінгу, а також алгоритми розрахунку сумісності між користувачами за трьома ключовими параметрами: досвід, навички та бачення. Це забезпечує більш якісний підбір потенційних учасників стартап-команд. Окремо було розглянуто проектування ключових функціональних модулів застосунку, зокрема системи онбордингу, свайп-механізму, пошуку, чатів та повідомлень. Кожен із цих компонентів інтегрований у загальну архітектуру системи та забезпечує зручну та інтуїтивну взаємодію користувача із застосунком. Отримані результати підтверджують правильність вибраного архітектурного підходу та технологічних рішень, а також створюють основу для подальшої реалізації мобільного застосунку [Synergy Hub](#), включаючи розробку бізнес-логіки, інтерфейсу користувача та серверної взаємодії. РОЗДІЛ 3. РЕАЛІЗАЦІЯ МОБІЛЬНОГО ДОДАТКА 3.1. Архітектура реалізації клієнтської частини на [Flutter](#) та [Dart](#) Клієнтська частина застосунку організована за фічерним принципом, де кожна функціональна область ізольована у власному модулі з дотриманням [Clean Architecture](#). Основні фічерні модулі: — [auth](#) — реєстрація та авторизація; — [onboard_survey](#) — онбординг та збір інформації; — [matches](#) — свайп-система та матчінг; — [chat](#) — обмін повідомленнями; — [user](#) — керування профілем; — [notifications](#) — сповіщення; — [home](#) — головна навігація та вкладки. Кожен модуль містить три рівні: — [domain/](#) — моделі, інтерфейси репозиторіїв, [Use Cases](#); — [data/](#) — [DTO](#), реалізації репозиторіїв; — [presentation/](#) — [BLoC/Cubit](#), сторінки, віджети. Для генерації коду використовуються: [Freezed](#) — для створення імутабельних [data](#)-класів з підтримкою [copyWith](#) та [pattern matching](#); [JsonSerializable](#) — для автоматичної серіалізації/десеріалізації [JSON](#); [go_router_builder](#) — для типобезпечної генерації маршрутів. Для керування станом застосовується патерн [BLoC](#) з використанням бібліотеки [flutter_bloc](#). [BLoC](#) отримує події ([Events](#)), обробляє їх та емітує нові стани ([States](#)). Для простих випадків використовується [Cubit](#) — спрощена версія [BLoC](#) без подій. Приклад структури [BLoC](#)-подій для модуля автентифікації: [@freezed sealed class](#)

`AuthEvent { const factory AuthEvent.init() = _Init; const factory AuthEvent.listen() = _Listen; const factory AuthEvent.signInViaGoogle() = _SignInViaGoogle; const factory AuthEvent.signOut() = _SignOut; }` BLoC-компоненти підключаються до дерева віджетів через `MultiBlocProvider` у кореневому віджеті додатку. Це забезпечує доступність станів автентифікації, користувача та навігації на всіх рівнях ієрархії: `MultiBlocProvider( providers: [ BlocProvider UserBloc .value( value: userBloc, ), BlocProvider AuthBloc .value( value: authBloc, ), ], child: MaterialApp.router(...) )` Для спостереження за станом BLoC-компонентів під час розробки реалізовано `AppObserver`, який логує події створення, зміни стану та виникнення помилок. Це значно спрощує процес налагодження та дозволяє відслідковувати послідовність подій. Моделі даних генеруються через `Freezed`, що забезпечує: імутабельність (всі поля `final`); автоматичну реалізацію `equality` (`==` та `hashCode`); метод `copyWith` для створення модифікованих копій; підтримку JSON серіалізації через `@JsonSerializable`; `sealed classes` для вичерпного `pattern matching`. Приклад моделі `UserData`: `@freezed sealed class UserData { const factory UserData({ required String uid, String? avatarUrl, String? firstName, String? lastName, int? age, String? aboutMe, required UserProfileer profiler, @TimestampConverter() required DateTime createdAt, }) = _UserData; }` Рис. 3.1. Профіль користувача 3.2.

Використання патерну `Repository`, `Unit of Work` та `Dependency Injection Repository Pattern`. Патерн `Repository` забезпечує абстрагування доступу до даних. У `Domain Layer` визначені інтерфейси (`IAuthRepository`, `IChatRepository`, `IMatchesRepository`, `IUserRepository`), а в `Data Layer` — їх реалізації з використанням `Firebase`. `Unit of Work`. На сервері (`TypeScript`) реалізовано патерн `Unit of Work` через клас `FirebaseUnitOfWork`, який обертає всі операції в єдину `Firestore`-транзакцію. Це гарантує атомарність операцій матчингу: `async run T (fn: (ctx: ITransactionContext) = Promise T) { return firestore().runTransaction(async (tx) = { const ctx: ITransactionContext = { matches: this.factories.matchesRepo(tx), notifications: this.factories.notificationsRepo(tx), chats: this.factories.chatsRepo(tx), userData: this.factories.userDataRepo(tx), }; return fn(ctx); }); }` `Dependency Injection`. На клієнті використовується `GetIt (Service Locator)` для керування залежностями. Ініціалізація виконується при запуску додатку: `final getIt = GetIt.instance; Future void initDependencies() async { getIt.registerLazySingleton(() = FirebaseFirestoreService(),); initAuthFeature(getIt); initMatchingFeature(getIt); initChatFeature(getIt); }` На сервері DI реалізовано через клас `Container`, який створює та зв'язує всі залежності: репозиторії, сервіси, `Use Cases` та `Unit of Work`. Ключовою особливістю серверного DI є фабричний метод для створення транзакційних репозиторіїв. Кожен репозиторій приймає опціональний параметр `Transaction`, що дозволяє використовувати один і той самий репозиторій як у транзакційному, так і в нетранзакційному контексті: `chatRepo = (tx?: Transaction) = new ChatFirebaseRepository(firestore, participantIdentifier, chatKeywordGenerator, tx);` Це забезпечує гнучкість використання: для операцій матчингу репозиторії створюються з транзакцією (для атомарності), а для операцій синхронізації профілю — без транзакції (для кращої продуктивності). На клієнті кожен фічерний модуль має власну функцію ініціалізації DI (наприклад, `initMatchingFeature`, `initChatFeature`), яка реєструє всі залежності модуля в контейнері `GetIt`. Це забезпечує чітке розмежування відповідальності та полегшує рефакторинг.

3.3. Реалізація анімованого фону з використанням GLSL-шейдерів

Для створення візуально привабливого анімованого фону в застосунку використовуються GLSL-шейдери (`OpenGL Shading Language`), інтегровані через бібліотеку `flutter_shaders`. Шейдери виконуються безпосередньо на GPU, що забезпечує плавну анімацію без навантаження на основний потік. Основний шейдер `animated_gradient.frag` реалізує складний анімований градієнт з такими ефектами: – множинні хвильові рухи на різних швидкостях для створення живого ефекту; – палітра кольорів `Coral/Salmon/Peach` для фірмового стилю; – пульсуючий ефект від центру з використанням функції `length()`; – процедурний шум для текстурності; – динамічна зміна яскравості. Фрагмент основної функції шейдера: `vec3 getGradientColor(vec2 uv, float time) { float wave1 = sin(uv.x * 3.0 + time * 0.8) * 0.5 + 0.5; float wave2 = cos(uv.y * 3.5 + time * 0.6) * 0.5 + 0.5; vec3 color1 = vec3(0.95, 0.45, 0.38); // Coral vec3 color2 = vec3(0.98, 0.62, 0.45); // Orange vec3 mixColor = mix(color1, color2, wave1); return mixColor; }` Шейдер приймає два `uniform`-параметри: `uSize` (розмір екрану) та `uTime` (поточний час для анімації). Відображення шейдера здійснюється через `CustomPaint` з використанням `ShaderPainter`, який передає параметри у фрагментний шейдер кожного кадру. Окрім `animated_gradient.frag`, у проєкті використовуються додаткові шейдери: – `check_pulse.frag` — анімація пульсуючої позначки, що використовується при завершенні онбордингу для візуального зворотного зв'язку; – `gradient_smooth.frag` — плавний градієнтний ефект для фону окремих екранів; – `ve.frag` —

розширений градієнтний ефект з додатковим змішуванням кольорів. Всі шейдери декларуються у `pubspec.yaml` в секції `flutter.shaders` та компілюються разом з додатком. `Flutter` забезпечує кросплатформну підтримку шейдерів: на мобільних платформах використовується `Impeller` (або `Skia`), на веб-платформі — `WebGL`. Інтеграція шейдера з `Flutter`-віджетом реалізована через ланцюжок: завантаження шейдера через `FragmentProgram.fromAsset()` → створення `FragmentShader` → передача `uniform`-параметрів → відображення через `CustomPaint` з `ShaderPainter`. Анімація забезпечується через `AnimationController`, який оновлює параметр `uTime` кожного кадру. Використання `GPU`-шейдерів замість стандартних `Flutter`-анімацій забезпечує значно вищу продуктивність, оскільки обчислення виконуються паралельно на графічному процесорі, не навантажуючи основний `UI`-потік. Рис. 3.2. `GLSL` Анімований фон сторінки з результатами матчингу 3.4.

Реалізація системи автентифікації користувачів Система автентифікації `Synergy Hub` підтримує два методи входу: електронна пошта з паролем та `Google OAuth`. Для веб-платформи реалізовано окрему імплементацію `Google Sign-In`. Архітектура автентифікації включає: - `IAuthRepository` — інтерфейс у `Domain Layer`; - `FirebaseAuthRepository` — реалізація з використанням `Firebase Auth`; - `AuthBloc` — `BLoC` для керування станом автентифікації; - `RegisterUseCase`, `LoginUseCase` — сценарії використання. `Firebase Auth` забезпечує: створення облікових записів з `email` та паролем; автентифікацію через `Google OAuth`; управління сесіями; потік даних про стан авторизації через `watchUser()`. Для обробки помилок використовується функціональний тип `Either` з бібліотеки `dartz`, що дозволяє явно моделювати успішний та помилковий результат: - `UnauthorizedFailure` — відсутність авторизації; - `UserExistsWithSuchEmailFailure` — обліковий запис вже існує; - `LoginOrPasswordIncorrectFailure` — невірні облікові дані; - `MaxTriesExceededException` — перевищено кількість спроб. `Google Sign-In` реалізовано з урахуванням платформних відмінностей через умовний імпорт: `import 'src/google_sign_in_service_mobile.dart' if (dart.library.html) 'src/google_sign_in_service_web.dart'`; Умовний імпорт у `Dart` дозволяє обирати різні реалізації залежно від платформи на етапі компіляції. Для мобільних платформ (`Android`, `iOS`) використовується пакет `google_sign_in`, який взаємодіє з нативним `Google Sign-In SDK`. Для веб-платформи використовується `google_sign_in_web`, який працює через `JavaScript Google Identity Services API`. `AuthBloc` прослуховує зміни стану автентифікації через потік `watchUser()`. При успішному вході `BLoC` перевіряє наявність заповненого профілю (через `GetCompletionStatusUseCase`). Якщо профіль не заповнений, користувач перенаправляється на онбординг. Якщо профіль заповнений — на головний екран. Безпека автентифікації забезпечується на кількох рівнях: `Firebase Auth` виконує валідацію облікових даних та генерацію `JWT`-токенів; `Firestore Security Rules` перевіряють автентифікацію при кожному запиті до бази даних; `Cloud Functions` перевіряють наявність `auth`-контексту перед виконанням операцій. Для обробки помилок автентифікації визначено ієрархію типів відмов (`Failure`). Кожен тип відмови містить повідомлення, яке може бути відображено користувачу. Це забезпечує детальний зворотний зв'язок при помилках входу або реєстрації, що покращує користувацький досвід. Рис. 3.3. Екран авторизації 3.5. Реалізація `Discover`-екрану та механізму свайпів `Discover`-екран є центральним елементом взаємодії користувача з додатком. Він реалізований за допомогою двох `BLoC`-компонентів: - `PeopleDataBloc` — завантаження та фільтрація кандидатів для перегляду; - `TinderCardCubit` — керування жестами та анімацією картки. `TinderCardCubit` обробляє три типи жестів: - `onPanStart` — початок перетягування; - `onPanUpdate` — оновлення позиції з розрахунком кута повороту; - `onPanEnd` — визначення результату за пороговим значенням (75px). Визначення статусу свайпу: `TinderCardStatus _getStatus(Offset position) { const offset = 75.0; if (position.dx > offset) return TinderCardStatus.liked; if (position.dx < -offset) return TinderCardStatus.reject; return TinderCardStatus.none; }` Після визначення рішення викликається `Cloud Function matchUser`, яка обробляє лайк або дизлайк у транзакції `Firestore`. Для запобігання повторного показу профілів використовується `GetExcludedMatchesUseCase`, який отримує список вже переглянутих `UID`. Архітектура `Discover`-екрану побудована на принципі стеку карток (`card stack`). На екрані одночасно присутні дві картки: верхня (поточна, з якою взаємодіє користувач) та нижня (наступна, частково видима для створення ефекту глибини). Після свайпу поточної картки наступна стає поточною, а з сервера завантажуються новий кандидат. Візуальна індикація рішення реалізована через `BuildTinderStatus` — віджет, який відображає напис

(зелений) або

Цитування: 0%

id: 16

"NOPE"

(червоний) з прозорістю, пропорційною зміщенню картки. Це забезпечує інтуїтивний зворотний зв'язок ще до відпускання картки. Фільтрація кандидатів здійснюється через модель `MatchesFilter`: `const factory MatchesFilter({ EExperience? experience, ESpecialization? specialization, @Default([]) List Skill skills, })` 3.6. Реалізація чатів та збереження повідомлень

Система чатів реалізована з використанням реальночасових `Firestore Snapshot Listeners`. Кожен чат зберігається як документ у колекції `chats`, а повідомлення — у вкладеній підколекції `messages`. Архітектура чатів включає: — `ChatBloc` — керування станом окремого чату (завантаження повідомлень, відправка, індикація набору тексту); — `ChatsFilterBloc` — фільтрація списку чатів (усі, улюблені, архівовані); — `IChatRepository` — інтерфейс для операцій з чатами. `Use Cases` для чатів: — `ListenChatsUseCase` — підписка на потік усіх чатів користувача; — `ListenChatByIdUseCase` — підписка на оновлення конкретного чату; — `UpdateTypingInfoUseCase` — трансляція статусу набору тексту; — `ToggleFavoriteChatUseCase` — управління улюбленими чатами; — `SearchChatsUseCase` — пошук чатів за ключовими словами. Підписка на чати реалізується через `emit.forEach` у `BLoC`, що автоматично оновлює стан при надходженні нових даних з `Firestore`: `await emit.forEach(chatRepository.getStream(uid), onData: (chats) = state.copyWith( status: ChatStatus.loaded, chats: chats, ));` Функціональність індикації набору тексту (`typing indicator`) реалізована через поле `typingUsers` у документі чату. При початку набору тексту `UID` користувача додається до масиву, при завершенні — видаляється. Оскільки `Firestore` забезпечує реальночасове оновлення, інші учасники чату миттєво бачать індикатор набору. Система пошуку чатів базується на полі `searchKeywords`, яке містить посимвольні префікси імен учасників. Це дозволяє реалізувати пошук з автодоповненням без використання зовнішніх пошукових сервісів (`Algolia`, `ElasticSearch`), що зменшує складність та вартість інфраструктури. Конфігурація чату (`ChatConfig`) дозволяє кожному учаснику індивідуально архівувати або додавати чат до улюблених. Дані зберігаються у масивах `archived` та `favorite` всередині об'єкту `config` документа чату. Модель повідомлення `ChatMessage` включає поля: `id` (унікальний ідентифікатор), `message` (текст повідомлення), `sentUid` (`UID` відправника) та `createdAt` (час створення). Повідомлення сортуються за часом створення для відображення у хронологічному порядку. Рис 3.4. Екран чату з повідомленнями 3.7.

Реалізація бізнес-логіки на основі патерну `Use Case Use Case` (сценарій використання) є ключовим елементом `Domain Layer`, що інкапсулює одну конкретну бізнес-операцію. У застосунку `Synergy Hub Use Cases` використовуються як на клієнті (`Dart`), так і на сервері (`TypeScript`). Клієнтські `Use Cases` (`Dart`). Кожен `Use Case` залежить від інтерфейсів репозиторіїв та не має прямих залежностей від `Firestore` чи `Flutter`. Приклади: — `LookForMatchesUseCase` — пошук кандидатів з урахуванням фільтрів та виключенням переглянутих; — `CreateUserMatchesUseCase` — фіксація рішення через виклик `Cloud Function`; — `ListenTwoWayMatchUseCase` — підписка на потік взаємних матчів; — `ListenChatsUseCase` — підписка на потік чатів; — `GetCompletionStatusUseCase` — перевірка стану онбордингу. Серверний `Use Cases` (`TypeScript`). На сервері `Use Cases` отримують `ITransactionContext` для виконання операцій у транзакції: — `MatchUseCase` — обробка лайку/дизлайку в транзакції; — `ProcessTwoWayMatchesUseCase` — створення чату та розрахунок сумісності; — `OnNewMessageUseCase` — обробка нового повідомлення; — `SyncUserProfileInChatsUseCase` — синхронізація профілю; — `SendNotificationsUseCase` — відправка `push`-сповіщень. Алгоритми оцінки сумісності реалізовані як окремі `Use Cases`: — `CalculateExperienceScoreUseCase` — порівняння рівнів досвіду; — `CalculateSkillScoreUseCase` — обчислення перетину навичок; — `CalculateVisionAlignUseCase` — текстовий аналіз полів `aboutMe`. Кожен `Use Case` дотримується принципу єдиної відповідальності (`Single Responsibility Principle`): один `Use Case` виконує одну бізнес-операцію. Це спрощує тестування, оскільки для кожного `Use Case` можна написати ізольовані юніт-тести з моками залежностей. Патерн `Use Case` також забезпечує чітку точку входу для кожної бізнес-операції. `BLoC`-компоненти не працюють напряму з репозиторіями, а виключно через `Use Cases`. Це додає додатковий рівень абстракції, який дозволяє змінювати логіку операції (наприклад, додати валідацію або кешування) без модифікації `Presentation Layer`. Обробка помилок у `Use Cases` реалізована через тип `Either Failure, Result` з бібліотеки `dartz`. `Left` містить об'єкт `Failure` з описом помилки, `Right` — успішний результат. Це дозволяє `BLoC`-компонентам елегантно обробляти обидва випадки: `final result = await loginUseCase.call( email: email, password: password, );`

`result.fold( (failure) = emit(state.copyWith( status: AuthStatus.error, errorMessage: failure.message, )), (_) = emit(state.copyWith( status: AuthStatus.authorized, )), );` 3.8.

Інтеграція з [Firebase Firestore](#) та [Firestore Security Rules](#). Взаємодія з [Firebase Firestore](#) реалізована через набір сервісів, інкапсульованих у [core/services/firebase/](#). Основним є [FirestoreService](#), який надає типізовані посилання на колекції та документи. Для кожної колекції визначено типізовані псевдоніми: `typedef DocRef = DocumentReference Map String, dynamic ; typedef ColRef = CollectionReference Map String, dynamic ; typedef QuerySnap = QuerySnapshot Map String, dynamic ;` [Firestore Security Rules](#) забезпечують безпеку даних на рівні бази. Основні правила: – дані користувача доступні для читання та запису лише самому користувачу; – чати доступні лише учасникам (перевірка наявності `UID` у масиві `participants`); – повідомлення у чаті доступні лише учасникам чату; – матчі доступні лише залученим користувачам; – публічні дані профілю доступні для читання авторизованим користувачам. Для оптимізації продуктивності використовуються складені індекси [Firestore](#) для запитів з фільтрацією та сортуванням. Кешування на клієнті забезпечується вбудованим механізмом [Firestore offline persistence](#). Для підтримки QA-тестування реалізовано підключення до [Firebase Emulator](#): `if (AppConfig.isQA) { await firebaseAuth.useAuthEmulator( AppConfig.emulatorAddress(), AppConfig.authEmulatorPort, ); }` [Firebase Emulator Suite](#) дозволяє запускати локальні версії [Auth](#), [Firestore](#) та [Cloud Functions](#). Це забезпечує можливість тестування без підключення до хмарної інфраструктури, що прискорює цикл розробки та запобігає випадковим змінам у [production](#)-базі. Конвертація типів між [Firestore](#) та [Dart](#)-моделями реалізована через кастомні конвертери. [TimestampConverter](#) автоматично перетворює [Firestore Timestamp](#) у [Dart DateTime](#) та навпаки. [ColorSerializable](#) забезпечує серіалізацію кольорів для збереження налаштувань теми. Оптимізація запитів до [Firestore](#) включає: використання [compound queries](#) з фільтрацією на сервері замість клієнтської фільтрації; обмеження кількості документів через `limit()` для пагінації; використання `orderBy()` для сортування на рівні бази даних; кешування результатів через вбудований механізм [Firestore offline persistence](#). Взаємодія з [Firestore](#) у клієнтській частині здійснюється виключно через репозиторії [Data Layer](#). Кожен репозиторій інкапсулює логіку конвертації [DTO](#) у доменні моделі та навпаки, обробку помилок та управління підписками на [real-time](#) оновлення.

3.9. Реалізація серверної логіки на основі [Firebase Cloud Functions \(TypeScript\)](#)

Серверна логіка застосунку реалізована у вигляді трьох [Firebase Cloud Functions](#) на [TypeScript](#). Серверний проєкт використовує [Node.js 20](#) та дотримується принципів [Clean Architecture](#) з [Dependency Injection](#).

3.9.1. Функція матчінгу користувачів у транзакції (`matchUser`)

Функція `matchUser` є [Callable HTTPS Function](#), яка викликається з клієнта для обробки рішення користувача (лайк або дизлайк). Усі операції виконуються в рамках єдиної [Firestore](#)-транзакції через патерн [Unit of Work](#). Алгоритм роботи: – перевірка автентифікації та валідація вхідних даних; – запобігання самотатчингу (`userId == matchId`); – генерація унікального ключа матчу через сортовану конкатенацію `UID`; – обробка дизлайку — оновлення запису з `completeness: none`; – перевірка наявності зустрічного лайку; – при взаємному лайку — виклик [ProcessTwoWayMatchesUseCase](#); – при односторонньому лайку — створення нового запису та відправка сповіщення.

Фрагмент точки входу [Cloud Function](#): `export const matchUser = onCall( async (req) = { const { data, auth } = req; if (!auth) throw new HttpsError( 'unauthenticated', 'User must be authenticated' ); await container.matchUseCase.call( auth.uid, data.matchId, data.status ); return { success: true }; } );` При двосторонньому матчі [ProcessTwoWayMatchesUseCase](#) виконує: – завантаження профілів обох користувачів; – розрахунок [Experience Score](#) (`max(10 - diff × 3, 2)`); – розрахунок [Skills Score](#) (перетин навичок / максимум); – розрахунок [Vision Score](#) (текстовий аналіз полів `aboutMe`); – створення чату між користувачами; – відправку `push`-сповіщень обох учасникам.

Повний лістинг функції `matchUser` наведено у Додатку А. Алгоритм розрахунку [Experience Score](#) базується на відстані між рівнями досвіду. Рівні впорядковані: [Junior](#) (0), [Middle](#) (1), [Senior](#) (2), [Lead/Expert](#) (3). Формула: $score = \max(10 - |pos_a - pos_b| \times 3, 2)$. Наприклад, два [Senior](#)-розробники отримують максимальний бал 10, а пара [Junior-Senior](#) — бал 4. Алгоритм [Skills Score](#) обчислює перетин множин навичок: $intersection = skills_a \cap skills_b$; $score = (|intersection| / \max(|skills_a|, |skills_b|)) \times 10$. Наприклад, якщо у першого користувача навички [[Flutter](#), [Firebase](#), [Dart](#)], а у другого [[Flutter](#), [React](#), [Dart](#)], перетин = [[Flutter](#), [Dart](#)], $score = 2/3 \times 10 \approx 6.7$.

[Vision Score](#) використовує текстовий аналіз полів `aboutMe`. Алгоритм: нормалізація тексту (`lowercase`, видалення пунктуації) → розбиття на слова → фільтрація стоп-слів (`a, the, and, startup, project` тощо) → фільтрація коротких слів (менше 3 символів) → обчислення подібності як відношення перетину до максимуму $\times 10$.

Рис. 3.5. Екран

взаємного матчу 3.9.2. Обробка нових повідомлень та розсилка [push](#)-сповіщень ([onNewMessage](#)) Функція [onNewMessage](#) є [Firestore Trigger](#), що активується при створенні нового документа за шляхом [chats/{chatId}/messages/{msgId}](#). Вона забезпечує відправку [push](#)-сповіщень усім учасникам чату, окрім відправника повідомлення. Алгоритм роботи: – витягнення [chatId](#) та даних повідомлення ([sentUid](#), [message](#)); – завантаження документа чату для отримання списку учасників; – фільтрація — виключення відправника зі списку отримувачів; – для кожного отримувача: завантаження профілю, отримання [FCM](#)-токенів, відправка сповіщення; – збереження сповіщення у [Firestore](#) (історія сповіщень). Тіло сповіщення містить ім'я відправника та текст повідомлення. Для [Android](#) встановлюється пріоритет [high](#), для [iOS](#) — стандартний звук [default](#). Сповіщення включає [payload](#) з метаданими: [chatId](#) (для навігації до відповідного чату), [sentUid](#) (для ідентифікації відправника) та [type](#):

Цитування: 0%

id: 17

“chat”

(для визначення типу сповіщення на клієнт). Клієнтський застосунок використовує ці метадані для навігації користувача безпосередньо до чату при натисканні на сповіщення. Окрім [push](#)-сповіщення, [OnNewMessageUseCase](#) також зберігає запис у колекції [notifications](#) кожного отримувача. Це забезпечує історію сповіщень навіть у випадку недоставки [push](#)-повідомлення (наприклад, коли [FCM](#)-токен застарів або пристрій не підключений до мережі). Продуктивність функції оптимізована через паралельне завантаження профілів та [FCM](#)-конфігурацій учасників. Для чатів з двома учасниками (поточна реалізація) це означає лише одне додаткове читання з бази даних, оскільки відправник виключається зі списку отримувачів. 3.9.3. Синхронізація профілю користувача в чатах ([onUserUpdate](#)) Функція [onUserUpdate](#) є [Firestore Trigger](#) на шлях [users/{userId}](#), який активується при оновленні документа користувача. Її призначення — синхронізація інформації профілю у всіх чатах, де бере участь користувач. Функція порівнює стан [before](#) та [after](#) оновлення. Якщо змінилися [firstName](#), [lastName](#) або [avatarUrl](#), виконується оновлення [participantInfo](#) у кожному чаті користувача через [SyncUserProfileInChatsUseCase](#). Оновлення використовує [nested field path](#) для атомарної зміни: [const fieldPath = `participantInfo.\\${userId}`](#); [ref.update\({ \[fieldPath\]: { name: newName, specialization: newSpecialization, avatarUrl: avatarUrl, } }\)](#); 3.10. Інтеграція [Firebase Cloud Messaging](#) для [push](#)-сповіщень [Push](#)-сповіщення у [Synergy Hub](#) реалізовані через [Firebase Cloud Messaging](#) ([FCM](#)). Інтеграція включає клієнтську та серверну частини. Клієнтська частина. [FirebaseMessagingService](#) відповідає за: – запит дозволу на сповіщення ([iOS](#)); – отримання та збереження [FCM](#)-токена пристрою; – обробку [foreground](#)-повідомлень через [onMessage](#); – реєстрацію [background handler](#) для повідомлень у фоновому режимі; – інтеграцію з [LocalNotificationService](#) для відображення локальних сповіщень. Для веб-платформи додатково використовується [VAPID](#)-ключ для отримання [push](#)-сертифікату. Серверна частина. Відправка сповіщень здійснюється через [SendNotificationsUseCase](#), який використовує [Firebase Admin SDK](#) ([FCMService](#)). Кожне сповіщення включає: – [title](#) та [body](#) — заголовок та текст; – [data](#) — [payload](#) з типом події ([match](#) або [chat](#)) та ідентифікаторами; – платформні налаштування ([Android](#): [high priority](#), [iOS](#): [default sound](#)). Сповіщення також зберігаються у [Firestore](#) (колекція [notifications](#) користувача), що забезпечує історію сповіщень навіть при недоставці [push](#)-повідомлення. Конфігурація сповіщень зберігається у підколекції [notificationConfig](#) кожного користувача. [NotificationConfig](#) містить: [enabledNotifications](#) (прапорець увімкнення/вимкнення сповіщень) та [fcmTokens](#) (масив токенів пристроїв). Один користувач може мати кілька [FCM](#)-токенів, якщо використовує додаток на різних пристроях або платформах. Обробка [foreground](#)-повідомлень на клієнті реалізована через [FirebaseMessaging.onMessage stream](#). При отриманні повідомлення в [foreground](#)-режимі створюється локальне сповіщення через [flutter_local_notifications](#), оскільки [FCM](#) за замовчуванням не відображає сповіщення, коли додаток активний. [Background handler](#) реєструється одноразово при ініціалізації додатку через [FirebaseMessaging.onBackgroundMessage](#). Ця функція виконується в окремому ізоляті ([isolate](#)) [Dart](#), що вимагає мінімальної ініціалізації [Firebase](#) без доступу до стану додатку. Для веб-платформи [push](#)-сповіщення обробляються через [Service Worker](#). [FCM Web SDK](#) автоматично реєструє [Service Worker](#), який отримує повідомлення навіть коли вкладка браузера неактивна. [VAPID](#)-ключ ([Voluntary Application Server Identification](#)) використовується для ідентифікації сервера при відправці [push](#)-повідомлень. Рис 3.6. [push](#)-сповіщення отримане з чату Висновки до розділу У даному розділі було представлено

реалізацію мобільного застосунку [Synergy Hub](#), побудованого на основі сучасних архітектурних підходів та технологій [Flutter](#) і [Dart](#). Детально описано клієнтську частину системи, яка організована за принципами [Clean Architecture](#) та фічерної модульності, що забезпечує ізолюваність функціональних компонентів, простоту масштабування та подальшого супроводу проєкту. Використання [BLoC/Cubit](#) як основного підходу до керування станом дозволило чітко розділити бізнес-логіку та [UI](#), а застосування генераторів коду ([Freezed](#), [JsonSerializable](#), [go_router_builder](#)) підвищило надійність та зменшило обсяг ручного коду. Окрему увагу приділено реалізації ключових підсистем застосунку, зокрема автентифікації користувачів, механізму матчингу, чатів у режимі реального часу, а також [Discover](#)-екрану зі свайп-взаємодією. Продемонстровано використання патерну [Repository](#), [Use Case](#) та [Dependency Injection](#) ([GetIt](#)), що забезпечило чітке розмежування шарів системи та інверсію залежностей. Серверна логіка реалізована через [Firebase Cloud Functions](#) із використанням транзакційного підходу ([Unit of Work](#)), що гарантує цілісність даних при виконанні критичних операцій. Додатково реалізовано механізми [push](#)-сповіщень через [Firebase Cloud Messaging](#), інтеграцію з [Firestore](#) у режимі реального часу, а також систему безпеки на рівні [Firestore Security Rules](#). Візуальна частина застосунку доповнена [GPU](#)-оптимізованими [GLSL](#)-шейдерами, що забезпечують плавну анімацію та покращують користувацький досвід без додаткового навантаження на основний потік. ВИСНОВКИ У ході виконання кваліфікаційної роботи було розроблено кросплатформенний мобільний застосунок [Synergy Hub](#) для формування стартап-команд. Під час роботи було виконано всі поставлені завдання та досягнуто мети дослідження. Основні результати роботи: 1. Проаналізовано предметну область формування стартап-команд та існуючі програмні рішення ([Tinder](#), [LinkedIn](#), [Meetup](#)). Визначено їх основні недоліки: відсутність спеціалізації на стартапах, відсутність алгоритмів оцінки сумісності та обмеженість комунікаційних інструментів. 2. Сформовано вимоги до програмного продукту та обґрунтовано концепцію застосунку [Synergy Hub](#), що поєднує свайп-механізм пошуку, професійний профіль, алгоритм матчингу та інтегровану систему чатів. 3. Спроектовано архітектуру програмного забезпечення на основі [Clean Architecture](#) з розподілом на рівні [Domain](#), [Data](#) та [Presentation](#). Спроектовано базу даних [Firebase Firestore](#) з ієрархічною структурою колекцій. 4. Реалізовано клієнтську частину на [Flutter](#) та [Dart](#) з використанням патерну [BLoC](#) для керування станом, [GoRouter](#) для типобезпечної маршрутизації та [GetIt](#) для впровадження залежностей. Реалізовано анімований фон з використанням [GLSL](#)-шейдерів. 5. Реалізовано серверну логіку на основі [Firebase Cloud Functions](#) ([TypeScript](#)) з трьома [Cloud Functions](#): [matchUser](#) (матчинг у транзакції), [onNewMessage](#) ([push](#)-сповіщення), [onUserUpdate](#) (синхронізація профілю). Застосовано патерн [Unit of Work](#) для забезпечення атомарності транзакцій. 6. Реалізовано алгоритм тривимірної оцінки сумісності учасників за параметрами досвіду, технічних навичок та бачення проєкту. 7. Проведено тестування застосунку на кількох рівнях: юніт-тести серверної логіки ([jest](#)), тестування [UI](#)-компонентів ([flutter_test](#)), ручне тестування на реальних пристроях. Налаштовано [CI/CD](#) через [Codemagic](#) для автоматичного деплою веб-версії на [Firebase Hosting](#). Розроблений мобільний застосунок [Synergy Hub](#) є працездатним програмним продуктом, який може бути використаний для формування стартап-команд та має потенціал для подальшого розвитку та комерціалізації. Застосування сучасних архітектурних підходів ([Clean Architecture](#), [BLoC](#), [Repository Pattern](#), [Unit of Work](#)) забезпечило високу якість кодової бази, її тестованість та можливість масштабування. Використання [Firebase](#) як [Backend-as-a-Service](#) дозволило мінімізувати витрати на інфраструктуру та зосередитися на розробці бізнес-логіки. Ключовою інновацією є тривимірна модель оцінки сумісності учасників ([Experience Score](#), [Skills Score](#), [Vision Score](#)), яка автоматизує процес підбору партнерів та надає кількісні показники якості матчу. Це принципово відрізняє [Synergy Hub](#) від існуючих рішень, які покладаються виключно на ручний підбір. Перспективи подальшого розвитку включають кілька ключових напрямків: Удосконалення алгоритмів матчингу — впровадження машинного навчання для аналізу поведінкових патернів користувачів, використання [NLP](#)-моделей ([sentence embeddings](#)) для семантичного порівняння текстів [Vision Score](#) замість простого перетину слів, а також побудова рекомендаційної системи на основі колаборативної фільтрації. Розширення функціоналу фільтрації — додавання фільтрів за геолокацією (з використанням [GeoFlutterFire](#)), мовами спілкування, доступністю (повна/часткова зайнятість) та розширеною системою тегів. Завдяки обраній архітектурі це потребуватиме змін лише на рівні [Data](#) та [Presentation layers](#) без модифікації серверних [Cloud Functions](#). Розширення платформ — публікація в [Google Play](#) та [App Store](#) після завершення програми закритого

бета-тестування, оптимізація адаптивного макету для планшетів, а також розробка десктопних версій для [macOS](#) та [Windows](#). Масштабування інфраструктури — використання автоматичних механізмів [Firebase](#) (горизонтальне масштабування [Cloud Functions](#), розподіл даних [Firestore](#), [CDN](#) для веб-версії), впровадження [A/B](#) тестування через [Firebase Remote Config](#) та інтеграція [Firebase Analytics](#) для відстеження поведінки користувачів. Монетизація — триступенева модель: [Free tier](#) з обмеженою кількістю свайпів на день, [Premium](#) з необмеженими свайпами, розширеними фільтрами та пріоритетним відображенням профілю, і [Enterprise](#)-план для інкубаторів та акселераторів з окремою панеллю управління.

Заявление об ограничении ответственности:

Этот отчет должен быть правильно истолкован и проанализирован квалифицированным специалистом, который несет ответственность за оценку!

Любая информация, представленная в этом отчете, не является окончательной и подлежит ручному просмотру и анализу. Пожалуйста, следуйте инструкциям: [Рекомендации по оценке](#)

Детектор Плагиата - Ваше право на оригинальность! ☐ SkyLine LLC