

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
Національний університет «Запорізька політехніка»

Факультет комп'ютерних наук і технологій

(повне найменування факультету)

Кафедра програмних засобів

(повне найменування кафедри)

Пояснювальна записка

до дипломного проєкту (роботи)

магістр

(ступінь вищої освіти)

на тему ДОСЛІДЖЕННЯ ТА РОЗРОБКА ОФЛАЙН-ПРІОРИТЕТНОГО
ПРОГРЕСИВНОГО ВЕБЗАСТОСУНКУ
RESEARCH AND DEVELOPMENT OF OFFLINE-FIRST PROGRESSIVE
WEB APPLICATION

Виконав(ла): студент(ка) 2 курсу, групи КНТ-124м
Спеціальності 121 Інженерія програмного
Забезпечення

(код і найменування спеціальності)

Освітня програма (спеціалізація)

Інженерія програмного забезпечення

ТРІНКАЛЬ М.Є.

(ПРІЗВИЩЕ та ініціали)

Керівник ГЛАДКОВА О.М.

(ПРІЗВИЩЕ та ініціали)

Рецензент ДЬЯЧУК Т.С.

(ПРІЗВИЩЕ та ініціали)

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
Національний університет «Запорізька політехніка»

Факультет КНТ
Кафедра програмних засобів
Ступінь вищої освіти магістр
Спеціальність 121 Інженерія програмного забезпечення
(код і найменування)
Освітня програма (спеціалізація) Інженерія програмного забезпечення
(назва освітньої програми (спеціалізації))

ЗАТВЕРДЖУЮ
Завідувач кафедри ПЗ, д.т.н, проф.
Сергій СУББОТІН
“ ” 2025 року

ЗАВДАННЯ
НА ДИПЛОМНИЙ ПРОЄКТ (РОБОТУ) СТУДЕНТА(КИ)

ТРІНКАЛЬ Максима Євгеновича
(ПРИЗВИЩЕ, ім'я, по батькові)

1. Тема проєкту (роботи) Дослідження та розробка офлайн-пріоритетного прогресивного вебзастосунку. Reasearch and Development of Offline-first Progressive Web Application

керівник проєкту (роботи) к.т.н., ГЛАДКОВА Ольга Миколаївна,
(науковий ступінь, вчене звання, ПРИЗВИЩЕ, ім'я, по батькові)

затверджені наказом закладу вищої освіти від “ 30 ” вересня 2025 року № 447

2. Строк подання студентом проєкту (роботи) 08 грудня 2025 року

3. Вихідні дані до проєкту (роботи) рекомендована література, технічне завданн

4. Зміст розрахунково-пояснювальної записки (перелік питань, які потрібно розробити)
1. Постановка завдання дослідження. 2. Огляд існуючих рішень. 3. Інструменти та алгоритми. 4. Опис програмної реалізації. 5. Порівняння PWA з традиційними рішеннями. 6. Робота користувача з розробленим застосунком.

5. Перелік графічного матеріалу (з точним зазначенням обов'язкових креслень)
Слайди презентації

6. Консультанти розділів проєкту (роботи)

Розділ	ПРИЗВИЩЕ, ініціали та посада консультанта	Підпис, дата	
		завдання видав	прийняв виконане завдання
1-5 Основна частина	ГЛАДКОВА О.М., доцент		
Нормоконтролер	БЄЛОВА А.В., асистент		

7. Дата видачі завдання “ 30 ” вересня 2025 року.

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів дипломного проєкту (роботи)	Строк виконання етапів проєкту (роботи)	Примітка
1	Постановка завдання роботи.	1 тиждень	Завдання, ТЗ
2	Аналіз предметної області.	2-3 тиждень	Розділ 1
3	Розробка та удосконалення методів, моделей й алгоритмів вирішення задачі.	4 тиждень	Розділ 2
4	Розробка архітектури програми.	5 тиждень	Розділ 3
5	Розробка програми.	6-7 тижні	Розділ 4
6	Тестування та експериментальне дослідження програмного забезпечення.	8-9 тиждень	Розділ 5, 6
7	Оформлення пояснювальної записки та документів до неї.	10 тиждень	Додатки
8	Нормоконтроль та рецензування.	11 тиждень	
9	Захист роботи.	12 тиждень	

Студент(ка)

_____ Максим ТРІНКАЛЬ
(підпис) (І'мя ПРИЗВИЩЕ)

Керівник проєкту (роботи)

_____ Ольга ГЛАДКОВА
(підпис) (І'мя ПРИЗВИЩЕ)

РЕФЕРАТ

Пояснювальна записка до дипломної кваліфікаційної роботи магістра:
112 с., 2 табл., 36 рис., 2 дод., 27 джерел.

ВЕБЗАСТОСУНОК, ОФЛАЙН-АВТОНОМНІСТЬ, СИНХРОНІЗАЦІЯ
ДАНИХ, МЕРЕЖЕВА ВІДМОВСТІЙКІСТЬ, МОЖЛИВОСТІ БРАУЗЕРА.

Об'єкт дослідження – процеси інженерії програмного забезпечення локального збереження та синхронізації даних у вебзастосунках.

Предмет дослідження – методи та алгоритми інженерії програмного забезпечення офлайн-пріоритетних вебзастосунків, що забезпечують локальне збереження, узгодженість і синхронізацію даних у клієнт-серверній архітектурі з метою зменшення витрат на розробку та підтримку кросплатформених рішень.

Мета роботи – підвищення доступності кросплатформених програмних рішень та скорочення витрат на їх розробку і підтримку шляхом використання офлайн-пріоритетного прогресивного вебзастосунку з єдиною кодовою базою та відкладеною синхронізацією даних.

Матеріали, методи та технічні засоби: функціональне та об'єктно-орієнтоване програмування, алгоритми мережевої синхронізації, MVC, Rest Api, MVVM, React, Express, Node, Docker, Type Script, Java Script, HTML, Less, ноутбук Lenovo Legion 5 з процесором Intel Core i7 під управлінням операційної системи Microsoft Windows 11, та для тестування мобільний пристрій One Plus Nord 2 5G на Android, Google Chrome браузер.

Результати. Розроблено автономний вебзастосунок, що працює на будь якому пристрої з підтримкою браузера, забезпечує миттєву реакцію без очікування відповіді сервера та здійснює відкладену синхронізацію після появи з'єднання. Продемонстровано, що PWA підхід підвищує стійкість до втрати зв'язку, знижує навантаження на сервер, прискорює взаємодію

користувача з застосунком та дозволяє працювати у повністю автономному режимі без потреби кросс-платформенної реалізації застосунку.

Висновки. Модель вебзастосунку забезпечує автономність, високу швидкодію та коректну синхронізацію змін, що робить її придатною для сценаріїв з нестабільним або дорогим доступом до мережі. Робить розробку мобільних застосунків дешевшим для замовника і доступнішим для користувача. Реалізована система підтверджує перспективність PWA підходу як для сучасних застосунків.

Галузь використання – внутрішні корпоративні системи, застосунки для бізнесу з мережевою нестабільністю, будь-які застосунки без необхідності в глибокому доступі до нативних можливостей системи, системи з необхідністю постійних оновлень.

ABSTRACT

Explanatory note to the diploma qualifying work of the master: 112 pages, 2 tables, 36 figures, 2 appendixes, 27 sources.

WEB APPLICATION, OFFLINE AUTONOMY, DATA
SYNCHRONIZATION, NETWORK RESILIENCE, BROWSER
CAPABILITIES.

The object of the research is the software engineering processes of local data storage and synchronization in web applications.

The subject of the research is the methods and algorithms of software engineering of offline-priority web applications that provide local data storage, consistency and synchronization in a client-server architecture in order to reduce the costs of developing and supporting cross-platform solutions.

The purpose of the work is to increase the availability of cross-platform software solutions and reduce the costs of their development and support by using an offline-priority progressive web application with a single code base and deferred data synchronization.

Materials, methods and technical means: functional and object-oriented programming, network synchronization algorithms, MVC, Rest Api, MVVM, React, Express, Node, Docker, Type Script, Java Script, HTML, Less, Lenovo Legion 5 laptop with Intel Core i7 processor running Microsoft Windows 11 operating system, and for testing, One Plus Nord 2 5G mobile device on Android, Google Chrome browser.

Results. An autonomous web application was developed that runs on any device with browser support, provides instant response without waiting for a server response and performs deferred synchronization after a connection appears. It is demonstrated that the PWA approach increases resistance to loss of connection, reduces the load on the server, speeds up user interaction with the application and

allows it to work in a completely autonomous mode without the need for cross-platform implementation of the application.

Conclusions. The web application model provides autonomy, high performance and correct synchronization of changes, which makes it suitable for scenarios with unstable or expensive network access. Makes mobile application development cheaper for the customer and more accessible for the user. The implemented system confirms the prospects of the PWA approach as a basic one for modern applications.

Field of use - internal corporate systems, business applications with network instability, any applications without the need for deep access to the native capabilities of the system, systems with the need for constant updates.

ЗМІСТ

	С.
Зміст	8
Перелік скорочень та умовних познач	11
Вступ.....	12
1 Постановка завдання дослідження.....	14
2 Огляд існуючих рішень.....	16
2.1 Twitter Lite - переваги PWA рішення в порівнянні з традиційними вебзастосунками	16
2.2 Google Maps Web, як альтернативна традиційним нативним застосункам без втрати основного оффлаїн функціоналу	17
2.3 Pinterest - PWA, як кросс-платформене рішення.....	18
2.4 Висновки до розділу.....	19
3 Інструменти та алгоритми	23
3.1 Середовище розробки WebStorm	23
3.2 Мови Java Script та Type Script.....	24
3.3 Vite	26
3.4 Pnpm.....	27
3.5 Docker	27
3.6 Frontend технології	28
3.6.1 HTML, алгоритм відмальовування в браузері	28
3.6.2 CSS та Less препроцесор.....	30
3.6.3 React	31
3.6.4 Zustand	32
3.6.5 Workbox	34
3.7 Backend технології.....	34
3.7.1 Node.js.....	34
3.7.2 Express.....	35
3.8 PostgreSQL.....	37

	9
3.9 Git	38
3.10 Сучасні браузерери та їх API	38
3.11 PWA технології та TWA технології.....	41
3.12 MVC	44
3.13 REST API	44
3.14 MVVM	45
3.15 Висновки до розділу.....	45
4 Опис програмної реалізації.....	47
4.1 PWA застосунок.....	47
4.1.1 Архітектура	48
4.1.2 Реалізація PWA	52
4.1.3 Система оффлаїн обробки даних	54
4.1.4 Система синхронізації даних.....	57
4.2 Backend сервіс	61
4.3 Docker	61
5 Порівняння рwa з традиційними рішеннями	63
5.1 Проблема нативного підходу	63
5.3 PWA, як кросс-платформене рішення	65
5.4 Проблеми PWA	66
5.5 Коли використовувати PWA	68
5.5 Висновки до розділу.....	70
6 Робота користувача з розробленим застосунком	73
6.1 Вимоги до користувача.....	73
6.1.1 Технічні вимоги до пристрою	73
6.1.2 Вимоги до дозволів браузера та системи	74
6.1.3 Вимоги до користувацьких навичок та взаємодії.....	74
6.2 Інструкція для користувача	74
6.2.1 Сценарії роботи користувача	75
6.2.2 Інтерфейс користувача.....	78

	10
6.3 Інструкція для завантаження PWA	84
Висновки.....	87
Перелік джерел посилання	89
Додаток А Текст програми	92
Додаток Б Слайди презентації.....	105

ПЕРЕЛІК СКОРОЧЕНЬ ТА УМОВНИХ ПОЗНАК

- API – програмний інтерфейс застосунку;
- VM – віртуальна машина;
- PWA – прогресивний веб застосунок;
- UI – користувацький інтерфейс.

ВСТУП

Користувачі сайтів часто стикаються з нестабільним інтернет-з'єднанням або обмеженим доступом до мережі, а традиційні вебзастосунки та мобільні застосунки виявляють низку недоліків. Звичайні вебсайти вимагають постійного підключення до мережі, що обмежує їхню доступність у реальних умовах експлуатації, а нативні мобільні програми потребують окремої розробки для кожної платформи, мають великі обсяги пам'яті, складну процедуру оновлення та розгортання, а також високі витрати на підтримку.

Саме тому особливого значення набувають прогресивні вебзастосунки (Progressive Web Applications, PWA) – нове покоління вебтехнологій, які поєднують найкращі властивості сайтів та нативних застосунків. Вони здатні працювати оффлайн, використовувати локальні сховища, кешувати дані, надсилати push-сповіщення, синхронізуватися із сервером після відновлення з'єднання, а також забезпечувати нативну взаємодію з користувачем через сучасні API браузерів. Усе це робить PWA однією з найперспективніших технологій у галузі розробки кросплатформених застосунків, які можуть функціонувати на будь-якому пристрої від смартфонів і планшетів до настільних систем, незалежно від операційного середовища.

Розвиток PWA-технологій відкриває нові можливості для створення високопродуктивних та стійких до мережевих збоїв систем. У контексті сучасних тенденцій веброзробки зростає попит на рішення, які забезпечують безперервність користувацького досвіду навіть без доступу до мережі. Це особливо важливо для корпоративних, логістичних, торгових, аналітичних і мобільних систем, де втрати зв'язку можуть призводити до збоїв у роботі або втрати даних. Відтак, дослідження методів побудови оффлайн-пріоритетних прогресивних вебзастосунків є актуальним як у науковому, так і у практичному аспектах.

Метою дослідження є підвищення доступності кросплатформених програмних рішень та скорочення витрат на їх розробку і підтримку шляхом

використання офлайн-пріоритетного прогресивного вебзастосунку з єдиною кодовою базою та відкладеною синхронізацією даних. Створення вебзастосунку, який забезпечує повноцінну роботу без постійного інтернет-з'єднання, дозволяє зберігати та обробляти дані локально, а після відновлення зв'язку виконує гарантовану синхронізацію змін із сервером. Для досягнення цієї мети передбачено комплексне вивчення сучасних технологій веброзробки, механізмів кешування, алгоритмів синхронізації даних та архітектурних підходів до побудови автономних систем.

У межах дослідження аналізуються існуючі рішення на базі PWA, визначаються їх переваги над традиційними мобільними та десктопними застосунками, а також проводиться порівняння ефективності використання оффлайн-механізмів у різних сценаріях. Значна увага приділяється архітектурі застосунку, яка поєднує фронтенд і бекенд частини через REST API, забезпечуючи узгоджену взаємодію між клієнтом і сервером навіть у разі тимчасової відсутності мережі.

Теоретичну основу роботи становлять концепції оффлайн-доступності у вебсередовищі, моделі синхронізації даних після втрати з'єднання, технології кешування ресурсів за допомогою Service Workers, IndexedDB, Cache API та сучасні підходи до створення реактивних інтерфейсів із використанням бібліотек React і фреймворку Node.js для серверної частини. .

Практична частина дослідження полягає у створенні повноцінного оффлайн-пріоритетного вебзастосунку, який може використовуватися у сферах, де мережеві збої або обмежена пропускна здатність є критичними — наприклад, у системах збору польових даних, мобільних бізнес-застосунках, корпоративних платформах чи навчальних сервісах. Розроблене рішення дозволяє мінімізувати втрати інформації, знизити навантаження на сервер і підвищити загальну стабільність роботи системи.

Таким чином, ця робота спрямована на комплексне дослідження архітектурних, технічних і програмних рішень, які дозволяють реалізувати повноцінний автономний вебзастосунок із гарантованою синхронізацією даних. Отримані результати мають практичну цінність для розробників

сучасних систем, які потребують стійкості до мережових перебоїв, а також підтверджують перспективність технології Progressive Web Application як фундаментальної платформи для майбутнього розвитку вебзастосунків.

1 ПОСТАНОВКА ЗАВДАННЯ ДОСЛІДЖЕННЯ

У сучасних умовах цифровізації бізнесу та постійного зростання вимог до швидкості й доступності інформаційних систем усе більшої актуальності набувають технології, що забезпечують безперебійну роботу застосунків навіть за відсутності стабільного інтернет-з'єднання. Традиційні підходи - нативні мобільні застосунки або звичайні вебсайти - мають свої недоліки: перші потребують окремої розробки під кожную платформу, другі залежать від постійного доступу до мережі.

Одним із найперспективніших напрямів вирішення цих обмежень є використання технології Progressive Web Application (PWA) – гібридного підходу, що поєднує переваги веб та нативних рішень. Такі застосунки мають змогу працювати оффлайн, кешувати дані, синхронізувати зміни після відновлення з'єднання, підтримувати push-сповіщення та інші сучасні браузерні API, забезпечуючи при цьому високу продуктивність і кросплатформеність.

Мета дослідження – підвищення доступності кросплатформених програмних рішень та скорочення витрат на їх розробку і підтримку шляхом використання офлайн-пріоритетного прогресивного вебзастосунку з єдиною кодовою базою та відкладеною синхронізацією даних. Розробка вебзастосунку з автономною роботою, з підвищеною стійкістю до втрати мережевого з'єднання та підвищеним рівнем відновлення даних після появи доступу до мережі шляхом застосування оптимізованих механізмів локального збереження та синхронізації.

Для досягнення цієї мети необхідно виконати такі завдання:

- проаналізувати існуючі рішення та визначити переваги й обмеження підходу PWA у порівнянні з нативними застосунками;
- дослідити архітектурні підходи до побудови автономних вебзастосунків із підтримкою кешування, локального сховища та синхронізації даних;
- розробити архітектуру і програмну реалізацію PWA-застосунку з

можливістю автономної роботи, використовуючи сучасні вебтехнології;

- реалізувати механізми локального зберігання даних, визначення стану з'єднання, черги відкладених запитів і синхронізації змін після появи мережі;

- продемонструвати придатність PWA для застосування в корпоративних, бізнес-орієнтованих і соціальних сценаріях, де потрібна надійність і стійкість до мережевих перебоїв.

Очікуваний результат дослідження створення універсального автономного вебзастосунку, що поєднує швидкодію, гнучкість оновлення та нативний користувацький досвід, доведення ефективності PWA як сучасної альтернативи традиційним мобільним рішенням і демонстрація потенціалу цієї технології для розвитку бізнес-застосунків.

2 ОГЛЯД ІСНУЮЧИХ РІШЕНЬ

У цьому розділі розглядаються декілька популярних існуючих PWA, переваги використання саме таких підходів для вирішення їх бізнес задач. Та результати запровадження цих рішень.

2.1 Twitter Lite - переваги PWA рішення в порівнянні з традиційними вебзастосунками

Twitter Lite – легка PWA-версія Twitter, впроваджена навесні 2017 року. Метою був швидкий та надійний доступ до Twitter у мобільних мережах з низькою пропускнуою здатністю, а також покращення користувацького досвіду для користувачів без можливості встановити повну нативну версію застосунку. Twitter Lite використовує сервіс-воркер для кешування контенту та стратегію «завантажуй-на-вимогу» (lazy-loading) даних, тому перше завантаження займає менше 3 секунд навіть на 3G, а після першого завантаження всі наступні входи на сайт використовують кеш без завантаження, який за потреби оновлюється фоново у випадку виходу нової версії сайту. Також використання PWA дозволило додати на головний екран пристроїв користувачів іконку з прямим входом сайт. Як наслідок, статистика взаємодії користувачів з сайтом Twitter [1] значно зросла - з PWA на 65% більше сторінок переглядають за сесію і на 75% більше твітів відправляють, а показник відмов упав на 20%. Крім того, Twitter Lite підтримує web-push сповіщення та показує підказку «Додати на головний екран», що дало ~250 000 щоденних відкриттів із головного екрану й понад 10 мільйонів відправлених сповіщень на день. А також, такий підхід забезпечив величезну економію трафіку і місця: Twitter Lite завантажує ~600 КБ, тоді як нативний Android-застосунок важить десятки мегабайт. У підсумку головними перевагами PWA-підходу для Twitter стали [1]:

- швидкість і легкість – PWA-версія завантажується майже миттєво і займає незначну пам'ять (менше 3% від розміру нативного застосунка);

- залучення користувачів – PWA дає можливість відправляти push-повідомлення і «встановлювати» сайт на головний екран, в результаті багато користувачів натомість, навіть без інсталяції з магазину;

- охоплення нових ринків – через менші технічні вимоги прогресивний застосунок працює на слабких телефонах і повільному зв'язку, розширюючи аудиторію у країнах з низьким рівнем мобільного зв'язку.

Таким чином впровадження PWA-технологій дало Twitter суттєві переваги з точки зору продуктивності та залучення користувачів за рахунок покращення основних метрик сайту та кешування статичних даних. Також користувачі частіше згадують про PWA ніж про звичайну версію сайту оскільки застосунок має нативний доступ через іконку з пристрою.

2.2 Google Maps Web, як альтернативна традиційним нативним застосункам без втрати основного оффлайн функціоналу

Google Maps Web використовує технологію PWA, як спосіб принести карти людям із слабкими телефонами і повільним інтернетом. У вебзастосунку зберігаються основні функції: визначення місцезнаходження, побудова маршруту (авто, пішки), оновлення трафіку, інформація про компанії (години роботи, адреси) тощо. При цьому при встановленні на пристрій PWA версія займає на в рази менше місця (його розмір у 100+ разів менший за звичайний нативний застосунок). Наприклад, користувач із повільною мережею або старим Android-пристроєм у країнах на зразок Індії тепер може швидко отримувати маршрути на картах без довгого завантаження. Нижче наведено ключові моменти щодо Google Maps PWA:

- швидкість і легкість – в порівнянні з нативним застосунком PWA-версія стартує швидше і рідше кешує важкі об'єкти, що добре для слабких пристроїв;

- широка доступність – завдяки PWA картами можуть користуватися пристрої з обмеженим сховищем та процесорними можливостями, які не мають змоги запустити звичайну версію;

- оффлайн доступність до обмеженого функціоналу – навідміну від звичайного сайту PWA версія дає змогу закешувати данні та переглядати карти оффлайн з кешу, а також користуватись іншими можливостями, які не потребують додаткових запитів на сервер, а також дає можливість входу напрямку з пристрою через іконку застосунку, а не через браузер.

У випадку з Google Maps, цей підхід дозволив охопити ширшу аудиторію, особливо в країнах з обмеженим доступом до інтернету, таких як Індія [2]. Користувачі можуть отримувати маршрути, інформацію про трафік та бізнеси навіть оффлайн, тоді як традиційний сайт вимагає постійного підключення та повільного перезавантаження сторінок, а нативний застосунок займає набагато більше місця та потребує більш дорогих мобільних пристроїв.

2.3 Pinterest - PWA, як кросс-платформене рішення

Pinterest спочатку мав звичайний мобільний веб-сайт з низькими коефіцієнтами конверсії. Команда побачила, що багато мобільних відвідувачів не встановлювали нативний застосунок, а сайт був повільний та важкий, особливо на мобільних пристроях. Тому було прийнято рішення повністю переробити веб у PWA, додати service worker для оффлайн-кешування, застосувати lazy-loading для зображень та даних і забезпечити поведінку, схожу на нативний застосунок (іконка на пристрої користувача, швидкі переходи, push-повідомлення там, де це можливо). Цей підхід дозволив доставити нативний досвід використання через браузер, зберігши єдину кодову базу для веб-інтерфейсу.

Пізніше Pinterest опублікував метрики [3], які показали значний бізнес-ефект - збільшення зацікавленості (приблизно на 60%), збільшення часу, проведеного на платформі (~40%), зростання кількості збережених пінів (понад дворазове в деяких вимірюваннях), а також різке зростання мобільної активності в країнах як Індія і Бразилія. Ці цифри відображають, як

технічна оптимізація та перехід на PWA прямо вплинули на ключові бізнес-метрики.

Перевагами PWA-підходу стали:

- швидкість і низьковартість – таке рішення було значно швидшим та дешевшим ніж розробка окремо застосунку для Android, окремо для Ios і окремий перепис самої мобільний версії сайту, для такого рішення було достатньо тільки однієї команди веб розробників і велика частина коду базового коду мала змогу перевикористуватись;
- залучення користувачів – інтерфейс подібний на застосунок та можливості PWA виявились більш зручними та інтуїтивними для користувачів, що підвищило їх метрики;
- охоплення нових ринків - через менші технічні вимоги прогресивний застосунок працює на слабких телефонах і не потребує великих ресурсів, як традиційні нативні застосунки.

PWA-підхід у Pinterest дозволив швидко та дешево перейти на нативний інтерфейс з підтримкою звичайних функцій до яких звикли користувачі подібних застосунків без необхідності розробки під окремі платформи і зберігаючи базовий код та переваги сайту.

2.4 Висновки до розділу

Розглянувши існуючі рішення на базі Progressive Web Apps, такі як Twitter Lite, Google Maps Web та Pinterest, можна зробити висновки щодо переваг цього підходу в порівнянні з традиційними вебзастосунками та нативними застосунками. Ці приклади демонструють, як PWA-технології ефективно вирішують ключові бізнес-задачі, пов'язані з продуктивністю, залученням користувачів, економією ресурсів та розширенням аудиторії. Кожен з проаналізованих випадків ілюструє різні сценарії використання PWA, від оптимізації для мереж з низькою пропускнуою здатністю до створення кросс-платформених рішень з оффлайн-функціоналом.

По-перше, PWA вирізняються значною перевагою в швидкості та

ефективності ресурсів, що є критичним для користувачів з обмеженим доступом до інтернету або слабкими пристроями. Загалом, у всіх трьох сценаріях PWA дозволяють уникнути проблем традиційних сайтів, таких як повільне перезавантаження сторінок, і нативних застосунків, які вимагають значного місця на пристрої та потужного апаратного забезпечення. Результати показують, що така оптимізація безпосередньо впливає на бізнес-метрики.

По-друге, PWA суттєво підвищують залучення користувачів завдяки функціям, що імітують нативні застосунки, таким як додавання іконки на головний екран, web-push сповіщення та швидкі переходи. Це перетворює звичайний веб-сайт на "встановлений" застосунок без потреби в завантаженні з магазину, що особливо корисно для користувачів, зі слабкими пристроями. У Google Maps Web ця функція поєднується з оффлайн-доступом, дозволяючи користувачам переглядати кешовані карти та маршрути без підключення, що робить досвід більш надійним і зручним, ніж на традиційному сайті, де кожна дія вимагає запиту до сервера. У сценаріях, де бізнес залежить від частоти взаємодії (соціальні мережі, карти, контент-платформи), PWA забезпечують вищий рівень утримання, ніж традиційні сайти, які не мають push-повідомлень чи прямого доступу з екрану, і нативні застосунки, які вимагають додаткових кроків для встановлення. Результати впровадження підкреслюють, що таке залучення не тільки покращує користувацький досвід, але й прямо впливає на монетизацію: наприклад, понад 10 мільйонів щоденних сповіщень у Twitter сприяють більшій активності та рекламі.

По-третє, PWA є потужним інструментом для розширення охоплення ринків та кросс-платформенної доступності, особливо в умовах різноманітності пристроїв і мереж. У сценаріях глобального бізнесу, де потрібно охопити різноманітні платформи без значних інвестицій, PWA виявляються дешевшим і швидшим рішенням. На відміну від нативних застосунків, які прив'язані до конкретних ОС і вимагають схвалення в магазинах, PWA працюють через браузер, але з нативними можливостями,

що робить їх універсальними. Результати показують економічний ефект: зменшення розміру застосунка в рази, зростання аудиторії і уникнення бар'єрів, як повільний інтернет.

Узагальнюючи розглянуті особливості та для наочності відмінності між підходами нижче подано порівняльну таблицю (табл. 2.1). У ній систематизовано ключові критерії, які впливають на вибір технологічного рішення: підтримка оффлайн-режиму, швидкість роботи, вимоги до пристрою, складність розробки, можливість доступу до системних функцій та інші.

Таблиця 2.1 – Переваги впровадження PWA рішення

	PWA	Звичайний вебзастосунок	Нативний застосунок
Єдина кодова база	Одна кодова база для всіх платформ	Одна кодова база, але без можливості “встановлення” чи оффлайн-роботи	Потрібно 2 окремі реалізації (Android + iOS), або гібридні фреймворки
Розмір застосунку	Дуже малий (сотні КБ – кілька МБ)	Немає “розміру застосунку”, лише завантаження сторінок	Десятки – сотні МБ
Швидкість роботи	Висока: кешування, локальні дані, миттєві переходи	Середня: кожна дія потребує запиту до сервера	Найвища: прямий доступ до системних ресурсів
Швидкість першого завантаження	Висока завдяки попередньому кешу	Залежить від мережі (повне завантаження з сервера)	Повільна інсталяція + запуск
Оновлення	Миттєві, непомітні через service worker	Миттєві після оновлення серверу	Потрібне оновлення через магазини, перевірка модераторів
Доступ до системних функцій пристрою	Частковий	Дуже обмежений	Повний доступ

Загалом результати впровадження демонструють: зростання ключових метрик (активність, час на платформі, конверсії), економія ресурсів (трафік, пам'ять, витрати на розробку) та розширення аудиторії. Для бізнесу, що розглядає PWA, рекомендується аналізувати цільову аудиторію, бізнес-задачі та потенційні ризики (обмежена підтримка в деяких браузерах). У підсумку, PWA представляють еволюційний крок у веб-розробці, що дозволяє компаніям досягати нативного рівня досвіду з веб-гнучкістю і можуть стати стандартом для майбутніх мобільних рішень.

3 ІНСТРУМЕНТИ ТА АЛГОРИТМИ

Програмний застосунок, написаний на мовах JavaScript та TypeScript в операційній системі Windows. В якості середовища розробки було обрано WebStorm від JetBrains. Для тестування використовувався Google Chrome браузер на ноутбучі Lenovo Legion 5 та мобільна версія на One Plus Nord 2 5G, для контейнеризації я використав Docker, тестові дані зберігались в PostgreSQL базі даних. Для збірки TypeScript в JavaScript використовувався Vite збиральник. Для завантаження сторонніх бібліотек використовувався Pnpm. Для контролю версій використовувався Git. UI реалізований на базі системи бібліотек React. Для внутрішнього стеїт менеджменту та кешування застосовувався Zustand. Також для роботи зі стилями використовується Less препроцесор. Для бекенд частини використовувався Node.js та фреймворк Express.

Для реалізації PWA функціоналу кешування використовувалась бібліотека workbox.

В реалізованій мною системі я використав такі алгоритми: для клієнт-серверної взаємодії я використав шаблон MVC з Rest API архітектурою; з боку PWA я використав MVVM шаблон для відокремлення UI, від бізнес логіки модель.

3.1 Середовище розробки WebStorm

WebStorm є потужним та спеціалізованим інтегрованим середовищем розробки (IDE), створеним компанією JetBrains, яке орієнтоване переважно на розробку вебзастосунків з використанням JavaScript, TypeScript, HTML, CSS та пов'язаних технологій. На відміну від універсальних редакторів, таких як Microsoft Visual Studio, WebStorm пропонує глибоку інтеграцію саме з екосистемою сучасного фронтенду, що покращує досвід з конкретними, популярними технологіями у сфері вебу. Наприклад для розробників, які працюють з фреймворками React, Angular, Vue.js, Node.js та іншими.

WebStorm має дуже зручний механізмів автодоповнення (IntelliSense-подібним, але адаптованим під JavaScript-екосистему), який враховує контекст проєкту, типи даних, імпорти та навіть конфігурації фреймворків. Завдяки вбудованому аналізатору коду, IDE миттєво виявляє помилки, пропонує рефакторинг та автоматично імпортує модулі. Це значно прискорює розробку та зменшує кількість ручних помилок.

За даними JetBrains Developer Ecosystem 2023 [4] року, близько 25% фронтенд-розробників обирають WebStorm як основне IDE, що робить його одним з лідерів у сегменті JavaScript/TypeScript (після Visual Studio Code з часткою понад 70%).

3.2 Мови Java Script та Type Script

Java Script, був створений, як легка скриптова мова для додавання інтерактивності вебсторінкам. Ця скриптова мова була створена для додавання динамічної інтерактивності у браузері Netscape Navigator але з часом Java Script еволюціонував від простого інструменту для клієнтської логіки до повноцінного стандарту веброзробки, який підтримується всіма сучасними браузерами. А також завдяки спеціалізованим платформам, як Node.js, Java Script вийшов за межі браузера і став універсальною мовою для серверної розробки, десктопних застосунків, мобільних застосунків, ігрових рушіїв та вбудованих систем [5].

JavaScript є інтерпретовною, динамічно типізованою, об'єктно-орієнтованою (з підтримкою прототипного наслідування), функціональною та подієво-орієнтованою мовою програмування. Він підтримує функції першого порядку (функції як об'єкти першого класу), що дозволяє передавати їх як аргументи, повертати з інших функцій і присвоювати змінним. Однією з ключових особливостей є асинхронне програмування за допомогою колбеків, промісів та сучасного синтаксису `async/await`, що робить мову ідеальною для роботи з мережевими запитами, таймерами, анімаціями та іншими операціями, які не блокують основний потік

виконання. Java Script має автоматичне керування пам'яттю через збірник сміття, що спрощує розробку.

Мова підтримує прототипне наслідування, на відміну від класичного об'єктно-орієнтованого програмування тут кожен об'єкт має прототип, від якого успадковує властивості та методи. З появою ES6 (ECMAScript 2015) з'явився синтаксичний цукор у вигляді ключового слова `class`, але під капотом це все ще прототипи. Java Script є однопоточним за своєю природою, але завдяки `event loop` ефективно обробляє асинхронні операції, використовуючи чергу `macrotask queue` та `microtask queue`. Це дозволяє виконувати неблокуючі операції, наприклад, читання файлів чи HTTP-запити, без створення додаткових потоків [5].

Type Script - це надмножина Java Script, тобто будь-який валідний JS-код є валідним і в Type Script, але додає можливість статичної типізації, а також інтерфейси, типи, генерації, повноцінні класи з модифікаторами доступу `public`, `private`, `protected` і `readonly`, а також декоратори для мета-програмування. Type Script компілюється в чистий Java Script за допомогою компілятора `tsc`, тому працює скрізь, де працює JS та дозволяє виявляти помилки на етапі компіляції, а не виконання, а також забезпечує потужне автодоповнення в IDE. Він ідеально інтегрується з React, Angular, Vue, NestJS і наразі вже є стандартом для розробки на JS.

Type Script компілюється в чистий Java Script за допомогою команди `tsc`, і весь типовий код, анотації, інтерфейси, модифікатори доступу повністю зникають у вихідному JS, тому продуктивність виконання ідентична звичайному Java Script, а розмір коду може навіть зменшитися за рахунок видалення типів. Налаштування компіляції задаються у файлі `tsconfig.json`, де можна обрати цільову версію Java Script, від ES3 до ESNext, увімкнути строгу перевірку типів, дозволити чи заборонити неявний `any`, увімкнути підтримку декораторів, модулів, JSX тощо [6].

У результаті, поява TypeScript змінила культуру розробки у JavaScript-екосистемі до більш формальних, контрольованих і масштабованих підходів. Висока щільність інструментальної підтримки дає змогу будувати великі

корпоративні проєкти з десятками чи сотнями розробників без втрати контрольованості над кодом. TypeScript підвищує передбачуваність та стабільність системи й дозволяє виявляти помилки ще на етапі розробки та компіляції, значно знижуючи вартість помилок у продакшні [6].

3.3 Vite

Vite - революційний інструмент збірки для сучасної веброботи. Vite використовує пре-бандлінг залежностей за допомогою надшвидкого бандлера esbuild, написаного на Go, який перетворює CommonJS-модулі в ESM у 10 разів швидше, ніж JavaScript-базовані альтернативи, і кешує їх для подальших запусків, уникаючи повторної обробки стабільних бібліотек на кшталт lodash чи react.

У режимі розробки Vite обробляє тільки зміни в коді користувача, використовуючи Hot Module Replacement (HMR), яке оновлює лише змінений модуль у браузері, зберігаючи стан застосунка - наприклад, якщо ви змінили стиль компонента в React, весь застосунок не перезавантажиться, а тільки цей компонент оновиться, що радикально прискорює ітерації та налагодження, роблячи процес розробки набагато приємнішим, особливо в великих проєктах.

Для продакшн-зборки Vite переходить на Rollup - зрілий і гнучкий бандлер, який оптимізує код для tree-shaking, code-splitting, бібліотечного режиму та сервер-сайд рендерингу, з налаштуваннями за замовчанням, які можливо кастомізувати через vite.config.js, наприклад, додати плагіни для PWA.

Синергія esbuild у дев-режимі та Rollup у продакшн-режимі робить Vite гібридною системою, яка поєднує буквально краще з обох світів - максимальна швидкість під час розробки й максимальні оптимізації під час релізної збірки [7].

3.4 Pnpm

Performant Node Package Manager, з'явився у 2016 році як інноваційний менеджер пакетів для Node.js з метою вирішення ключових проблем npm, таких як повільне встановлення залежностей, надмірне споживання дискового простору та потенційних проблем безпеки через неконтрольований доступ до пакетів. PNPM використовує унікальну архітектуру глобального сховища файлів, де кожен файл пакета зберігається лише раз на диску з унікальним хешем, а в локальному `node_modules` створюється структура символічних лінків (symlinks) або жорстких лінків (hard links) на ці файли, що дозволяє уникнути дублювання: якщо у вас 100 проєктів, що залежать від lodash, npm запише 100 повних копій lodash, займаючи гігабайти, тоді як PNPM запише lodash лише раз у глобальне сховище, а в кожному проєкті створить лише посилання, зменшуючи використання диска до 90-95% [8].

Ці плюси PNPM роблять його гарним рішенням для локальних машин з обмеженим SSD чи CI/CD з ефемерними, контейнерними образами, як Docker, де npm міг би заповнити більше простору та довше працювати, що напряду вплинуло б на CI/CD [8].

3.5 Docker

Docker, став революційним інструментом у світі DevOps і хмарних технологій, змінивши спосіб, як розробники створюють, розгортають і керують застосунками, завдяки концепції контейнеризації - легкої, портативної, ізольованої та ефективної альтернативи віртуальним машинам. На відміну від класичних VM, які віртуалізують апаратне забезпечення (CPU, RAM, диск) і запускають повну операційну систему для кожного застосунку, Docker використовує можливості ядра Linux, щоб ізолювати процеси на рівні операційної системи, дозволяючи запускати кілька застосунків у контейнерах, які ділять одне ядро хоста, але мають власні файлові системи,

мережеві інтерфейси, процеси та обмеження ресурсів - тому контейнери легші за ВМ [9].

Основою Docker є Docker Engine - клієнт-серверна програма, що складається з демона `dockerd`, який керує образами, контейнерами, мережами та томами, і CLI `docker`, через який розробники віддають команди. Образ - це незмінний шаблон, створений на основі `Dockerfile`, текстового файлу з інструкціями, що описує, як зібрати застосунок.

Docker усуває проблеми розгортання на різних пристроях та системах, ізоляція запобігає конфліктам залежностей, масштабованість дозволяє запускати тисячі контейнерів на кластері, ефективність зменшує витрати в хмарах [9].

3.6 Frontend технології

3.6.1 HTML, алгоритм відмальовування в браузері

HTML - це мова розмітки, що визначає структуру, семантику та взаємодію сучасних вебсторінок, дозволяючи браузерам інтерпретувати текст, зображення, посилання, форми, мультимедіа та складні компоненти. HTML не є мовою програмування - це мова розмітки, тобто набір тегів (наприклад, `<p>`, `<div>`, `<img>`, `<a>`), які описують, що саме є на сторінці: заголовки, абзаци, списки, таблиці, кнопки, поля вводу, і семантичну роль кожного елемента, що критично важливо не лише для відображення, але й для доступності (screen readers для людей з вадами зору).

Кожен HTML-документ починається з `<!DOCTYPE html>`, що вказує браузеру використовувати сучасний стандарт (HTML5+), далі йде кореневий елемент `<html>`, всередині якого `<head>` містить метадані (заголовок `<title>`, підключення CSS через `<link>`, JavaScript через `<script>`, мета-теги для `viewport`, `charset=utf-8`, опису сторінки), а у `<body>` видимий вміст: текст, зображення, відео, canvas, SVG, інтерактивні елементи. HTML5, представлений у 2014 році, радикально розширив можливості: додав нативну підтримку `<video>`, `<audio>`, `<canvas>` для графіки, `<svg>` для векторів, Web

Storage (localStorage, sessionStorage), Web Workers, Geolocation API, Drag-and-Drop, семантичні теги (<section>, <article>, <aside>, <figure>, <main>), атрибути для форм (required, pattern, autocomplete), мікродані (itemscope, itemtype) та ARIA-ролі для доступності.

Коли браузер отримує HTML-документ, починається складний, але чітко визначений процес рендерингу, який складається з кількох етапів, що виконуються послідовно і асинхронно для максимальної продуктивності. Спочатку відбувається парсинг HTML: браузер читає байти, декодує їх у символи, а потім розбиває на теги, атрибути, текст. Під час цього будуються дві ключові структури: DOM (Document Object Model) - деревоподібне представлення документа, де кожен тег стає вузлом і CSSOM (CSS Object Model) - аналогічна структура, але для стилів, яка будується паралельно, коли браузер стикається з <link rel="stylesheet"> або <style>. Парсер HTML толерантний, тому навіть при помилках він намагається виправити і продовжити, але це може призводити до неочікуваної поведінки.

Після побудови DOM і CSSOM настає етап Render Tree: браузер об'єднує їх, відкидаючи невидимі елементи (display: none, <head>, <script>), і створює дерево рендерингу, де кожен вузол знає свої стилі (обчислені з CSS-правилами, спадкування, каскаду, специфічності). Далі йде Layout (або Reflow) - обчислення геометрії: для кожного елемента в Render Tree визначаються розміри (width, height), позиція (x, y) на екрані з урахуванням типу відображення (block, inline, flex, grid), float, position (static, relative, absolute, fixed, sticky). Це ресурсоємкий процес: зміна ширини одного елемента може викликати перерахунок усіх нащадків і сусідів [10].

Потім відбувається Painting, браузер перетворює геометрію і стилі в пікселі на екрані, малюючи фони, межі, текст, зображення, тіні, градієнти. Сучасні браузери оптимізують це - сторінка розбивається на шари, які малюються окремо GPU, і при анімаціях (наприклад, transform, opacity) оновлюється лише шар, а не весь екран - це основа 60 fps анімацій.

Після етапу відмалювання відбувається ще один важливий внутрішній процес - оновлення композиції: браузер зводить усі окремі шари у кінцевий

кадр, що відображається на моніторі. Це теж оптимізується, якщо змінюється лише прозорість або перетворення (наприклад, рух елемента на сторінці за допомогою transform або зміна opacity), браузер може оновити тільки певний шар без повного повторного обчислення розташування і повторного розфарбовування всього дерева рендерингу. Таким чином, сучасний механізм рендерингу складений з алгоритма незалежних етапів [10].

3.6.2 CSS та Less препроцесор

Cascading Style Sheets - це мова стилізації, створена для чіткого відокремлення презентаційного шару від структурного HTML. Вона дозволяє визначати кольори, шрифти, відступи, позиціонування елементів, анімовану поведінку, адаптивність під різні розміри екранів та інші візуальні ефекти без зміни самої розмітки. Стили застосовуються з урахуванням походження (inline, embedded, external), специфічності селекторів, порядку оголошення і спадкування. Отже, результуючий вигляд елемента - це підсумок усіх цих правил, що працюють одночасно. CSS складається з селекторів (наприклад .class, #id, div > p, атрибутні селектори, псевдокласи :hover чи :active та псевдоелементи ::before чи ::after), властивостей (color, margin, display, transition, grid-template-columns) та конкретних значень (наприклад red, 16px, flex, 1fr 2fr, ease-in-out).

Less - це препроцесор, який розширює можливості CSS і додає в нього елементи програмування. Він підтримує змінні, вкладені правила, міксини (фактично багаторазові фрагменти стилів), функції, арифметичні операції над стилями, умовні конструкції, цикли та імпорти. Це робить написання стилів значно ефективнішим і зручнішим, особливо в середніх та великих проєктах. Однак в кінцевий збірник Less не входить, бо перед тим, як потрапити у браузер, код препроцесора компілюється у звичайний CSS, який інтерпретують усі браузери.

Less Module – це спосіб забезпечити модульність у стилях, усуваючи типовий недолік звичайного CSS, де всі імена класів живуть у глобальному

просторі і можуть випадково конфліктувати. Модульна модель автоматично генерує унікальні локальні назви класів при компіляції, наприклад `.button` перетворюється на `.button__a3f82`, що виключає зіткнення стилів між компонентами з однаковими іменами класів. Крім того, `Less Module` експортує мапу локальних назв у JavaScript, що дозволяє імпортувати стилі як об'єкт – `import styles from './Button.module.less'`; і використовувати їх, звертаючись до конкретної властивості, наприклад `div className={styles.button}`. Це робить стилізацію компонентів локальною і передбачуваною, що важливо у сучасних компонентних архітектурах на `React`, де кожен компонент має свій ізольований набір стилів [11].

3.6.3 React

`React` – це JavaScript бібліотека для створення інтерфейсів користувача, яка зосереджується на побудові динамічних, інтерактивних вебзастосунків через компонентний підхід. Вона не є повноцінним фреймворком на кшталт `Angular`, а радше фундаментальною основою для UI, яка може бути доповнена іншими інструментами: маршрутизаторами, менеджерами стану, серверним рендерингом з використанням `Next.js`. `React` дозволяє розробнику описати, як має виглядати інтерфейс у певному стані декларативно, без необхідності безпосередньо змінювати DOM вручну. Такий декларативний стиль робить код структурованим і легшим у підтримці.

Компоненти `React` отримують дані через `props`, а свої внутрішні значення зберігають у стані, який керується через хуки. Найбільш базовий механізм – `useState`: `const [count, setCount] = useState(0)`, де `count` – значення, а `setCount` – функція його зміни, яка тригерить повторний рендер. Усередині `React` працює `Virtual DOM` – легка копія реального DOM у пам'яті. Після зміни стану `React` генерує нову версію віртуального дерева, порівнює її з попередньою за допомогою алгоритму `Fiber` і вносить лише мінімальні зміни у реальний DOM. Такий алгоритм підвищує продуктивність, особливо коли

застосунок великий і має багато компонентів, а ключі у списках допомагають React ефективно відслідковувати, які елементи змінились, а які - ні.

React часто порівнюють з Vue.js та Angular. Усі вони будують SPA через компоненти, але підходи різні. Vue.js простіший у вході, використовує Proху для реактивності без Virtual DOM, розділяє структуру у файлах .vue і має менший розмір бібліотеки, що робить його зручним для MVP та середніх продуктів. React краще підходить для більш складних інтерфейсів з гнучкою архітектурою та великою кількістю взаємодій. Angular, натомість, це повний фреймворк з вбудованою маршрутизацією, DI, формами і вимогою використовувати TypeScript, що часто робить його вибором для надвеликих корпоративних систем, але за рахунок складності та меншої швидкості розробки.

Для клієнтської маршрутизації у React-застосунках найчастіше використовується бібліотека React Router. Вона дозволяє змінювати URL та оновлювати інтерфейс без перезавантаження сторінки, інтегруючись з браузерною історією. React Router надає декларативний API для опису маршрутів через JSX або конфігураційні масиви. Компонент RouterProvider рендериться у корені, а бібліотека визначає, який компонент показати для поточного шляху. Маршрути можуть бути статичними чи динамічними (наприклад, `"/users/:id"`), а також вкладеними, що дозволяє створювати ієрархії сторінок із загальними лейаутами. При пошуку маршруту React Router використовує ранжування: статичні шляхи мають вищий пріоритет над динамічними, а вкладені - логічно вибудовують структуру застосунку і роблять маршрутизацію більш природною та передбачуваною для розробника [12].

3.6.4 Zustand

Zustand був створений для спрощення управління глобальним станом у застосунках на React. На відміну від більш важких та формалізованих інструментів як Redux, він робить акцент на мінімалізмі та прямому доступі

до стану. Розробник створює store за допомогою функції create, яка повертає спеціальний гачок для доступу до стану та до функцій, що цей стан змінюють. Завдяки цьому немає потреби створювати провайдери та обгортати дерево компонентів додатковими шарами, бо магазин можна підключити до будь-якого компонента без зайвої логіки [13].

Zustand також має механізм селекторів, що дозволяє вибирати лише конкретні частини стану і не викликати повторні рендери компонентів, коли ці частини не змінилися. Це дає суттєву перевагу у продуктивності під час роботи з великими інтерфейсами. Сам стан можна оновлювати як простими об'єктами, так і за допомогою функційного оновлення, що робить процес змін прозорим та передбачуваним.

Бібліотека підтримує асинхронні операції на рівні store без додаткових надбудов. Розробник може в самому магазині виконувати очікування від мережевого запиту та оновлювати стан, коли дані завантажені. У результаті логіка стану стає централізованою, а компоненти залишаються чистими та простими, фактично отримуючи тільки готові значення.

Порівняно з Redux, Zustand позбавляє від надлишку шаблонного коду - немає необхідності визначати типи дій, окремо писати редуктор, окремо додавати підключення до компонентів. Це зменшує розмір кодової бази, пришвидшує написання функціональності і робить сам проєкт більш зрозумілим для нових членів команди. Крім того, невеликий розмір бібліотеки зменшує загальний розмір бандла, що позитивно впливає на швидкість завантаження застосунку, особливо якщо він відкривається на мобільних пристроях.

Завдяки цим властивостям Zustand сьогодні є одним із найбільш зручних рішень для управління станом у React, поєднуючи простоту, високу продуктивність та гнучкість, що дозволяє застосовувати його як у невеликих проєктах.

3.6.5 Workbox

Workbox - це бібліотека від Google, яка значно спрощує роботу з service worker, зменшуючи складність створення та підтримки PWA. Розробка service worker вручну складна, оскільки потрібно писати код для кешування, управління оновленнями, інвалідації кешів, створення стратегій відповіді на запити, а також слідкувати, щоб кеш був консистентним і передбачуваним. Workbox надає вже готові модулі та утиліти, які автоматизують більшість цих процесів, дозволяючи сконцентруватися на логіці застосунку, а не на низькорівневих деталях реалізації.

Одним з головних механізмів Workbox є надійне атомарне попереднє кешування, яке гарантує, що всі критичні ресурси будуть завантажені та доступні оффлайн відразу після інсталяції service worker'а. Workbox генерує список файлів з унікальними хешами і автоматично інвалідовує старі версії, коли з'являються нові збірки.

Другий важливий механізм - це runtime strategies, тобто стратегії обробки запитів під час виконання. Workbox надає набір готових стратегій, таких як Cache First, Network First, Network Only, Cache Only, Stale-While-Revalidate, а також дозволяє комбінувати ці стратегії або створювати свої. Наприклад, критичні ресурси можна завжди брати з кеша, медіаконтент кешувати на вимогу, а API-запити до сервера обробляти за якоюсь іншою.

Таким чином, Workbox вирішує основні проблеми сучасних PWA знижуючи поріг входу для початківців, а також дозволяє будувати складні та масштабовані системи з розумним кешуванням, не ризикуючи потонути в хаосі ручного управління service worker [14].

3.7 Backend технології

3.7.1 Node.js

Node.js - це середовище виконання JavaScript поза браузером, що використовує двигун V8 для компіляції JavaScript у машинний код і

бібліотеку `libuv` для неблокуючого введення-виведення. Основна ідея Node полягає в тому, щоб зробити JavaScript придатним не тільки для роботи з DOM у браузері, але і для серверного програмування, роботи з мережевими API, файловою системою, базами даних і виконанням високонавантажених систем, які мають обробляти тисячі одночасних підключень.

Архітектура Node.js ґрунтується на однопоточній моделі з механізмом `event loop`. Саме `event loop` відповідає за обробку колбеків і асинхронних операцій, дозволяючи уникати блокування. Неблокуючий I/O реалізований завдяки `libuv`, яка працює з системними викликами на рівні операційної системи, використовуючи такі механізми, як `epoll` (в Linux) або `kqueue` (в macOS), що робить можливим масштабовану асинхронну роботу [15].

Для використання апаратних можливостей багатоядерних процесорів у Node існує можливість створювати додаткові процеси, зокрема через `child_process.fork()`, а також застосовувати модуль `cluster`, який дозволяє створювати групи воркер-процесів і розподіляти навантаження між ними. Це дозволяє Node застосовувати паралелізм не за рахунок потоків у одному процесі, а за рахунок множинних процесів, що спільно обслуговують клієнтів.

Node.js містить як асинхронні, так і синхронні API, однак синхронні методи блокують `event loop`, тому їх застосування не рекомендується. Натомість асинхронна модель з колбеками, `Promise` та `async/await` є стандартом. Асинхронний підхід робить Node.js особливо ефективним для систем, де велика частина часу витрачається на очікування відповідей від зовнішніх сервісів, наприклад від бази даних або мережесих ресурсів [15].

3.7.2 Express

Express - це мінімалістичний фреймворк поверх Node.js, який значно спрощує розробку вебзастосунків, додаючи зручну маршрутизацію, механізм проміжних функцій та гнучкий підхід до роботи з HTTP. Він створений так, щоб максимізувати швидкість розробки, при цьому залишаючи розробнику

повний контроль над серверною логікою. Express працює поверх HTTP-модуля Node і фактично робить його більш простим і практичним для повсякденних задач бекенда.

Фреймворк не нав'язує жодної жорсткої структури проєкту. Розробник може сам обирати архітектурні патерни: MVC, MVVM, чисті модулі, гексагональну архітектуру - або будь-яку іншу схему. Це дозволяє використовувати Express як в дуже маленьких сервісах, так і у великих корпоративних системах, де важлива гнучкість налаштування. Також Express без проблем інтегрується з популярними базами даних, такими як MongoDB, PostgreSQL, MySQL, Redis та іншими. Для кожної з цих баз існують готові, перевірені часом драйвери та ORM/ODM, які легко інтегруються в екосистему Express [16].

Ключовим поняттям у Express є *middleware* - проміжні обробники, що працюють через функції з параметрами *req*, *res* та *next*. Саме вони дозволяють додавати будь-яку додаткову логіку між моментом отримання запиту від клієнта та формуванням відповіді. Наприклад, *middleware* може займатися аутентифікацією, логуванням, парсингом JSON, валідацією даних, перевіркою токенів, кешуванням, обробкою помилок та іншими задачами. Така модульність робить Express масштабованим та дає можливість розділяти відповідальність між різними частинами застосунку.

Маршрутизація в Express реалізується через методи типу `app.METHOD(path, callback)`, де *METHOD* - це HTTP-тег, наприклад GET, POST, PUT, DELETE. Шлях може бути простим рядком, регулярним виразом чи навіть складним паттерном. Для роботи зі складними API Express також підтримує Router - можливість створювати окремі міні-маршрутизатори, які потім підключаються до основного застосунку як підмодулі. Це дозволяє ефективно розбивати великий код на логічні домени та створювати чітку структуру модулів [16].

Завдяки такому підходу Express робить розробку вебзастосунків на Node значно швидшою, зрозумілішою та більш зручною, водночас повністю зберігаючи ту гнучкість і контроль, яку дає чистий Node.

3.8 PostgreSQL

PostgreSQL - це реляційна система керування базами даних. Вона зберегла чистоту теорії реляційності, але доповнила це практичною гнучкістю і можливістю адаптуватися під реальні бізнес задачі. Система орієнтована на сувору узгодженість даних, забезпечує коректність транзакцій, складні зв'язки між таблицями, перевірки цілісності і гарантує, що при серйозних навантаженнях, конкурентному доступі та різноманітних операціях з даними, структура залишиться правильною.

З PostgreSQL можна працювати як з класичним реляційним підходом або розширювати модель через механізми системи. У PostgreSQL можна додавати власні типи даних, власні функції, власні оператори, власні методи індексації. Це дає можливість підлаштувати зберігання інформації під специфіку проєкту. PostgreSQL дотримується принципів транзакційного контролю, ізоляції і узгодженості. Це означає, що у середовищі високої конкуренції запитів, коли різні користувачі одночасно читають і змінюють дані, система не дозволяє змінити базу так, щоб вона опинилась у некоректному стані.

Також є підтримка зберігання напівструктурованих даних через типи для роботи з структурованими документами, але без перетворення у текст. Це дозволяє створювати гібридні моделі, де ядро бази використовує реляційну схему, але деякі колонки можуть зберігати гнучкий вміст, який зручно змінювати без перебудови таблиці.

Ця база даних, дає можливість будувати дуже складні структури даних, але при цьому залишає простір для гнучких сценаріїв і розширень - тому її часто обирають для великих систем і її популярність наразі тільки зростає.

3.9 Git

Git – це найпопулярніша система контролю версій, яка дозволяє розробнику зберігати стан проєкту і управляти історією змін. Будь-яка зміна - це окремий об'єкт, який можна відтворити, зберегти, порівняти, з'єднати з іншими або відкинути. Git дає можливість працювати з кодом не лінійно, а паралельно, з декількома гілками розвитку. Таким чином проєкт перестає бути послідовним набором файлів. Кожен комміт - це фіксація певного стану, яка спонукала зробити зміну. І коли треба повернутися назад, знайти причину виникнення помилки, дослідити, коли саме було прийнято неправильне рішення, Git дозволяє робити це без проблем, як точка збереження в відеоіграх. Git пам'ятає всі стани, і може відтворити будь-яку версію так, ніби це поточна.

3.10 Сучасні браузерери та їх API

Сьогодні веб-браузерери - це потужні платформи для запуску складних застосунків. Усі великі браузерери (Chrome, Safari, Firefox та ін.) підтримують стандарти HTML5, CSS і JavaScript і надають численні Web API для роботи з мультимедіа, графікою, комунікаціями, пристроями та іншими функціями. Навіть мобільні браузерери (Android Chrome, Safari на iOS) мають дуже схожий набір можливостей, хоча iOS вимагає використання WebKit-ядра для всіх браузерів.

Сучасний браузер - це практично ОС у вікні, вони мають вбудовані API та рішення, які закривають величезну кількість задач: від рендерингу статичного HTML/CSS до складної апаратно-прискореної графіки, мультимедіа, реального часу та офлайн-режимів. Основні можливості браузеру перераховані нижче.

HTML/CSS/JavaScript - базові технології інтерфейсу. DOM API дає змогу динамічно змінювати документ і працювати з HTML та CSS, а JS-

движок (V8 в Chrome, Nitro в Safari, SpiderMonkey в Firefox) швидко виконує код, змагаючись із рідними застосунками.

Графіка та анімація - Canvas 2D і SVG дозволяють малювати векторні двовимірні об'єкти. Для 3D є WebGL та OpenGL, а також новий WebGPU. WebGPU вже доступний у Chrome і незабаром з'явиться в інших браузерах. Він дозволяє робити високопродуктивні обчислення та складну графіку, як-от обробку штучного інтелекту на GPU.

Мультимедіа - елементи `<audio>` і `<video>` для відтворення потокового чи локального медіаконтенту. Media Source Extensions дають можливість організовувати складні адаптивні трансляції. Через Web Audio API можна обробляти звук (фільтри, мікшери, синтез). Користувачам доступні камера та мікрофон через `getUserMedia`, а WebRTC дозволяє створювати відеоконференції, стрімінги та передачу даних без сервера.

Комунікація – сучасні браузери підтримують HTTP/HTTPS запити через XHR, Fetch API і продовжують впроваджувати SPDY/HTTP2/3. Також є WebSocket та Server-Sent Events для взаємного двобічного спілкування та для стрімінгу даних від сервера. WebRTC забезпечує реальний час (аудіо/відео) прямо між браузерами. Нові API, як WebTransport (двобічна передача даних з низькою затримкою), активно розвиваються.

Оффлайн та сховище - cookies, localStorage/sessionStorage для простих даних, IndexedDB для великих структурованих баз даних. Cache API дозволяє кешувати запити. Service Worker - скрипт, що працює у фоновому режимі; він керує кешем, робить застосунок доступним оффлайн і підтримує Push API (push-повідомлення). Завдяки цьому створюються Progressive Web Apps. На Android Chrome PWA можуть ставати окремими застосунками зі своєю іконкою та сповіщеннями, на iOS Safari поступово додає ці можливості [17].

Пристрої й сенсори - браузери можуть читати дані з апаратури: геолокація (GPS), акселерометр/гірометр, сенсор освітленості, наближення тощо. Для взаємодії з зовнішнім обладнанням впроваджено такі API, як WebUSB, WebBluetooth, WebHID, WebNFC (Safari і Firefox їх поки не

підтримують на iOS через обмеження), а в Chrome - це вже реалізовані функції.

Камера й мікрофон - через стандарт Media Capture. Також є Geolocation API, WebXR для VR/AR (підтримка в Chrome/Safari), WebAuthn (на основі криптографічних ключів для безпечної аутентифікації) Payment Request API для оплати картками/ApplePay, Clipboard API, тощо.

Безпека та приватність - браузері підтримують HTTPS/TLS, Content Security Policy, CORS (контроль доступу між доменами). Для шифрування даних доступний WebCrypto API. Firefox і Safari мають вбудовані фільтри трекерів. WebAuthn (Credential Management) дозволяє замінити паролі на ключі й біометрію. Нові функції як Strict Secure Context (доступ до камери/мікрофона тільки по HTTPS) стало стандартом у всіх браузерах.

Розширення (Extensions) - можливість доповнювати браузер власним кодом. Всі сучасні браузері (Chrome, Edge, Opera, Firefox, Safari) перейшли на WebExtensions API, тому розширення можна писати одразу для кількох браузерів. Chrome використовує Manifest V3, Safari підтримує WebExtensions починаючи з версії 14, Firefox повністю сумісний із повною версією API.

Інші API: File System Access (прямий доступ до файлової системи користувача - підтримує Chrome), WebRTC DataChannel (для надшвидкого обміну даними між клієнтами), Background Sync, Web Locks, Idle API, тощо. Загалом, сучасні браузері зближують можливості нативних застосунків і вебу.

Браузері постійно додають нові API, що раніше були доступні лише в нативному коді, - від 3D/VR до машинного навчання. Важливий тренд – стандартизація (всі три основні рушії - Blink у Chrome, WebKit у Safari, Gecko у Firefox - беруть участь у розробці API) і взаємна сумісність. Chrome зазвичай впроваджує інновації найшвидше, але всі вони сьогодні пропонують розвинений набір веб-технологій, і їхній розвиток продовжується.

3.11 PWA технології та TWA технології

Прогресивний вебзастосунок (PWA) - це вебзастосунок з розширеними можливостями нативного застосунку. Тобто це звичайний сайт-застосунок на базі HTML/CSS/JavaScript, який можна встановити на мобільний пристрій або комп'ютер через браузер [18]. PWA підтримує оффлайн-режим за рахунок Service Worker - фонові JS-програми, що перехоплює мережеві запити і кешує ресурси. Крім того, PWA може відправляти push-повідомлення, виконувати фонову синхронізацію даних і реалізовувати інші функції, властиві нативним застосункам. Завдяки PWA не потрібно окремо розробляти мобільний застосунок під iOS/Android, бо одна кодова база працює на будь-яких платформах і в будь-яких браузерах.

Основними технологіями у PWA є Service Worker і Web App Manifest. Service Worker - це скрипт, що працює у фоновому режимі окремим потоком. Він виконує кешування файлів, відповідає на HTTP-запити й може самостійно відправляти push-повідомлення чи синхронізувати дані з сервером, навіть коли основний застосунок закритий. Маніфест потрібен, щоб браузер розумів налаштування PWA при інсталяції, там передається його іконка, назва, колір заставки та початкової сторінки. На основі цих даних браузер пропонує додати PWA на головний екран, після чого сайт працюватиме як звичайне мобільний застосунок (рис. 3.1) [19].

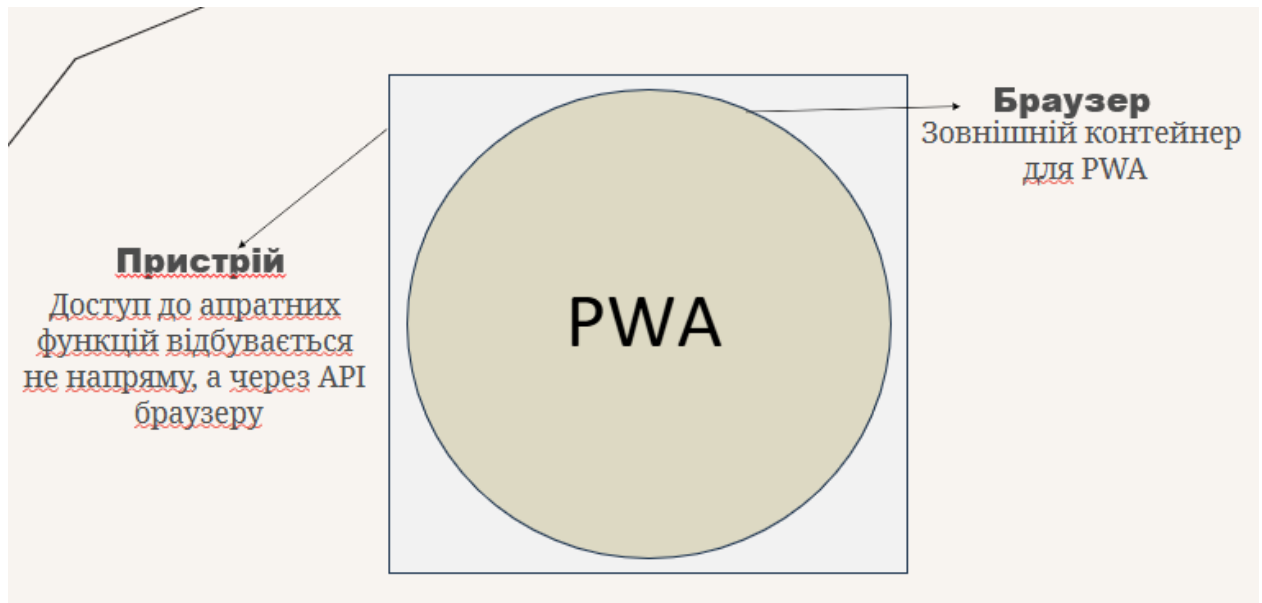


Рисунок 3.1 – Браузер, як обгортка PWA

Завдяки Service Worker PWA може працювати оффлайн - при першому запуску Service Worker завантажує й кешує необхідні файли (HTML, JS, CSS, зображення тощо), а надалі повертає їх із локального сховища, навіть якщо мережа недоступна. Зазвичай реалізують стратегію cache-first - це коли спочатку віддають ресурси з кешу, а потім пробують оновити їх з мережі. Таким чином забезпечується швидке завантаження контенту. Перелік можливих функцій дуже широкий: геолокація, доступ до камери/мікрофону, покупки через API Payment Request, робота з буфером обміну, розпізнавання мовлення тощо. Проте, деякі функції (як AR/VR, WebUSB тощо) можуть бути обмежені на певних платформах.

Існують спеціалізовані інструменти, такі як Workbox - набір бібліотек від Google для управління кешем і роботою Service Worker, PWABuilder від Microsoft, який генерує оптимізовані пакети для різних платформ і допомагає розмістити PWA у магазинах застосунків, як Google Play.

Фактично браузер для PWA – це платформа, яка надає весь доступ до апаратних та внутрішніх функцій, а також керує самим застосунком і його дозволами. PWA може працювати як звичайний веб-сайт, якщо запустити URL у браузері, або як встановлений застосунок, якщо користувач підтвердить інсталяцію. У встановленому стані браузер відображає PWA в

окремому вікні чи повноекранному режимі згідно з маніфестом. Завдяки цьому UI програми не прив'язаний до вкладки браузера - немає адресного рядка чи інструментів навігації. Якщо пристрій втрачає з'єднання з інтернетом, PWA усе одно може продовжувати роботу завдяки Service Worker. Окрім цього, сучасні браузери дозволяють PWA працювати у фоні (через Background Sync, Push API тощо) і навіть отримувати оновлення автоматично без прямого залучення користувача.

Також існує Trusted Web Activity - це спосіб запускати PWA всередині звичайного Android-застосунка з використанням технології Chrome Custom Tabs. Це інструмент, який відкриває PWA у повноекранному вікні браузера без видимого інтерфейсу Chrome. У TWA весь інтерфейс забезпечує браузер на пристрої - але оскільки контент вважається довіреним, Chrome запускає його без адресного рядка і дозволяє розміщення в нативних мобільних магазинах зостосунків, як Google Play. Таким чином це Android-застосунок оболочка, що відображає PWA і дозволяє розширити нативні можливості застосунка.

Великою перевагою є автоматичне оновлення застосунку - розробник оновлює код на сервері і при наступному підключенні Service Worker самостійно завантажує нові файли та активує нову версію застосунку. Тобто користувач не проходить ніяких ручних оновлень, не бачить сповіщень маркета, не чекає установку патчів. У PWA концепція апдейта працює так само, як оновлення звичайного сайту - розробник деплоїть новий білд, Service Worker його кешує, і користувач просто має найновішу версію без зайвих кроків, що особливо критично для великих систем з частими релізами та безперервною інтеграцією.

PWA і TWA – це, коли браузер сприймається, як ОС. Такий підхід зменшує межу між вебom та нативними застосунками, дозволяючи використовувати одну кодову базу на всіх платформах.

3.12 MVC

Архітектурний шаблон MVC – це спосіб організації застосунку, при якому логіка розділяється на три незалежні складові: модель, уявлення і контролер. MVC дозволяє уникнути змішування різних логік, дає можливість тестувати кожен рівень окремо та пришвидшує розробку великих систем. Завдяки MVC легко змінити дизайн, не торкаючись бізнес-логіки, розширити бізнес-правила, не змінюючи UI, або провести міграцію даних, не чіпаючи екранів і контролерів. Саме тому MVC став фундаментом для багатьох фреймворків веброботки. У центрі ідеї MVC - незалежність шарів, роль кожного шару строго визначена і зрозуміла, а застосунок стає масштабованішим, керованішим та значно легшим у супроводі.

3.13 REST API

REST API - це стиль побудови вебсервісів, при якому клієнт і сервер спілкуються між собою через звичайні вебзапити, використовуючи стандартні методи протоколу HTTP. Його створили для того, щоб зробити взаємодію між застосунками простою, передбачуваною з легким масштабуванням. Ідея REST полягає в тому, що сервер дає доступ не до функцій у вигляді виклику команди, а до ресурсів у вигляді запиту. Якщо клієнт хоче отримати дані - надсилається HTTP-запит на адресу ресурсу, сервер відповідає структурованими даними, зазвичай у форматі JSON. У REST немає стану сесії на сервері - кожен запит повинен містити всю необхідну інформацію для обробки. Також сервер не має запам'ятовувати контекст між запитами. Це дозволяє відокремити інтерфейс від внутрішньої логіки системи, а також дає можливість мати багато клієнтів на один бекенд.

3.14 MVVM

Шаблон MVVM - це архітектурний підхід, який з'явився щоб відокремити дані і бізнес-логіку від представлення, але при цьому зробити цей зв'язок максимально швидким. Якщо у MVC контролер керує оновленням представлення, то у MVVM все побудовано на двосторонньому зв'язуванні даних, де View просто дивиться на ViewModel, а ViewModel містить стан і логіку для цього View.

MVVM особливо добре підходить для сучасних фронтенд фреймворків, як Angular, Vue, React з хуками та Zustand/MobX/Recoil, бо сама технологія забезпечує механізми автоматичного оновлення представлення при зміні стану. Саме за рахунок зсуву, від імперативного управління UI до декларативного, MVVM дозволяє писати застосунки масштабніші, зрозуміліші та простіші. Таким чином інтерфейс автоматично реагує на зміну бізнес-стану, і фронтенд перестає бути набором обробників подій, а перетворюється на відображення єдиного ресурсу правди.

3.15 Висновки до розділу

У третьому розділі було розглянуто комплекс інструментів та алгоритмів, що формують основу розробки сучасного прогресивного вебзастосунку (PWA). Обраний технологічний стек (рис. 3.2), побудований на JavaScript та TypeScript, забезпечує високу продуктивність, гнучкість і масштабованість завдяки комбінації фронтенд-технологій (React, Zustand, Less, Vite) з бекенд-компонентами (Node.js, Express, PostgreSQL) та інструментами DevOps (Docker, PNPM, Git). Середовище розробки WebStorm від JetBrains оптимізує процес кодування за рахунок інтелектуального автодоповнення та аналізу, тоді як Vite прискорює збірку, а PNPM - управління залежностями.

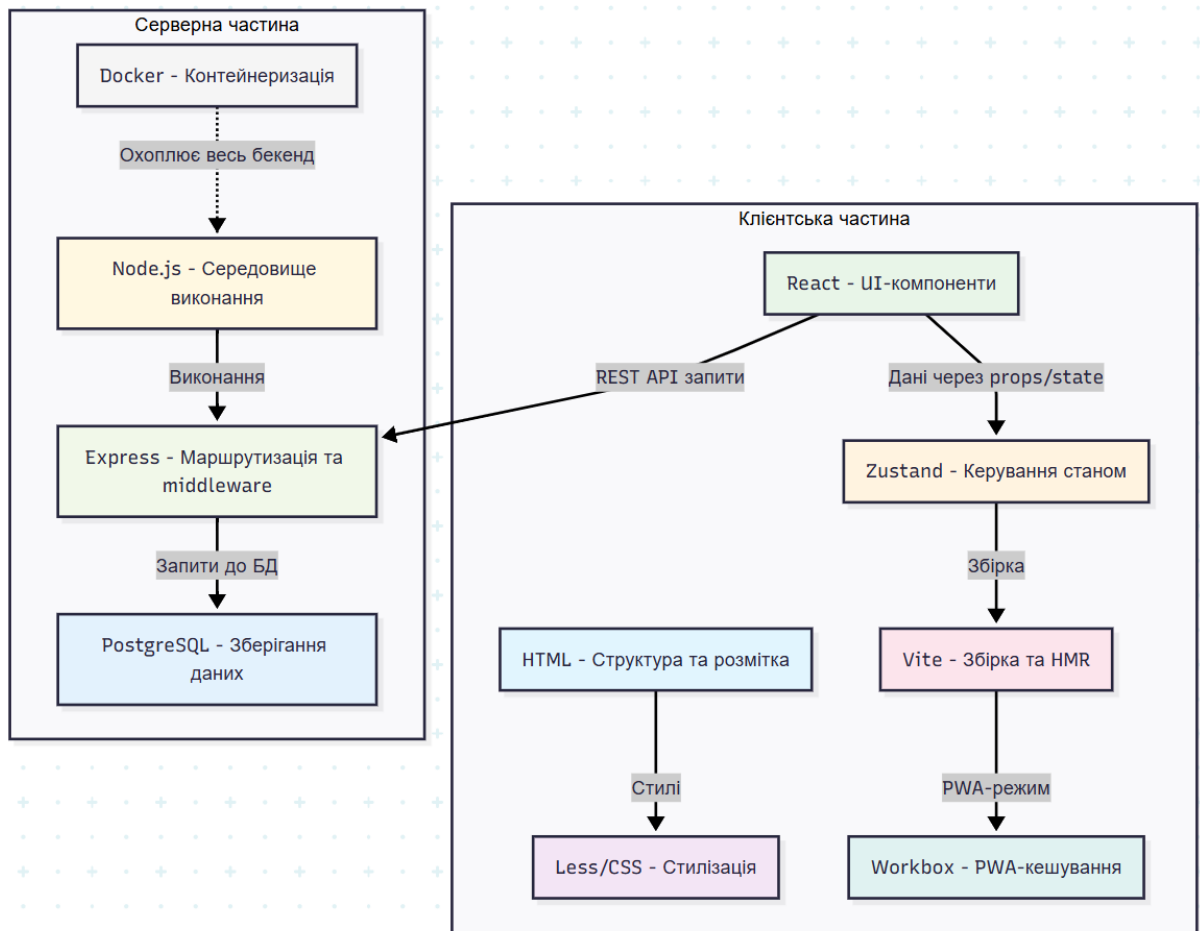


Рисунок 3.2 – Зв’язок технологій у проєкті та їх відповідальність

Ключові архітектурні шаблони, такі як MVC для клієнт-серверної взаємодії з REST API та MVVM для PWA-логіки, дозволяють чітко відокремити UI від бізнес-логіки, забезпечуючи реактивність і легкість тестування. Реалізація PWA-функціоналу через Workbox робить застосунок доступним оффлайн, з автоматичним оновленням і нативною інтеграцією на мобільних платформах, наближаючи веб до можливостей нативних застосунків.

Загалом, цей стек не лише прискорює розробку та розгортання, але й сприяє створенню надійної, кросс-платформової системи, готової до реальних навантажень.

4 ОПИС ПРОГРАМНОЇ РЕАЛІЗАЦІЇ

Програмна реалізація є комплексним рішенням, розробленим для ефективного управління процесами доставок та збору звітів, з акцентом на гнучкість, надійність та адаптивність. Ця реалізація базується на принципах сучасної веб-розробки, де PWA виступає, як автономний клієнтський застосунок, здатний працювати без постійного підключення до мережі, бекенд обробляє серверну логіку та зберігання даних, а Docker гарантує ізоляцію та легкість управління залежностями.

4.1 PWA застосунок

Прогресивний вебзастосунок, розроблений для ефективного управління процесами доставок та збору звітів про доставки, забезпечуючи безперебійну роботу в умовах відсутності інтернет-з'єднання завдяки принципу offline-first. Всі операції спочатку виконуються локально, а потім відбувається синхронізація з сервером при відновленні мережі. Такий підхід дозволяє використовувати єдину кодову базу для більшості платформ і не залежить від бекенду реалізуючи основний бізнес функціонал.

Локальні дані зберігаються в доступній в браузері базі даних IndexedDB через бібліотеку Dexie, забезпечуючи швидкий доступ і маніпуляцію з ними без потреби в постійному підключенні. Чутливі дані записуються в зашифрованому вигляді завдяки Crypto API браузеру. Також для зменшення об'єму медіа даних використовується бібліотека рако, яка архівує медіа у форматі Base64.

Підтримка PWA-функцій, дозволяє встановлювати застосунок як нативний на пристрої користувача з можливістю роботи в автономному режимі з отриманням push-сповіщень. Інтеграція з камерою та сканерами для роботи з QR-кодами та фотографіями документів, робить процес збору звітів доставки швидким, без необхідності в додатковому забезпеченні. Для зчитування QR кодів використовується бібліотека jsQr, а для доступу до

камери використовується API WebRTC MediaTrack.

Інтерфейс застосунку орієнтований на мобільні пристрої, проте має адаптацію під будь які розміри екрану. Головний екран наведено на рисунку 4.1.

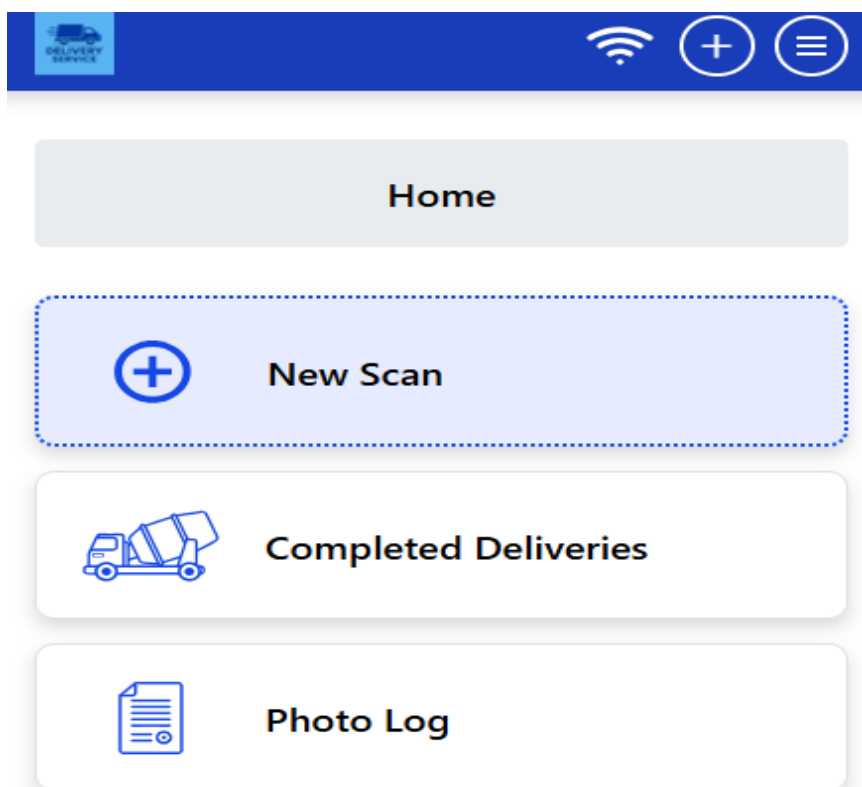


Рисунок 4.1 – Головний екран застосунку на телефоні

Оптимізації та продуктивність досягаються через code splitting з lazy loading, оптимізацію зображень та кешування, роблячи застосунок швидким та ефективним.

4.1.1 Архітектура

Архітектурні принципи, на яких базується цей проєкт, включають чітке розділення відповідальностей через багат шарову MVVM-подібну архітектуру, де кожен шар має свою спеціалізовану роль. Також використовується паттерн dependency injection через composition root, реалізований у ModelRootComposer, який забезпечує централізоване

створення та налаштування всіх залежностей, роблячи ініціалізацію застосунку передбачуваною та легкою для тестування, оскільки всі компоненти конфігуруються в одному місці, що полегшує інтеграцію нових модулів чи заміну існуючих без впливу на решту системи.

Структура проєкту організована в чітку ієрархію папок, де Core містить конфігурації, декоратори, сутності, допоміжні функції, інтерфейси та базові сервіси, такі як шифрування та архівація. Model фокусується на бізнес-логіці з bootstrap-компонентами для ініціалізації, функціональними модулями для аутентифікації, доставок, доказів, інформації про користувача та синхронізації, а також інфраструктурними елементами. ViewModels керує станом застосунку через спеціалізовані модулі для, аутентифікації, камери, доставок, доказів, інформації користувача та синхронізації, а View відповідає за UI-компоненти та віджети, що разом створює модульну та масштабовану структуру, де кожен елемент легко ізолювати та розвивати незалежно.

У Core визначені базові сутності, такі як TDelivery для опису доставок, TEvidence для доказів, TDocumentImage для зображень документів та TMeInfo для інформації про користувача, а сервіси забезпечують функціональність на кшталт EncryptedStorageService для захищеного зберігання. EncryptorService використовується для шифрування та дешифрування даних, ArchiveService для стиснення за допомогою рако, а також SyncNotificationsSetup для налаштування сповіщень про синхронізацію. Декоратори на зразок RequestDecorator обгортають HTTP-запити з обробкою помилок, а StorageDecorator забезпечує роботу з зашифрованим сховищем. Допоміжні функції, включають DateTimeFormatter для форматування дат, EntitySyncStatusParser для парсингу статусів синхронізації, Hasher для хешування, ImageLoader для завантаження зображень та JpegHandler для обробки JPEG.

Model, як шар бізнес-логіки, застосовує паттерни repository для доступу до даних, gateway для зовнішніх API, mediator для координації між ними, facade для спрощення доступу та налаштування до модулів моделі та систему синхронізації. DeliveryGateway обробляє API-запити для створення

та отримання доставок, DeliveryMediator координує локальні та віддалені дані, реєструючи функції для черги синхронізації, а DeliverySyncEntityFacade надає публічний API для отримання, відправки та додавання доставок з автоматичною обробкою онлайн/офлайн-сценаріїв. Bootstrap-компоненти - це ModelRootComposer який створює та зберігає залежності та ModelOrchestrator який оркеструє модель, відстежуючи зміни токена, мережевого статусу, запускаючи polling і т.д.

ViewModel шар, відповідальний за управління станом, використовує Zustand для кешу з реактивністю до змін, інтегруючись з Model шаром через фасади (рис. 4.2).

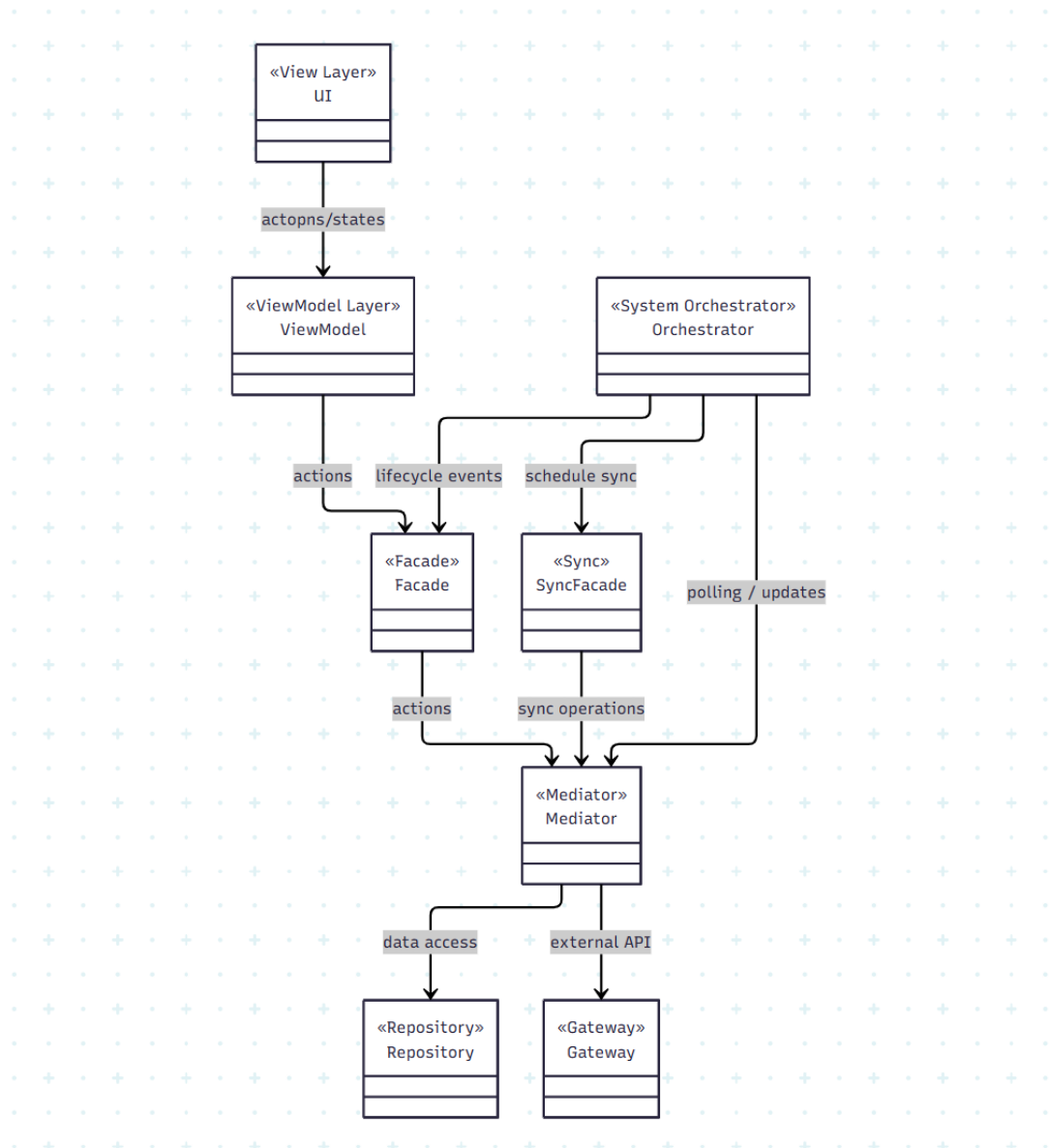


Рисунок 4.2 – Архітектура PWA застосунку

View шар, як шар представлення, побудований на React-компонентах з модульною структурою, де кожен модуль має точку входу, компоненти, сторінки, контексти, хуки та сервіси. AuthModule обробляє авторизацію, HomeModule надає головну навігацію, CameraModule інтегрує сканери та фотографування, DeliveryModule відображає списки та деталі доставок, а EvidenceModule керує доказами, з спільними елементами для повторного використання.

Паттерни проєктування, застосовані в проєкті, включають repository для абстракції даних з базовим класом AEntityRepository та конкретними реалізаціями, що полегшує заміну джерел та тестування, gateway для ізоляції API з обробкою помилок, mediator для розв'язання залежностей та централізованої синхронізації, facade для приховування складності з простим публічним API, strategy для вибору обробки залежно від онлайн/офлайн-стану, observer для реакції на зміни через live queries та event listeners, а composition root для централізованого створення залежностей, роблячи архітектуру гнучкою та розширюваною.

Модульна структура деталізує функціональність кожного модуля, наприклад, delivery-модуль дозволяє створювати доставки з фото, переглядати списки з фільтрацією, деталі та відправку, з компонентами на зразок DeliveryHistoryPage та DeliveryDetails, evidence-модуль аналогічно обробляє докази з прив'язкою до доставок, camera-модуль забезпечує сканування QR, фотографування та обробку зображень з UI на зразок ScannerPage та DocumentPhotoPage, а auth-модуль керує входом, токенами та захистом маршрутів через SessionPolicy. Важливо що модулі кожного шару незалежні один від одного і не спілкуються напряму між собою. У View спільні UI компоненти знаходяться в Shared, а навігація і основні налаштування, а також сервіси пов'язані виключно з View шаром чи React знаходяться в App. В Model використовується ООП підхід і більш чітка структура тому більшість загальної логіки вже описана в абстракціях, а універсальна логіка знаходиться в Core [20].

Архітектура побудована на принципах offline-first, розділення відповідальностей, розширюваності, надійності, продуктивності та безпеки, використовуючи сучасні паттерни та технології. Такий підхід дозволяє перевикористовувати код, швидко розширювати і тестувати застосунок.

4.1.2 Реалізація PWA

PWA-функціональність включає service worker через vite-plugin-pwa для кешування з стратегіями StaleWhileRevalidate та NetworkOnly, manifest для іконок, кольорів та display mode, а також сповіщення синхронізації через background sync.

Реєстрація Service Worker виконується через vite-plugin-pwa з використанням віртуального модуля virtual:pwa-register. Стратегія оновлень базується на параметрі registerType: "autoUpdate", що дозволяє плагіну самостійно перевіряти та оновлювати сервіс-воркер. При виявленні нової версії через подію onNeedRefresh застосунок автоматично перезавантажується для застосування змін, забезпечуючи актуальність коду без необхідності ручного оновлення користувачем (див. Додаток А, підрозділ А.1).

Для обробки помилок під час розгортання нової версії, реалізовано механізм в ErrorBoundary.tsx, де в методі componentDidCatch перевіряється, чи є помилка пов'язана з оновленням PWA через функцію isPWAUpdateError, і якщо так, то виконується до двох спроб перезавантаження з повідомленням "New version available. Reloading...". Якщо помилка зберігається, обробляється стандартним чином, що запобігає зацикленню та забезпечує стабільність застосунку під час оновлень (див. Додаток А, підрозділ А.2).

Конфігурація PWA здійснюється через Vite з інтеграцією Workbox та генерації маніфесту, де в файлі vite.config.ts активується VitePWA. Маніфест формується динамічно з використанням змінних середовища ENV, включаючи параметр display: "standalone" для повноекранного режиму після установки, як нативного застосунка. Також підключається масив іконок для

різних платформ, таких як Android з розмірами 512x512, 192x192, 144x144 та 48x48 пікселів у форматі PNG, Windows 11 з розмірами 16x16, 32x32, 72x72, 96x96 та 256x256, та iOS з 180x180 та 1024x1024, що забезпечує оптимальне відображення на різних пристроях (див. Додаток А, підрозділ А.1).

Workbox налаштований для предзавантаження ресурсів з параметрами `navigateFallback: "/index.html"` для обробки SPA-маршрутів, `skipWaiting: true` та `clientsClaim: true` для швидкого активації нового сервіс-воркера. Для кешування статичних файлів з розширеннями `js`, `css`, `html`, `woff2` та `png`, використовується `globIgnores` з флагом `StaleWhileRevalidate` та кеш-іменем `"static-assets"` для швидкого доступу з кешу з фоновим оновленням. Для API-запитів реалізована окрема система зберігання даних і окрема система синхронізації для більшого контролю, і через необхідність важчої обробки ніж пропонується в стандартному рішенні. Для виключення з автоматичного кешування запитів з префіксом `VITE_API_URL` використовується флаг `NetworkOnly` (див. Додаток А, підрозділ А.1).

Підключення маніфесту та іконок реалізовано в файлі `index.html`, де через тег підключається генерований маніфест, а також набір для Android, iOS та Windows з різними розмірами та типами, забезпечуючи сумісність з різними платформами та автоматичне відображення іконок під час установки (див. Додаток А, підрозділ А.3).

Сповіщення для нагадування про необхідність синхронізації для користувачів та спроби автоматичної синхронізації реалізовані як фонові завдання, такі як `Background Sync` та `Periodic Background Sync`, через окремий кастомний сервіс-воркер у файлі `public/sync-notification-sw.js`. Реєстрація відбувається при старті застосунку в `index.tsx` через виклик `setupNotifications` перед `registerSW`. Налаштування в `SyncNotificationsSetup.ts` перевіряє дозвіл на сповіщення через `Notification.permission` з запитом при необхідності. Якщо дозвіл надано, реєструє сервіс-воркер з `scope '/'`, який чекає на готовність через `navigator.serviceWorker.ready`. Для планування перевірок черги синхронізації реєструється `Periodic Background Sync` з тегом `'pending-count-check'` та `minInterval periodicSyncMinInterval`, з логуванням успіху чи

помилки. Periodic Background Sync підтримується не в усіх браузерах, тому передбачено фолбек на setInterval для забезпечення хоча б обмеженої функціональності. Також використовується Background Sync з тегом 'send-notification' і перевіряється наявність з graceful degradation, а як фолбек використовує setInterval для checkAndNotify з fallbackInterval (див. Додаток А, підрозділ А.4).

PWA розроблений адаптивним до різних середовищ, зберігаючи базову оффлайн-логіку та синхронізацію незалежно від підтримки розширених API, тим самим забезпечуючи стабільність та користувацький досвід на Chrome, Firefox, Safari та інших сучасних браузерах.

4.1.3 Система оффлайн обробки даних

Загальна схем (рис. 4.3) – це потік від ViewModel Layer вниз до Facade Layer у Model, де фасади діють як єдина точка входу, далі до Mediator Layer для прийняття рішень, з розгалуженням на Gateway для мережеских запитів або до Repository для локального зберігання в IndexedDB через Dexie. Потім після отримання результату дії обидва сценарії сходяться в Adaptor Layer для уніфікації даних та Services для допоміжних операцій, забезпечуючи, що дані обробляються послідовно, з валідацією на кожному етапі. Система може працювати як в онлайн, так і в оффлайн режимах, з автоматичним перемиканням стратегій для збереження продуктивності та цілісності інформації.

Фасади, слугують точкою входу для ViewModel, приховуючи деталі взаємодії з медіаторами, репозиторіями та мережею, надаючи простий API для операцій, де базовий клас ASyncEntityFacade параметризується типами T для сутності, TDB для бази даних та R для репозиторію. Фасади валідують дані, оновлюють локальну базу перед відправкою, викликають медіатори для логіки, обробляють помилки з поверненням структурованих результатів на зразок TDelivery чи IEntityModelError, і надають методи для чисто локальних операцій, таких як addLocalDelivery чи updateLocalDelivery, роблячи

інтерфейс простим і захищеним від внутрішньої складності, з фокусом на користувацький досвід без блокувань.

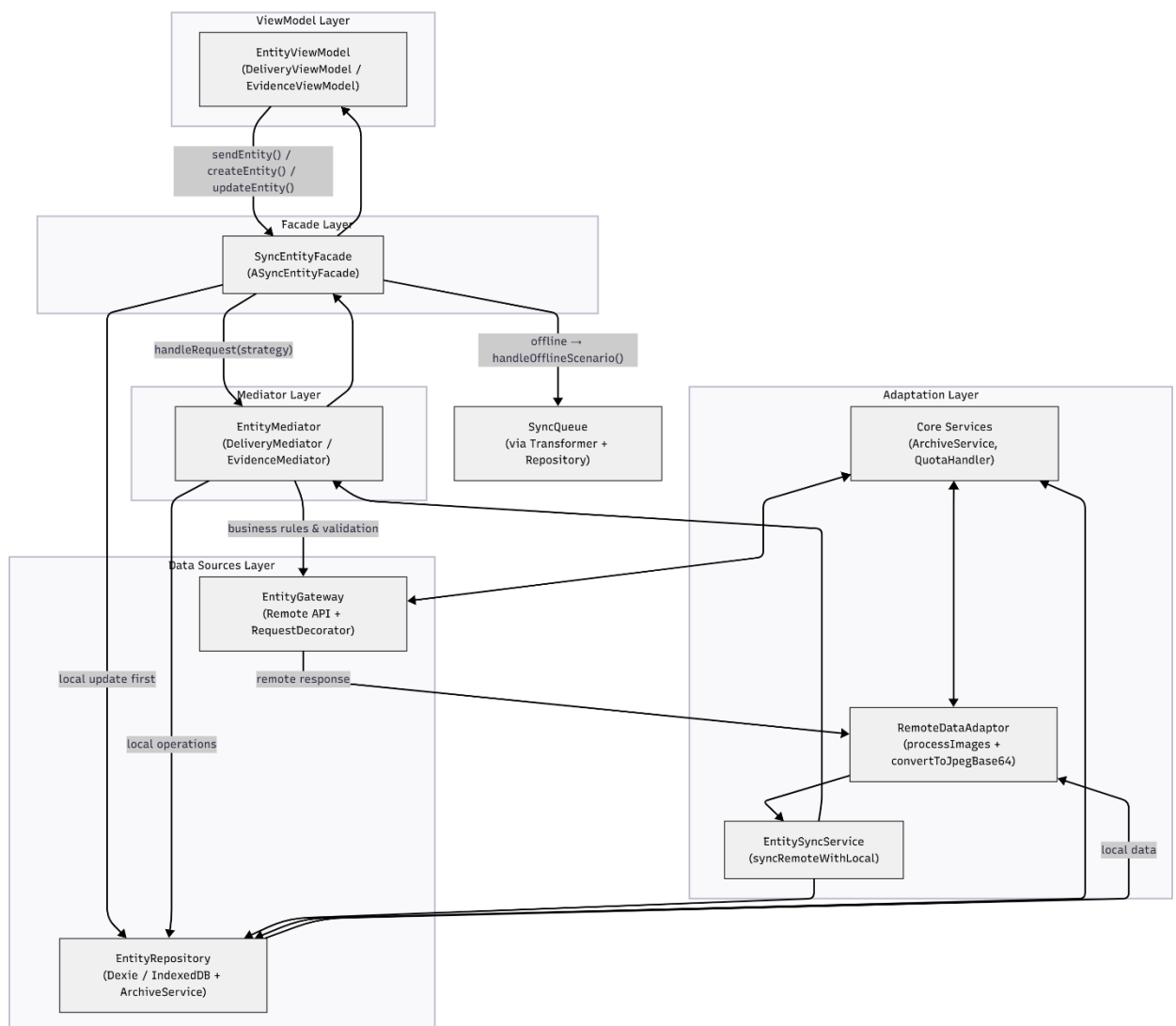


Рисунок 4.3 – Система оффлаїн обробки даних

Медіатори, як Mediator Layer, координують роботу між Gateway та Repository, реалізуючи бізнес-правила та вирішуючи, коли звертатися до API, а коли використовувати локальні дані. Основні методи медіаторів включають `getAllDeliveries`, який приймає опції для пагінації, фільтрації, сортування та пошуку. `localOnlyStatuses` для статусів, що існують тільки локально, який перевіряє, чи фільтр чи пошук стосується тільки локальних даних, якщо так - повертає з `repository.getAll`, інакше отримує з `remote` через `deliveryGateway.getDeliveries`, потім у `syncService.syncRemoteDeliveries`

синхронізує з опцією `syncEvidence` для підтягування звітів. Потім отримує локальні дані з `repository` та комбінує з пагінацією з `remote`, повертаючи `IPaginateResult` або масив `TDelivery`. `createDelivery` викликає `gateway.createDelivery`, при помилці оновлює статус на `UploadError` та кидає помилку, або синхронізує, перевіряє на видалення після створення та повертає синхронізовану `TDelivery`. Реєстрація функцій `bulkCreateDeliveries` та `createDelivery` потрібні для черги синхронізацій, що дозволяє медіаторам координувати асинхронну синхронізацію, забезпечуючи бізнес-правила, такі як перевірка статусів чи залежностей, роблячи шар ключовим для логіки без прив'язки до UI чи зберігання.

Gateway, як шар для роботи з віддаленим API, формує запити, обробляє відповіді та перетворює дані. Декоратор `Request` у `resource` та `options`, додає `Authorization` з токеном з локального сховища якщо `isNeedAuth`, виконує `fetch`, обробляє 401 статус через подію `unauthorized`, логує помилки та повертає `Response` або помилковий статус 500, що дозволяє Gateway формувати HTTP-запити з GET, POST та `FormData`, додавати авторизацію, обробляти помилки з поверненням структурованих результатів, стискати зображення перед відправкою та перетворювати локальні формати в API, забезпечуючи ізоляцію мережевої логіки.

Репозиторії, як шар для локальної бази даних `Dexie/IndexedDB`, зберігають, отримують, оновлюють та видаляють дані з пошуком та фільтрацією. Базовий клас `AEntityRepository`, визначає абстрактні методи для `table`, `mapDBToEntity` та `prepareEntityForDB`, а `decodeImages` у рядках декодує `compressedImages` з ASCII85, `decompress` та парсить JSON для `TDocumentImage[]`. `prepareEntityForDB` серіалізує `images`, стискає, кодує в ASCII85, визначає `mainImageId`, генерує `uuid`, парсить статус, формує `TDeliveryDB` з `timestamps`. Обробка `QuotaExceededError` у `AEntityRepository` у через `withQuotaClearRetry` намагається виконати операцію, при помилці чистить `old completed` через `clearOldCompleted` та повторює, а також обробляє `AbortError` з `inner QuotaExceededError` [21]. Пошук та фільтрація у `AEntityRepository` створює `collection` з `where.anyOf` для фільтрів, `orderBy` для

сортування, filter для пошуку, reverse для desc, рахує totalCount для пагінації, offset та limit, toArray, map до entity, повертає IRepositoryResult з IPaginateResult або масивом.

Адаптери, перетворюють дані між API та локальною моделлю, де базовий ARemoteDataAdaptor ініціалізує imageWorker для обробки зображень, onmessage для відповідей, onerror для помилок, adaptForRepository оновлює existing з remote через updateFromRemote, shouldDelete перевіряє deleted. Обробка зображень у ARemoteDataAdaptor.ts processImages приймає remoteImages, для кожного fetch, перевіряє contentType, для image конвертує blob в jpeg base64 через convertToJpegBase64 з quality 0.8, формує TDocumentImage з id, image, coords, taken_at, для pdf логує помилку, для інших кидає unsupported, при помилці додає з imageUrl як fallback, що дозволяє адаптерам перетворювати поля, статуси, завантажувати та обробляти зображення, генерувати UUID, визначати видалення.

Приклад потоку даних для створення доставки - ViewModel викликає deliveryFacade.sendDelivery, facade оновлює локально через repository.updateSingle з isSend true та UploadInProgress, викликає mediator.createDelivery, mediator делегує до gateway.createDelivery, gateway стискає image через smart64BaseCompress, формує FormData з file, coords, dates, відправляє POST через Request, mediator отримує TRemoteDelivery, викликає syncService.syncRemoteDeliveryWithLocal, syncService використовує adaptor для перетворення, adaptor processImages з fetch та convertToJpegBase64, syncService зберігає через repository.upsert, repository стискає images, кодує ASCII85, зберігає в Dexie, забезпечуючи повний цикл з локальним збереженням, мережевою відправкою та синхронізацією [22].

4.1.4 Система синхронізації даних

Система синхронізації даних є одним з найважливіших компонентів, що забезпечує offline-first підхід, дозволяючи накопичувати зміни локально та автоматично синхронізувати їх з сервером при відновленні мережі.

Система самостійно обробляє черги, оптимізацію, пріоритизацію та помилки, роблячи процес надійним, ефективним і стійким до збоїв (рис. 4.4).

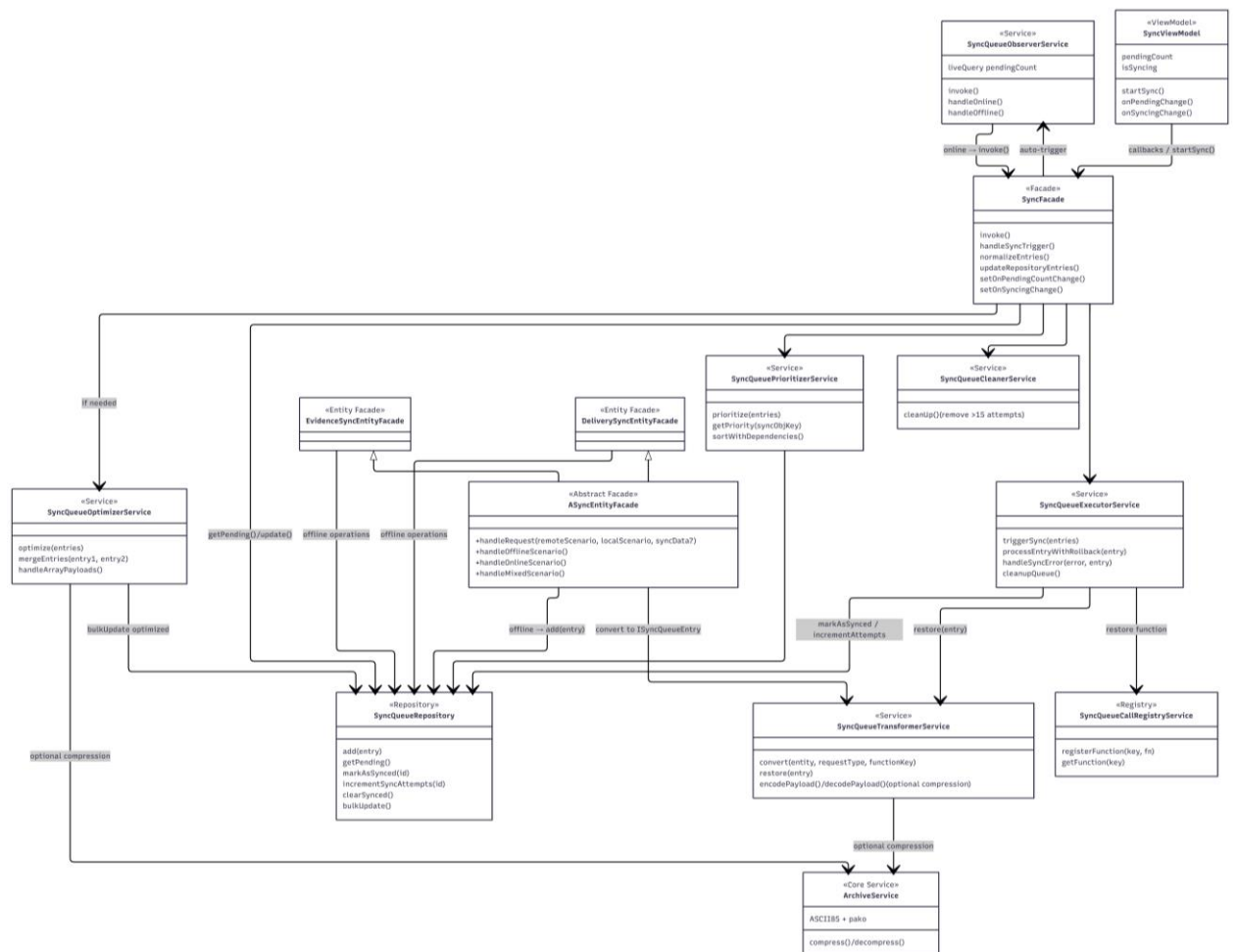


Рисунок 4.4 – Система синхронізації даних

SyncService порівнює локальні та віддалені копії сутностей, визначає необхідність оновлень на основі хешів чи дат змін, і запускає каскадні операції, такі як підтягування залежних звітів при синхронізації доставки. SyncFacade діє як оркестратор, запускаючи обробку черги, відстежуючи стан синхронізації та інтегруючи сповіщення про несинхронізовані елементи, що дозволяє системі реагувати на події мережі [23].

Для збереження історії дій використовуються таблиця `syncQueue` у `IndexedDB`. Для черги синхронізації з полями `id`, `isSynced`, `addedAt`, `syncAttempts`, `requestAction` (ключ зареєстрованої функції для виконання), `requestType` (тип запиту, наприклад, `CREATE` чи `UPDATE`), `payload`

(серіалізовані дані запиту, можливо стиснені) та `syncObjKey` (ключ сутності, як "Delivery" чи "Evidence").

Формування черги синхронізації починається з користувацької дії, наприклад, виклику `sendDelivery` у `DeliverySyncEntityFacade`, який делегує до `ASyncEntityFacade.handleRequest` у файлі `ASyncEntityFacade.ts` де спочатку намагається виконати віддалений запит, але при оффлайн чи помилці перемикається на локальний сценарій через `handleOfflineScenario`. Якщо надані `syncData` (з ключем `registeredFunctionKey`, `payload`, `requestType` та `syncObjKey`), створюється запис у черзі через `syncTransformer.convert`, додається до `syncQueueRepository`, а потім виконується локальна операція, повертаючи результат або помилку, що дозволяє користувачу бачити успішну операцію негайно, навіть без мережі, з маркуванням статусу як `UploadInProgress` [24].

`SyncQueueTransformerService` серіалізує `payload` у строку, якщо потрібно стискає через `ArchiveService.ascii85Encode` після `compress`, і формує `ISyncQueueEntry`.

Ключ `registeredFunctionKey` реєструється медіатором. Метод `register` додає функції до `SyncQueueCallRegistryService` для `BULK_CREATE_KEY` та `CREATE_KEY`, асоціюючи їх з методами `bulkCreateDeliveries` та `createDelivery` і аналогічно для інших дій: видалення та оновлення, що дозволяє пізніше відновити та виконати точну дію під час синхронізації, роблячи чергу гнучкою.

`SyncQueueObserverService.ts` використовує `liveQuery` для моніторингу `getPending`, маркує `hasOfflineChanges` при оффлайн, а при онлайн через `handleOnline` отримує `pending` та викликає `syncHandler`, реагуючи на події `window.addEventListener('online')` та `'offline'` для автоматичного запуску синхронізації.

Очищення та оптимізація включають `SyncQueueCleanerService` для видалення записів з `syncAttempts > 15`, щоб уникнути накопичення невдалих спроб, та `SyncQueueOptimizerService`, яка розбиває записи через `splitPayload`, групує за сутностями через `groupByEntity`, а потім зливає групи через

mergeGroups для зменшення зайвих операцій. Наприклад, CREATE + UPDATE стає одним CREATE з останніми даними, зменшуючи кількість запитів [25].

Пріоритизація в SyncQueuePrioritizerService сортує записи за пріоритетом з getPriority, враховуючи залежності та addedAt для хронологічного порядку, забезпечуючи, щоб Delivery синхронізувалися перед Evidence.

SyncQueueExecutorService перевіряє shouldProcess, встановлює syncing true, чистить чергу, обробляє записи через processEntriesWithSafety з processEntryWithRollback, де transformer.restore відновлює функцію, виконує її, маркує asSynced при успіху або інкрементує attempts при помилці, перериваючи при мережових помилках через shouldAbort = true.

Синхронізація сутностей деталізується для deliveries, де в DeliverySyncService метод syncRemoteDeliveries приймає масив віддалених доставок. Для кожної перевіряє локальну доставку по remote_id через getDeliveryByRemoteId, викликає checkAndSyncDelivery з опцією syncEvidence для каскадного підтягування доказів, повертаючи синхронізовані TDelivery. У DeliveryMediator createDelivery викликає gateway.createDelivery, при помилці маркує UploadError, інакше синхронізує через syncService.syncRemoteDeliveryWithLocal, перевіряючи на видалення після створення.

Для evidences аналогічно, але EvidenceSyncService перевіряє shouldDeleteEvidence, підтягує delivery через deliveryMediator.getByRemoteId якщо відсутній локально, кидаючи помилку якщо mediator не встановлений, забезпечуючи цілісність залежностей.

Приклад сценарію де спочатку користувач оффлайн, а потім переходить в онлайн - при створенні доставки оффлайн sendDelivery оновлює чергу синхронізації, ставить isSend true, статус UploadInProgress, додає до черги з ключем DeliveryMediator.create. При появі мережі observer запускає SyncFacade, оптимізована та пріоритизована черга обробляється executor, відновлюється createDelivery з запитом та syncRemoteDeliveryWithLocal, статус змінюється на uploadedSuccessfully, запис маркується asSynced, кеш

/pending-count оновлюється, а SW показує сповіщення якщо черга не порожня.

4.2 Backend сервіс

Backend в рамках данної дипломної роботи - це утилітарна система для підтримки PWA та виконує переважно роль віддаленого сервісу для збереження даних. Проєкт притримується REST API принципів та побудований на базі Express.js, як легкого та гнучкого фреймворку для обробки HTTP-запитів, а також використовує PostgreSQL базу даних. Система фокусується на ключових аспектах, таких як управління доставками з детальними статусами та координатами, звітами доставки з прикріпленими файлами та коментарями, а також користувачами з аутентифікацією.

Архітектура проєкту слідує класичному MVC, де кожен шар має чітку відповідальність. Controllers для обробки HTTP-запитів та відповідей, де вони валідають вхідні дані, викликають сервіси та повертають JSON-відповіді з статусами. Services для бізнес-логіки, де реалізуються правила на зразок перевірки аутентифікації, створення сутностей чи оновлення статусів. Repositories використовуються для абстракції над базою даних, де виконуються SQL-запити через pg-модуль, забезпечуючи ізоляцію від конкретної БД. Databases відповідає за управління схемами, де створюються таблиці при старті, з перевіркою existence для уникнення помилок.

Разом така система створює чисту, модульну структуру, де зміни в одному шарі не впливають на інші, полегшуючи підтримку та розширення.

4.3 Docker

Docker використовується для створення середовища для Node.js бекенду з базою даних PostgreSQL, де Dockerfile визначає, як побудувати образ для бекенду, а docker-compose.yml оркеструє запуск сервісів,

дозволяючи швидко підняти повне середовище розробки з hot reload, персистентними даними та залежностями між контейнерами (рис. 4.5).

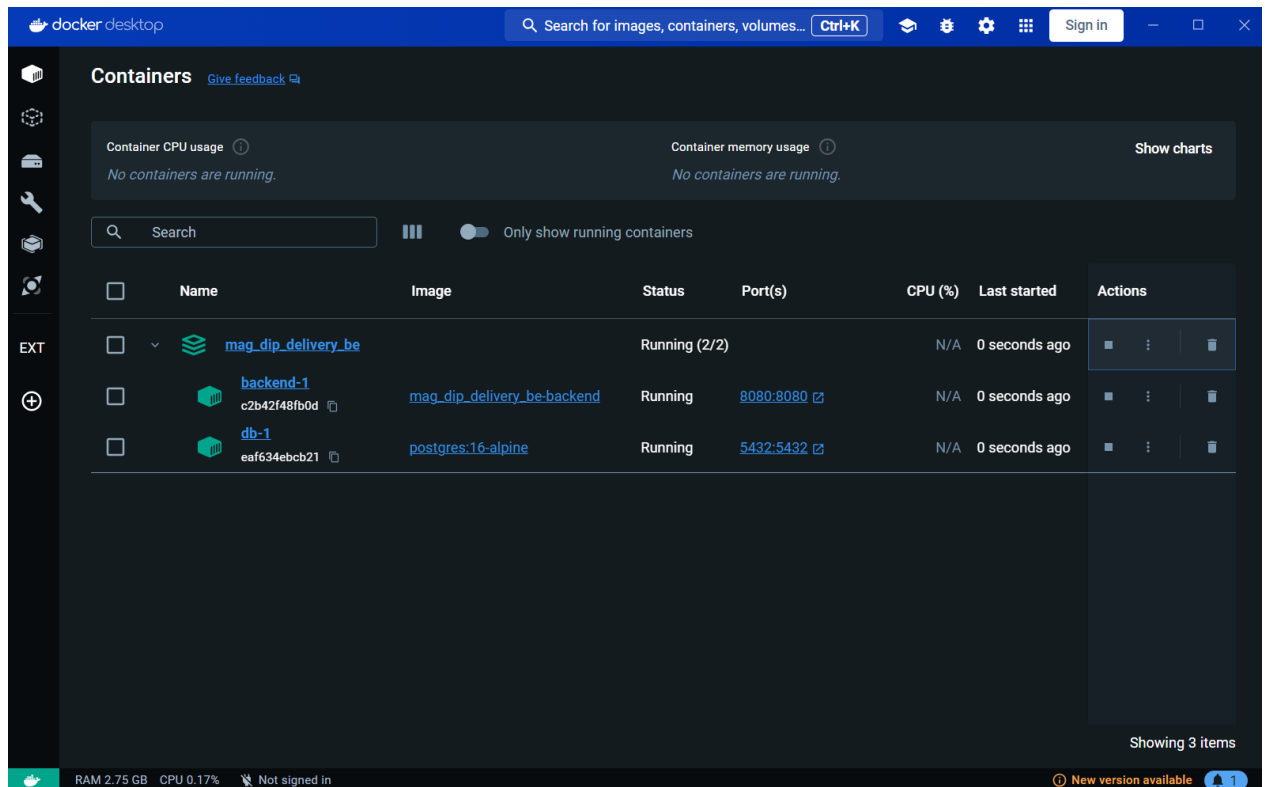


Рисунок 4.5 – Інтерфейс Docker з контейнерами проєкту

docker-compose.yml, як файл конфігурації для Docker Compose - інструменту для визначення та запуску мультиконтейнерних застосунків та Dockerfile, як файл інструкцій для побудови Docker-образу дозволяють оркеструвати кілька сервісів. Вони створюють образ з бази даних postgres:16-alpine - легкій версії PostgreSQL 16 та визначають параметри доступу і порт "5432:5432". Сервіс backend контейнера базується на офіційному образі node:20-alpine - легкій версії Node.js 20 на базі Alpine Linux, яка мінімізує розмір іміджу (близько 100-200 МБ) за рахунок відсутності непотрібних пакетів. command: yarn dev перевизначає CMD для запуску yarn dev, volumes:

- ./mag_dip_delivery_be монтує поточну директорію хосту в /mag_dip_delivery_be контейнера для hot reload, де зміни на хості відображаються в контейнері без перебудови та відкриває доступ з порту "8080:8080".

5 ПОРІВННЯ PWA З ТРАДИЦІЙНИМИ РІШЕННЯМИ

Вибір технології для розробки застосунків - це величезний спектр рішень, від чисто веб-орієнтованих до глибоко інтегрованих у платформу. У центрі цього спектру знаходяться кілька ключових підходів.

Нативні застосунки розробляються окремо для кожної платформи (наприклад, Swift/Objective-C для iOS, Kotlin/Java для Android, C# для Windows). Вони забезпечують максимальну продуктивність і повний доступ до апаратних ресурсів пристрою (камера, GPS, push-повідомлення тощо).

Кросс-платформенні рішення: такі фреймворки, як React Native, Flutter або Xamarin, дозволяють писати код один раз і компілювати його для кількох платформ. Це компроміс між продуктивністю та ефективністю розробки.

Progressive Web Apps (PWA) – це вебзастосунки, які поведуться як нативні, завдяки технологіям на кшталт Service Workers, Web App Manifest і HTTPS. Вони працюють у браузері, але можуть бути "встановлені" на пристрій, працювати оффлайн і інтегруватися з деякими функціями пристрою.

5.1 Проблема нативного підходу

З точки зору необхідності створення окремого застосунка для кожної операційної системи (наприклад, iOS, Android, Windows), нативний підхід створює значні виклики.

Кожна платформа має власну екосистему: різні мови програмування, інструменти розробки (Xcode для iOS, Android Studio для Android), та процеси публікації (App Store vs. Google Play), а це вимагає наймати окремі команди розробників. Наприклад, iOS-розробники спеціалізуються на Swift і екосистемі Apple, тоді як Android-розробники - на Kotlin і Google APIs. Це збільшує витрати: зарплати, навчання, інструменти. Якщо проєкт охоплює 3-4 платформи, може знадобитися 2-3 окремі команди, що ускладнює

координацію. Також таку кодову базу не вийде легко перевикористовувати між платформами, тому функції пишуться з нуля для кожної ОС, що значно подовжує час розробки.

Крім того є ризик фрагментації аудиторії у випадку не равноцінного фокусу. Наприклад, якщо фокус на Android (70% глобального ринку), але ігнорується iOS (30%), втрачаються преміум-користувачі а, повна підтримка всіх платформ вимагає величезних ресурсів, що не вигідно для стартапів чи малого бізнесу.

5.2 Проблеми кросс-платформених рішень

Кросс-платформені рішення намагаються вирішити проблему множини платформ, дозволяючи писати код один раз і розгортати його на кількох ОС. Однак, з точки зору необхідності застосунка для кожної системи, вони мають власні обмеження.

Фреймворки як React Native чи Flutter компілюють код у нативні компоненти, але не завжди досягають 100% нативної швидкості. Наприклад, складні анімації чи обробка великих даних можуть гальмувати на слабких пристроях. Глибока інтеграція з апаратним забезпеченням (наприклад, біометрія чи сенсори) вимагає додаткових "мостів", що ускладнює код і може призводити до помилок. Якщо застосунок потрібен для кожної системи з повною функціональністю, кросс-платформений підхід може не впоратися з платформоспецифічними функціями, змушуючи писати нативні модулі окремо.

Також розробники залежать від оновлень фреймворку. Якщо ОС змінює API (як iOS 17 з новими правилами конфіденційності), фреймворк може відставати, що затримує релізи. Це створює "необхідність" постійного моніторингу, бо один код бази повинен адаптуватися до всіх систем, але на практиці часто потрібні платформоспецифічні налаштування.

Крім того нема ідеального рішення яке б дозволило мати виключно одну кодову базу, якщо є необхідність в одразу в мобільному рішенні,

рішенні для комп'ютерів/ноутбуків та веб застосунку. В такому випадку доведеться все одно розробляти декілька окремих додаків, хоч і менше ніж у повністю нативному випадку.

5.3 PWA, як кросс-платформене рішення

PWA є еволюцією веб-підходу у світ, де бізнес хоче єдине рішення для всіх платформ одночасно. Основна перевага в тому, що замість окремих кодових баз на Swift/Kotlin/Flutter, створюється один продукт на стандартних web-технологіях (HTML, CSS, JavaScript), який працює у браузері, але поводить себе як нативний застосунок. Це означає, що не потрібні спеціалізовані команди під кожен ОС, достатньо однієї веб-команди, що дуже сильно знижує фінансові витрати і час розробки.

Service Worker дає можливість кешування і роботи оффлайн, що додає можливість реалізовувати повністю автономні застосунки з бізнес логікою, а не тільки користувацький інтерфейс. При цьому застосунок не буде працювати повільно, як працював би звичайний сайт, за рахунок кешування він буде завантажуватись тільки один раз і навіть у вході в веб версію не буде необхідності у повторному завантаженні.

Ще одна ключова перевага - не потрібні App Store або Google Play для релізу. PWA оновлюється усюди миттєво, як звичайний сайт - достатньо викласти нову версію на сервер, і користувачі отримають оновлену версію автоматично, без черг маркетплейсів та без необхідності у участі самого користувача, що може бути дуже корисно для застосунків, які часто змінюються.

Сучасний браузер сьогодні фактично став платформою, яка надає доступ до багатьох апаратних функцій - push-повідомлення, геолокація, доступ до камери і мікрофона, Bluetooth API, NFC, Web Share API, background-sync тощо. Тобто PWA може отримати більшість необхідних бізнесу апаратних можливостей, що дозволяє розробляти досить складний функціонал без потреби використовувати різні мови під різні платформи.

Фактично PWA часто дозволяє використати одну кодову базу на всі платформи де підтримуються сучасні браузер: Android, iOS, Windows, macOS, Linux. При цьому розробник отримує уніфіковану систему розгортання, значно меншу вартість підтримки, вищу швидкість оновлень і можливість додати користувачеві досвід майже на рівні нативних застосунків, навіть веб версія застосунку стає більш швидкою і отримує доступ до більшого функціоналу ніж звичайний сайт.

5.4 Проблеми PWA

PWA стикається з низкою технічних обмежень і незручностей, які часто виникають на різних пристроях та браузерах. Насамперед, це пов'язано з обмеженням локального сховища. Наприклад, на iOS/Safari кеш PWA урізається до приблизно 150 МБ, а якщо застосунок не використовувався протягом кількох тижнів, то дані можуть бути автоматично видалені браузером. За деякими оцінками, навіть IndexedDB може поводитися нестабільно на iOS, бо платформа довільно очищає кеш та бази даних після тижня бездіяльності застосунку [26]. На Android ліміти сховища менш суворі, але також існують обмеження квот часто залежні від вільного місця на диску та агресивні політики очищення старих даних у WebView. Для багатьох застосунків це означає, що можливості оффлайн-роботи PWA обмежені - при спробі кешувати великі ресурси можна швидко вичерпати квоту. Таким чином, можливості зберігати дані на пристрої обмежені і непередбачувані, що ускладнює реалізацію дуже великих застосунків, чи застосунків, які потребують постійної роботи зі сховищем у PWA.

Окремою групою проблем є обмеження доступу до апаратних можливостей. PWA працюють на базі браузера, тому не можуть повноцінно використовувати всі інтерфейси пристрою. Зокрема, на iOS програми на базі веба не мають доступу до біометрії (Face ID/Touch ID), до NFC, до Bluetooth, Siri і інших системних сервісів. Датчики руху і освітлення можуть працювати дуже обмежено, а операції з USB-пристроями або принтером взагалі

недоступні. У Android з Chrome доступ до камери, геолокації чи базових сенсорів є, але і там наприклад Bluetooth і файлову систему доводиться реалізовувати через нові Web API (Web Bluetooth, File System API), які ще підтримуються не завжди. Через ці обмеження PWA не можуть реалізувати деякі складні функції, властиві нативним застосункам. Як результат, галузі з високими апаратними вимогами - геймдев, доповнена реальність, складні системи моніторингу доволі рідко реалізуються через PWA.

Також є проблема підтримки різними браузерами. Різні виробники браузерів не в повній мірі впровадили однаковий набір API для PWA. Наприклад, інтернет-браузер на iOS завжди використовує движок WebKit, і тільки Safari на iOS підтримує додавання на головний екран і push-повідомлення (лише з iOS 16.4). Інші браузери на iPhone, як Chrome чи Firefox, технічно обмежені вимогами Apple і часто не можуть робити нічого, окрім простого відображення сайту без додаткових сервіс-воркерів чи фонових завдань. На Android та десктопі, навпаки, Chrome, Edge чи Firefox надають ширші можливості - автозапит на встановлення PWA, фонові синхронізації, більші квоти зберігання і т. д. Така фрагментація означає, що досвід користувача PWA може суттєво відрізнитися залежно від моделі пристрою, операційної системи і версії браузера.

Ще одна велика складність - це просування та встановлення. На відміну від нативних застосунків, PWA зазвичай не розміщуються централізовано в маркетах (особливо це відчутно на iOS та десктопах). Наприклад, Apple не дозволяє публікувати вебзастосунок в App Store без обгортки. Користувачі не звикли шукати програми у браузері чи в покрокових інструкціях, а процес "Додати на головний екран" у Safari є багатоступеневим і інтуїтивно незрозумілим для багатьох. Крім того, виникають питання довіри і виявлення, бо користувач частіше шукає застосунки у відомих App Store, а не через браузер і для нього це виглядає незвично і не інтуїтивно. Розміщення в Google Play та Microsoft Store можливе через Trusted Web Activity, це вимагає додаткових дій від розробника і не дає рівної функціональності з нативними програмами [27].

Варто також згадати що, PWA більше споживають ресурси пристрою і заряд батареї: через те, що браузерні технології менш енергоефективні за нативні, інтенсивні анімації чи постійне з'єднання з мережею можуть швидше розрядити акумулятор.

Таким чином, хоча PWA надають зручний кросс-платформений спосіб доставки контенту, вони мають низку помітних обмежень: від обмеженого сховища, відсутності деяких апаратних можливостей і ускладненої поширення.

5.5 Коли використовувати PWA

Прогресивні вебзастосунки найчастіше виправдовують себе тоді, коли бізнес прагне швидко вийти на ринок з одним продуктом, який працюватиме на багатьох пристроях одночасно. Для стартапу, що має обмежені ресурси, PWA дозволяє зосередити інженерні зусилля в одній веб команді, скоротити витрати на розробку і підтримку кількох нативних версій і зменшити час до першого релізу. Якщо мета - перевірити гіпотези, оцінити ринковий попит і швидко ітеративно змінювати бізнес-логіку та інтерфейс без марних черг у маркетах, PWA - ідеальних вибір. Особливо це стосується продуктів з високою залежністю від трафіку з вебу і SEO, де пошукова видимість і миттєвий доступ через посилання важливіше за присутність у крамницях застосунків.

Якщо бізнес потребує постійних непомітних оновлень для користувача, PWA дає вирішальну перевагу, бо оновлення розгортуються на сервері миттєво, контроль версій і відкат прості, а користувачам не потрібно вручну завантажувати їх. Це робить PWA привабливими для продуктів з частими змінами контенту, A/B-тестуванням інтерфейсів, адаптивними продажами або сервісів із регулярно мінливими правилами та пропозиціями. У таких випадках важливо, щоб оновлення були невидимими для користувача і щоб бізнес міг швидко реагувати на аналітику та поведінку аудиторії.

Коли у компанії вже є веб-сайт, трансформація частини функціоналу у прогресивний вебзастосунок - це логічний шлях. PWA гарно підходять для читання контенту, каталогів, замовлення квитків, бронювання, інформаційних сервісів і процесів, де користувачі часто заходять з браузера. Якщо сайт має стабільну аудиторію, додавання можливості «встановити» сайт як застосунок, робота оффлайн для ключових сторінок і пуш-повідомлення для повернення користувача можуть значно підвищити утримання без великих додаткових інвестицій. Тому для бізнесів з уже наявним веб-продуктом PWA часто стає першою еволюцією до мобільного досвіду.

Водночас архітектурний вибір має базуватися на конкретних бізнес-функціях. Якщо застосунок вимагає глибокої інтеграції з апаратурою пристрою - доступу до біометрії для підпису транзакцій, широкого використання NFC для платежів чи зчитування карт, низькорівневої роботи з USB або високопродуктивної графіки для ігор або доповненої реальності - PWA, ймовірно, не підійде. Подібні функції або взагалі недоступні, або їх підтримка є урізаною і непослідовною в різних браузерах. Коли безпека вимагає апаратного захисту ключів або сертифікованих SDK платіжних провайдерів у складі нативного середовища, варто обирати нативні рішення.

Також цільова аудиторія грає ключову роль у виборі. Якщо користувачі вашого продукту вміють та звикли працювати у браузері і вам важливий миттєвий доступ через посилання, PWA буде позитивним рішенням. Проте якщо аудиторія - масовий ринок, звиклий шукати і завантажувати застосунки через App Store, тоді відсутність застосунку у маркеті може знизити довіру. Також варто врахувати регіональні відмінності - у країнах зі значною перевагою Android PWA можуть кращим підходом, тоді як у регіонах з переважанням iPhone питання обмеженої підтримки деяких API на iOS заслуговують на додаткову перевірку.

Інша важлива ознака - характер користувацької взаємодії. Якщо сесії короткі, часті та орієнтовані на читання або легкі транзакції, PWA підходить відмінно. Якщо ж взаємодія передбачає довгі сеанси з інтенсивною

обробкою, реальним часом або високими вимогами до графіки і мультимедіа, переваги нативних рішень зростають.

Інколи оптимальним рішенням стає гібридний підхід. Можна почати з прогресивного вебзастосунку для швидкої перевірки продукту і покриття веб-аудиторії, а згодом, якщо певні користувацькі функції або монетизація вимагають нативних можливостей, розробити обгортку або нативні модулі. Такий шлях дозволяє зменшити ризики інвестицій, одночасно зберігаючи можливість масштабування.

У підсумку, рішення використовувати PWA має бути результатом зіставлення технічних обмежень, бізнес-цілей і поведінки цільової аудиторії. Для стартапів і продуктів, що орієнтовані на швидке тестування і часті оновлення, для сервісів зі значною частиною веб-трафіку і для внутрішніх інструментів PWA часто є оптимальним вибором. Для продуктів із високими апаратними вимогами, потребою в апаратному захисті, або для випадків, коли маркет-експозиція в крамницях є вирішальною, слід розглядати нативні або гібридні шляхи.

5.5 Висновки до розділу

У цьому розділі було проведено комплексне порівняння трьох основних підходів до створення сучасних цифрових продуктів: нативних мобільних застосунків, кросс-платформених та PWA. Аналіз показав, що кожен із методів має власні сильні та слабкі сторони, які по-різному впливають на швидкість розробки, вартість підтримки, технічні можливості та якість користувацького досвіду.

Нативні мобільні застосунки залишаються найбільш потужним рішенням у випадках, коли потрібна максимальна продуктивність, розширені можливості роботи з апаратним забезпеченням, а також повна інтеграція з екосистемою платформи. Водночас вони є найбільш дорогими й ресурсозатратними через необхідність підтримувати окремі кодові бази та команди під кожную операційну систему.

Кросс-платформенні підходи частково вирішують проблему дублювання розробки, однак вони не позбавляють від обмежень, пов'язаних із продуктивністю, складністю підтримки платформоспецифічних функцій та залежністю від зовнішніх фреймворків. Порівняння наведено в таблиці 5.1.

Таблиця 5.1 – Порівняння PWA, як кросс-платформенного рішення

	PWA	Кросс-платформенні фреймворки	Нативні застосунки
Кількість кодових баз та команд	Одна команда, одна кодова база	Переважно одна кодова база, але зі змішаним нативним кодом	Окремі команди та кодові бази
Вартість розробки та підтримки	Найнижча	Середня	Найвища
Час розробки	Найшвидший	Швидкий	Найповільніший
Продуктивність	Висока, але нижча за нативну; залежить від браузера	Вища за PWA, наближена до нативної	Найвища
Доступ до апаратних API	Обмежений (особливо на iOS)	Android, iOS (+ інколи desktop)	Повний
Підтримка платформ	Будь яка платформа де можна встановити браузер	Android, iOS (+ інколи desktop)	Лише конкретна ОС
Оновлення та релізи	Миттєві без участі користувача	Через магазини (швидше, ніж нативні)	Через магазини (повільно)
Відповідність складним вимогам (AR, сенсори, біометрія)	Погана (відсутні або урізані API)	Середня (часто потребує нативних модулів)	Відмінна

PWA, натомість, демонструють оптимальний баланс між універсальністю вебу та можливостями нативних застосунків. Вони

забезпечують єдину кодову базу, роботу оффлайн, швидке розгортання без маркетплейсів, миттєві оновлення та значну економію ресурсів. Проте цей підхід також має суттєві обмеження: нестабільне локальне сховище (особливо на iOS), неповний доступ до апаратних API, складності з інсталяцією на деяких платформах та різну підтримку функцій у браузерях.

Отже, результати дослідження демонструють, що PWA є технологією, яка займає проміжне, але стратегічно вигідне положення між традиційним вебom та нативними застосунками. Вони забезпечують високий рівень мобільності, гнучкість розробки, можливість оффлайн-роботи та швидку доставку оновлень, що робить їх оптимальним вибором для великої кількості сучасних бізнес-сценаріїв. Водночас у сферах, де критичними є апаратні можливості, продуктивність на рівні системи або потреба у маркет-присутності, нативні рішення залишаються пріоритетними.

Таким чином, PWA можна розглядати як перспективний підхід для проєктів, спрямованих на широку аудиторію, швидку розробку та підтримку, а також мінімізацію витрат - що й обґрунтовує їх застосування в межах даного дипломного проєкту.

6 РОБОТА КОРИСТУВАЧА З РОЗРОБЛЕНИМ ЗАСТОСУНКОМ

6.1 Вимоги до користувача

Для ефективної роботи з розробленим прогресивним вебзастосунком користувач повинен відповідати низці технічних, організаційних та програмних вимог. Ці вимоги визначають необхідні умови, за яких PWA може забезпечити повну функціональність, у тому числі оффлайн-режим, локальне зберігання даних, обробку інформації без доступу до Інтернету та подальшу синхронізацію із сервером після відновлення з'єднання.

Вимоги поділяються на декілька груп: технічні, програмні, мережеві, апаратні, організаційні та поведінкові.

6.1.1 Технічні вимоги до пристрою

Для запуску та стабільної роботи розробленого вебзастосунку користувач повинен володіти пристроєм, що відповідає таким критеріям:

Пристрій має бути сумісним із браузерами, які реалізують необхідні API: Service Worker, Cache Storage API, IndexedDB, Fetch API, Local Storage, ES6+ JavaScript.

Через використання локальної бази даних IndexedDB, кешу та інших механізмів зберігання, пристрій повинен мати доступний вільний обсяг пам'яті. Браузери можуть накладати власні обмеження, залежно від платформи (Android, iOS, Windows, macOS).

Сумісність з операційними системами:

- Android 8.0 або новіше;
- iOS 13+ (з підтримкою PWA у Safari);
- Windows 10/11;
- macOS Catalina або новіше;
- Linux-дистрибутиви з підтримкою актуальних браузерів.

6.1.2 Вимоги до дозволів браузера та системи

Для повноцінної взаємодії із застосунком користувач повинен надати низку дозволів, якщо вони необхідні в межах реалізованого функціоналу:

Дозвіл на доступ до камери - бо застосунок передбачає завантаження фотографій, сканування документів або інші подібні функції.

Дозвіл на доступ до геолокації - функціонал застосунку передбачає визначення координат користувача.

Дозвіл на push-сповіщення - за потреби отримання нагадувань, оновлень синхронізації або системних повідомлень.

Дозвіл на збереження даних у локальному сховищі (включно з кешем, IndexedDB і LocalStorage).

У разі відмови від надання дозволів частина функціональних можливостей може бути обмежена.

6.1.3 Вимоги до користувацьких навичок та взаємодії

Користувач має володіти базовими навичками роботи з веббраузером і мобільним/настільним пристроєм.

Уміння здійснювати базові операції у вебінтерфейсі.

Розуміння того, що частина операцій може бути виконана не миттєво. У разі роботи без Інтернету деякі зміни відправляються до сервера лише після відновлення зв'язку. Користувач повинен розуміти це, для уникнення помилкових дій та розуміння стану змін, які він створює.

6.2 Інструкція для користувача

У цьому розділі надається детальна інструкція для користувача щодо роботи з розробленим прогресивним вебзастосунком, призначеним для управління доставками. Інструкція охоплює ключові сценарії використання, включаючи аутентифікацію, створення доставок та додавання звітів, з

акцентом на роботу в онлайн- та оффлайн-режимах. Також описано інтерфейс користувача з ілюстраціями екранів, що допомагає зрозуміти послідовність дій та стани системи.

6.2.1 Сценарії роботи користувача

Робота з розробленим застосунком починається з відкриття у браузері на мобільному або іншому пристрої з підтримкою необхідних технологій та переходить за вказаною адресою, де розгорнуто систему (рис. 6.1). Після першого завантаження застосунок автоматично ініціалізує всі необхідні модулі, включаючи сервіс-воркер, локальну базу даних і систему кешування, завдяки чому користувач отримує можливість працювати як у звичайному режимі онлайн, так і повністю автономно без доступу до мережі.

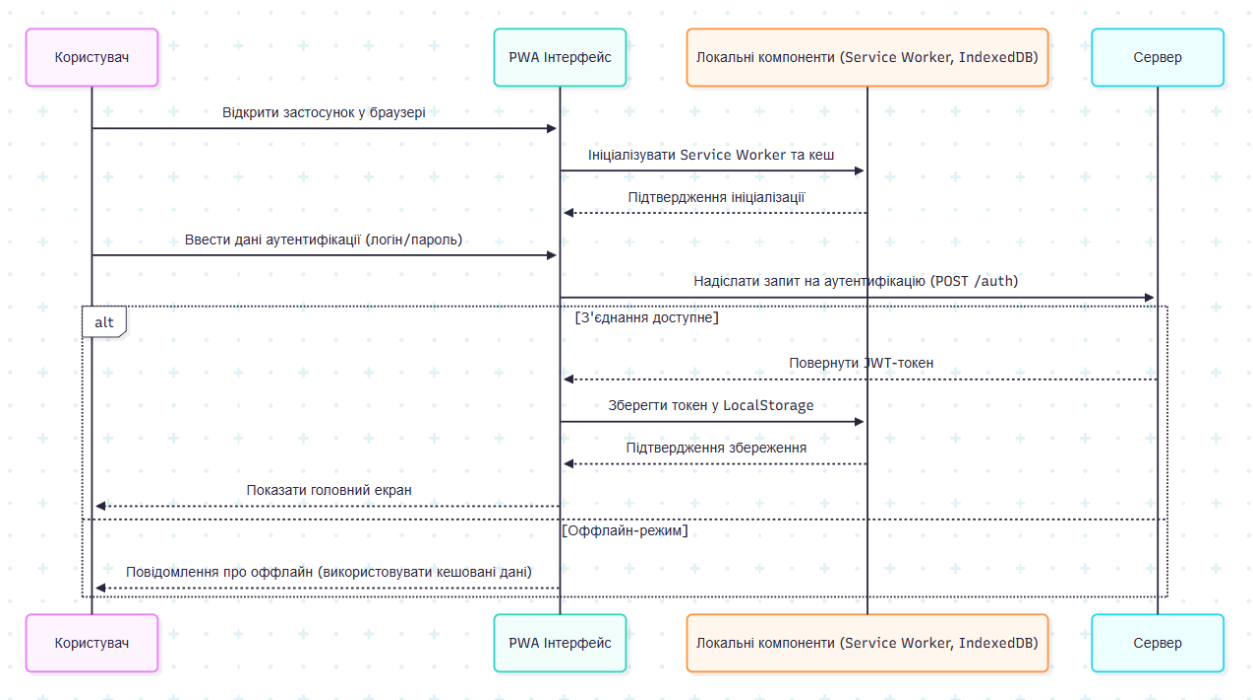


Рисунок 6.1 – Сценарій входу користувача

Під час аутентифікації: користувач відкриває застосунок у браузері, після чого PWA інтерфейс ініціює локальні компоненти, такі як Service Worker для кешування та IndexedDB для зберігання даних. Далі користувач вводить свої дані для входу (логін і пароль), і PWA надсилає запит на сервер

для перевірки. Якщо з'єднання з інтернетом доступне, сервер повертає JWT-токен, який зберігається локально в LocalStorage для подальших авторизованих дій, і користувач переходить на головний екран. У випадку відсутності інтернету, система переходить в оффлайн-режим, показуючи відповідне повідомлення та дозволяючи працювати з кешованими даними, де можливо. Цей сценарій підкреслює гнучкість PWA в умовах нестабільного з'єднання, забезпечуючи безперервність роботи за рахунок локального зберігання та синхронізації.

Користувач сканує код або вводить номер вручну, після чого дані обробляються локально: створюється запис доставки в статусі Draft і зберігається в IndexedDB. PWA показує користувачу сторінку доставки з цим статусом, дозволяючи переглянути деталі (рис. 6.2).

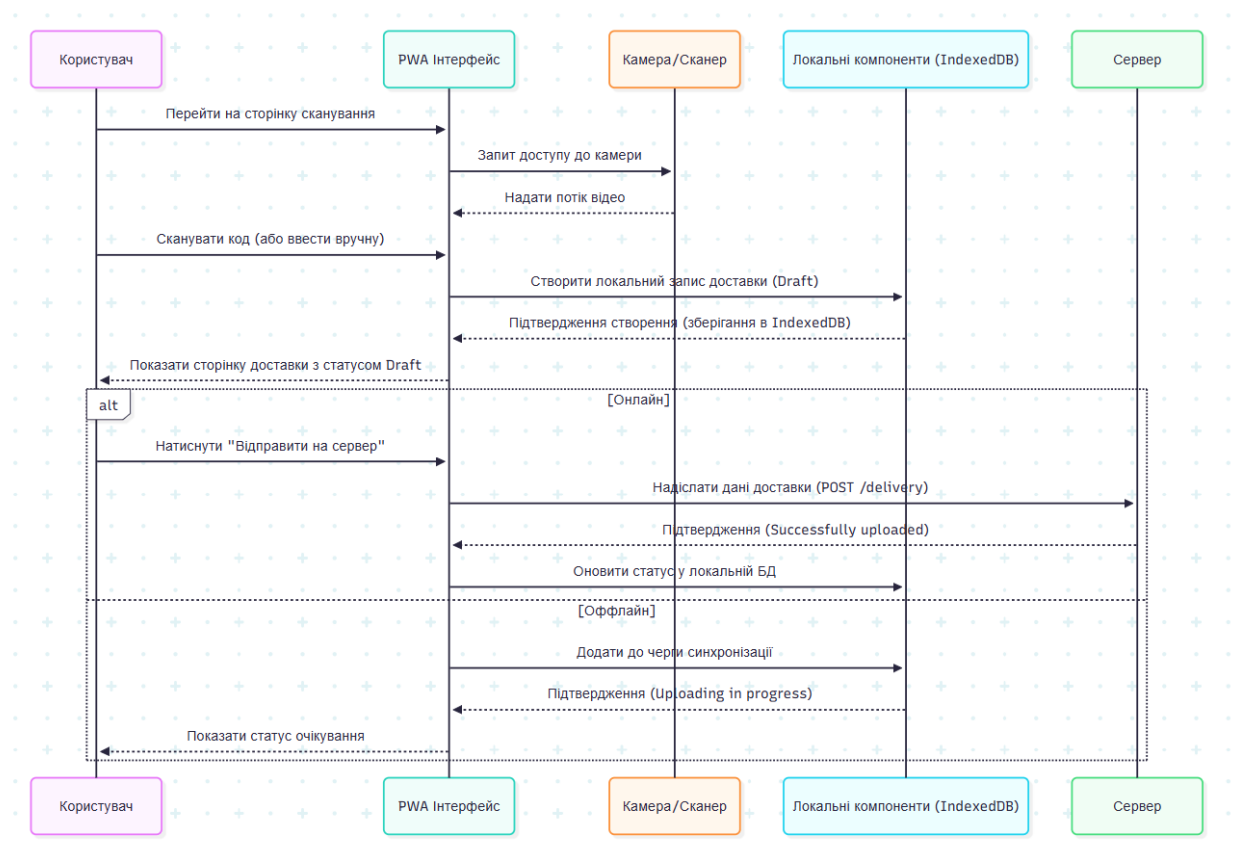


Рисунок 6.2 – Сценарій нового сканування

Якщо інтернет доступний, користувач може натиснути кнопку "Відправити на сервер", що ініціює POST-запит для завантаження даних; сервер підтверджує успіх, і статус оновлюється локально. У оффлайн-режимі

дані додаються до черги синхронізації з статусом *Uploading in progress*, а користувач бачить індикатор очікування. Цей сценарій демонструє, як PWA забезпечує автономну роботу, дозволяючи створювати записи без інтернету з подальшою автоматичною синхронізацією, що є критичним для мобільних користувачів у польових умовах.

До існуючої доставки в статусі *Draft*: користувач переходить на сторінку доставки, де додає звіт з коментарями та фото, для чого PWA запитує доступ до камери або галереї (рис. 6.3).

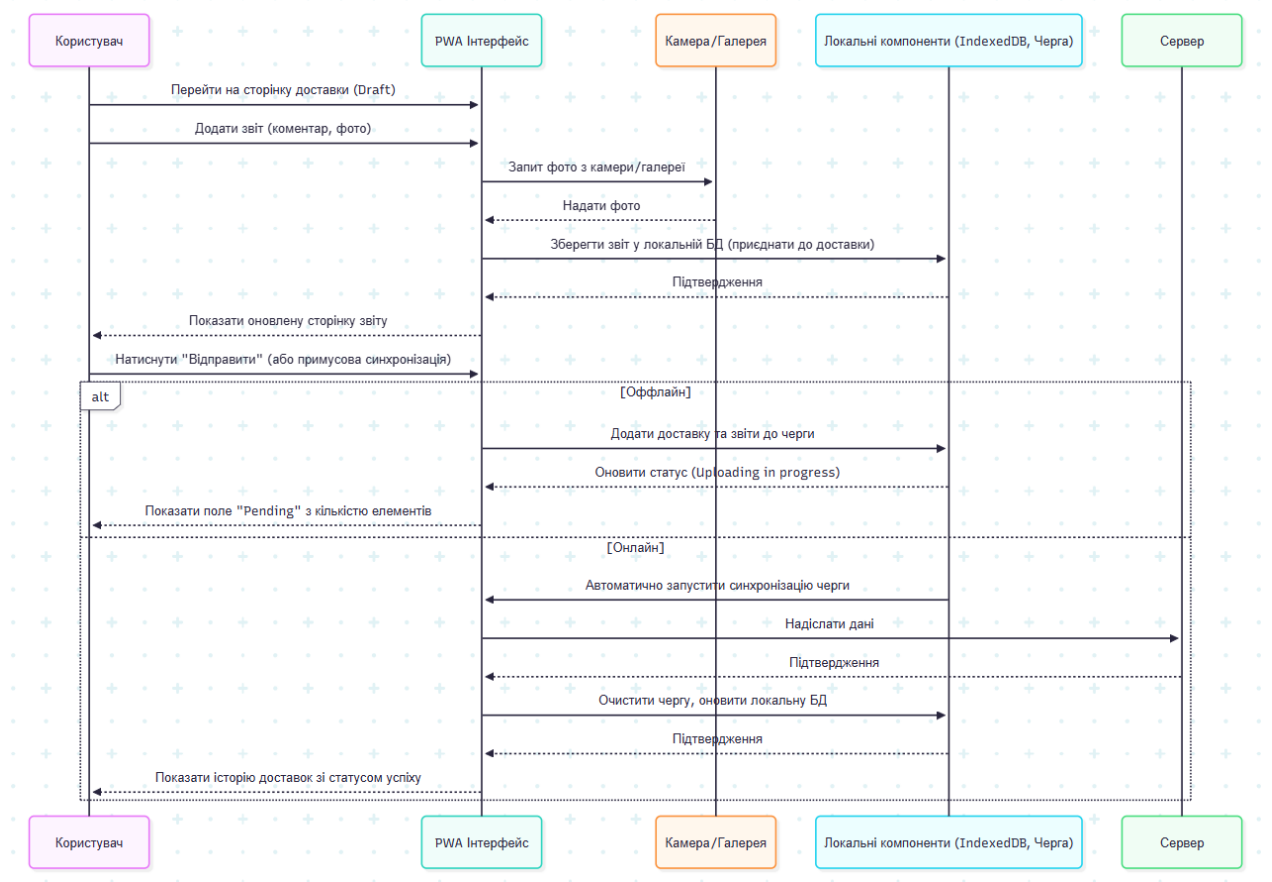


Рисунок 6.3 – Сценарій додавання звіту

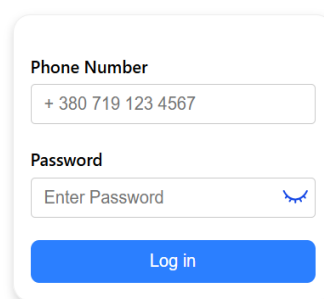
Отримані дані зберігаються локально в IndexedDB, приєднуючись до запису доставки, і PWA оновлює інтерфейс для показу змін. Користувач може ініціювати відправку натисканням кнопки "Відправити" або через примусову синхронізацію. В оффлайн-режимі доставка та звіти додаються до черги з оновленням статусу *Uploading in progress*, а користувач бачить поле "Pending" з кількістю елементів у черзі. При відновленні з'єднання система

автоматично запускає синхронізацію: дані надсилаються на сервер через POST-запити для доставки та звіту, сервер підтверджує успіх, черга очищається, локальна база даних оновлюється, і користувач переходить до історії доставок зі статусом Successfully uploaded. Цей сценарій підкреслює механізм черги синхронізації, який забезпечує надійність даних у нестабільних мережах, дозволяючи користувачеві продовжувати роботу без втрат інформації та автоматично завершувати процес при відновленні з'єднання.

Робота з розробленим застосунком починається з відкриття у браузері на мобільному або іншому пристрої з підтримкою необхідних технологій та переходить за вказаною адресою, де розгорнуто систему. Після першого завантаження застосунок автоматично ініціалізує всі необхідні модулі, включаючи сервіс-воркер, локальну базу даних і систему кешування, завдяки чому користувач отримує можливість працювати як у звичайному режимі онлайн, так і повністю автономно без доступу до мережі.

6.2.2 Інтерфейс користувача

Користувач починає роботу з форми аутентифікації (рис.6.4). Він вводить дані під якими зареєстрований в системі, після чого система ідентифікує його і повертає JWT токен для подальшої взаємодії.



Phone Number

+ 380 719 123 4567

Password

Enter Password

Log in

Рисунок 6.4 – Форма аутентифікації користувача

Головний екран (рис. 6.5) пропонує основні функції, такі як: сканування, перегляд історії доставок, та список звітів. У навігаційній панелі

знаходяться: іконка застосунку, іконка, яка показує стан мережі, швидкий вхід у сканування та меню.

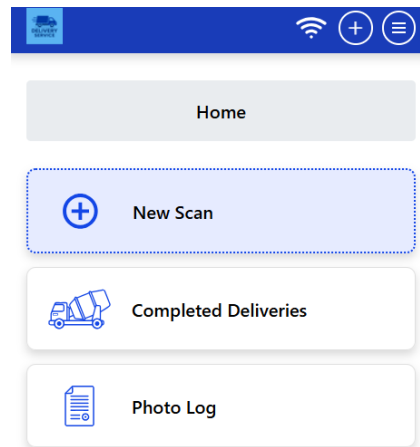


Рисунок 6.5 – Головний екран

На сторінці сканування (рис. 6.6) користувач бачить область для сканування, можливість вводу номеру вручну та режим перемикавання камери. Також поточна камера відображається у лівому нижньому куті. Іконка фотографії, робить фото та сканує код.



Рисунок 6.6 – Сторінка сканування

Після створення доставки створюється новий запис і користувач потрапляє на сторінку доставки (рис. 6.7). Тут він має доступ до основної інформації про доставку, фотографії та стану завантаження. В даному

випадку Draft говорить про те, що доставка коректно створена, але наразі існує тільки локально і потребує відправки на сервер.

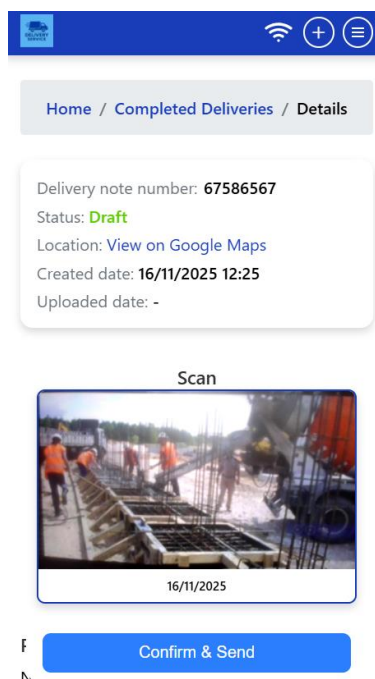


Рисунок 6.7 – Сторінка доставки

Нижче на цій сторінці (рис. 6.8) користувач може додати звіт у разі якихось проблем та передивитись існуючи. Також кнопка для відправки на сервер плаває і доступна на всій сторінці.

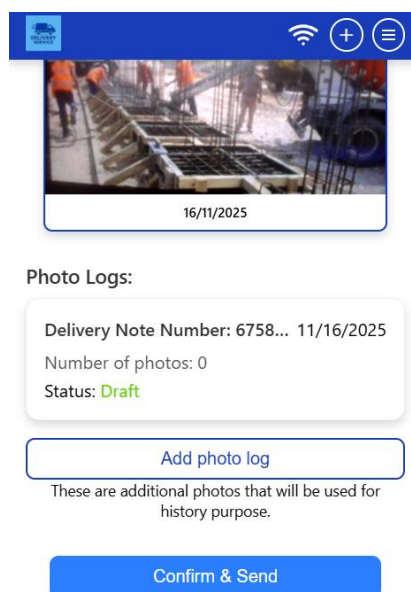


Рисунок 6.8 – Сторінка доставки, секція звітів

На сторінці звіту (рис. 6.9) користувач також бачить основну інформацію про цю сутність. Має змогу завантажити фото з галереї або зробити через камеру. Також може додати коментар. Якщо доставка підтвержена то він може відправити звіт, якщо ні то звіт створиться і буде очікувати відправки доставки.

DELIVERY SERVICE

Home / Photo Log / Details

Delivery note number: 67586567

Status: **Draft**

Number of photos: 0

Created date: 16/11/2025 12:26

Uploaded date: -

+ ↓

Товару в машині менше ніж зазначено у кошторисі.

Confirmation and send is not available until delivery is confirmed

Рисунок 6.9 – Сторінка звіту

Після відправлення, якщо нема інтернету статус зміниться на Uploading in progress та дость у чергу синхронізації (рис. 6.10). Як тільки з'явиться інтернет і ресурси не будуть сильно зайняті все, що є в черзі

автоматично оновиться. Також користувач має змогу примусово почати синхронізацію через меню.

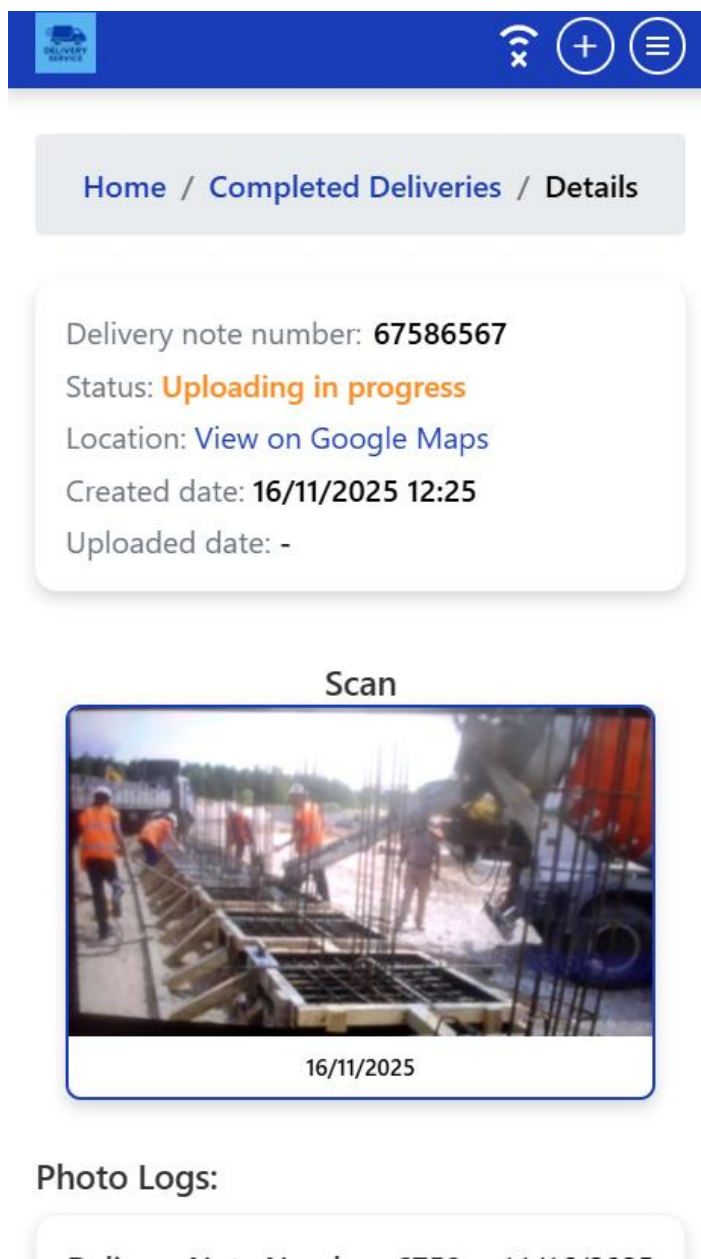


Рисунок 6.10 – Доставка у стані відправки

У користувача з'являється поле пендінг (рис. 6.11), яке показує скільки сутностей зараз у черзі синхронізації, а коли буде інтернет з'єднання кнопка синхронізації буде доступна.

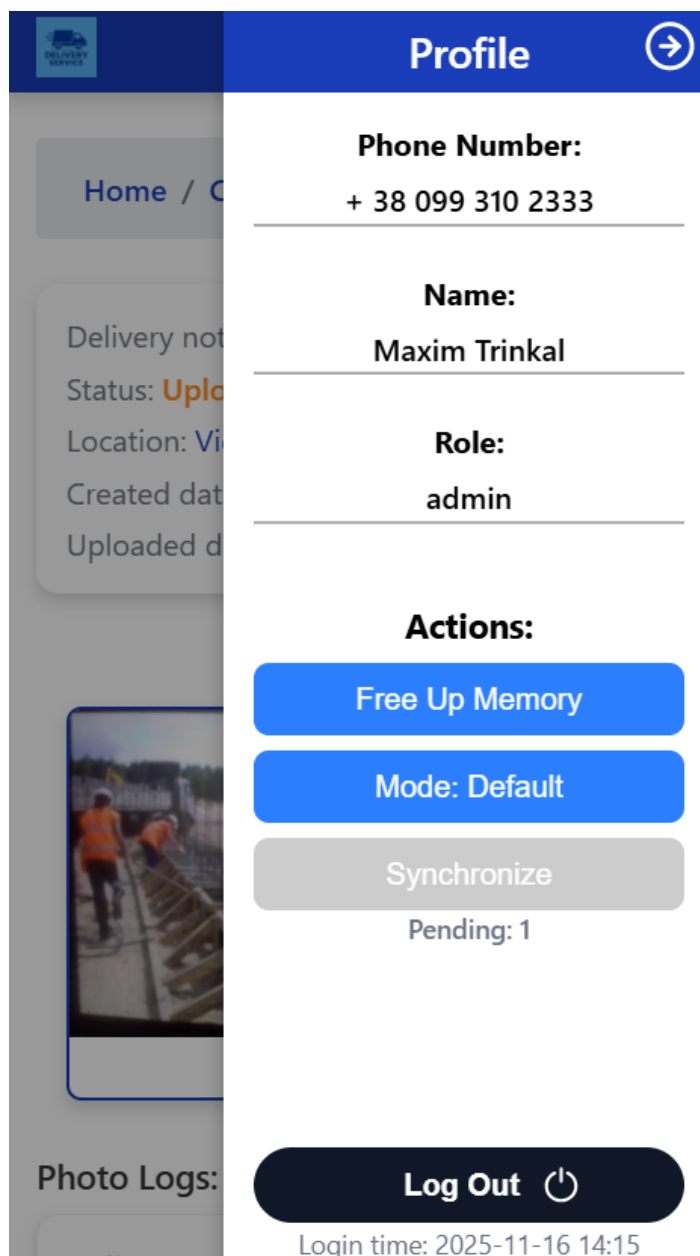


Рисунок 6.11 – Меню користувача та поле пендінг

Синхронізовані сутності відображаються з статусом **Successfully uploaded** і доступні у історії доставок (рис. 6.12). Звіти виглядають та поводять себе так само на сторінці історії звітів.

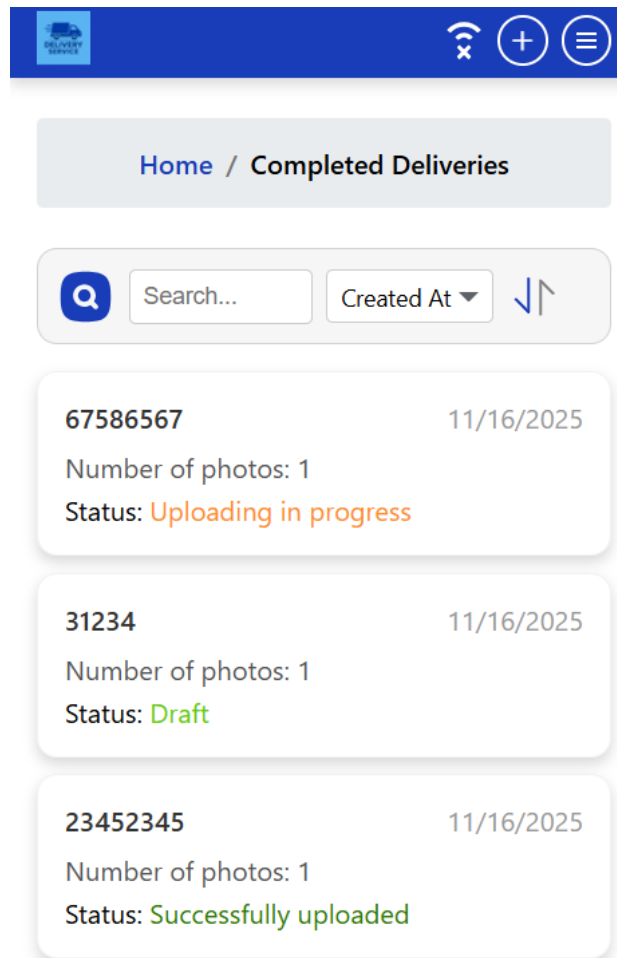


Рисунок 6.12 – Історія доставок

Таким чином досягається досить інтуїтивний інтерфейс з явними флагами та широкими можливостями, що дозволяє навіть не досвідченому користувачу орієнтуватися і користуватись застосунком.

6.3 Інструкція для завантаження PWA

Після першого завантаження застосунок автоматично ініціалізує всі необхідні модулі, включаючи сервіс-воркер, локальну базу даних і систему кешування, завдяки чому користувач отримує можливість працювати як у звичайному режимі онлайн, так і повністю автономно без доступу до мережі. Після відкриття застосунку користувач може встановити його на свій пристрій як повноцінний прогресивний вебзастосунок. У більшості сучасних браузерів кнопка встановлення з'являється безпосередньо в інтерфейсі (рис. 6.13) – у верхній частині вікна або в основному меню. Після натискання цієї

кнопки браузер пропонує підтвердити встановлення, після чого застосунок додається на головний екран смартфона або на робочий стіл комп'ютера й відкривається у вигляді окремої програми. На iOS додавання застосунку здійснюється через стандартне меню “Поділитися”, де потрібно обрати пункт “Додати на головний екран”.

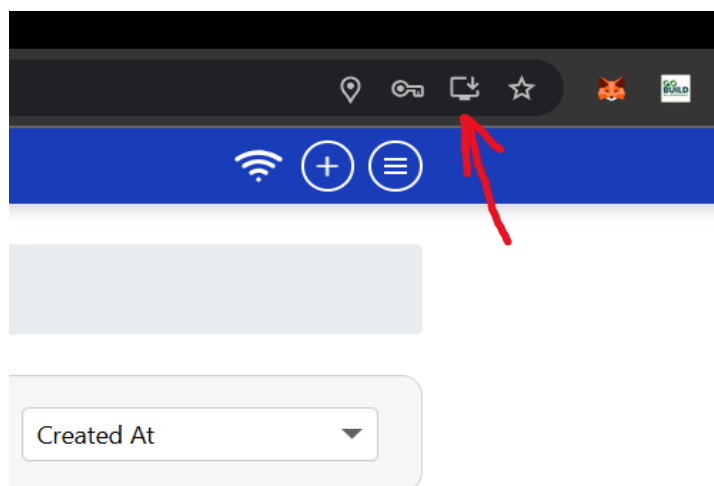


Рисунок 6.13 – Кнопка завантаження застосунку

Після встановлення застосунок працює окремо від браузера, без адресного рядка і сторонніх елементів, що створює відчуття нативної програми. А також з'являється іконка на головному екрані (рис. 6.14), яка дозволяє входити напряму у застосунок.

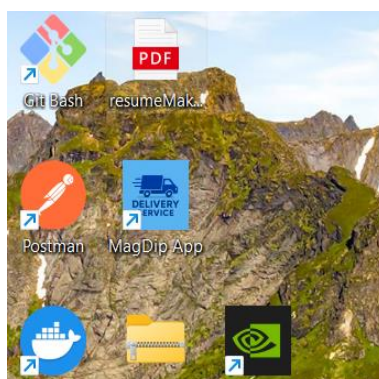


Рисунок 6.14 – Іконка застосунку на головному екрані

В завантаженому вигляді застосунок (рис. 6.15) може працювати оффлайн, не дозавантажує статичні ресурси, а також автоматично оновлюється при виході нової версії.

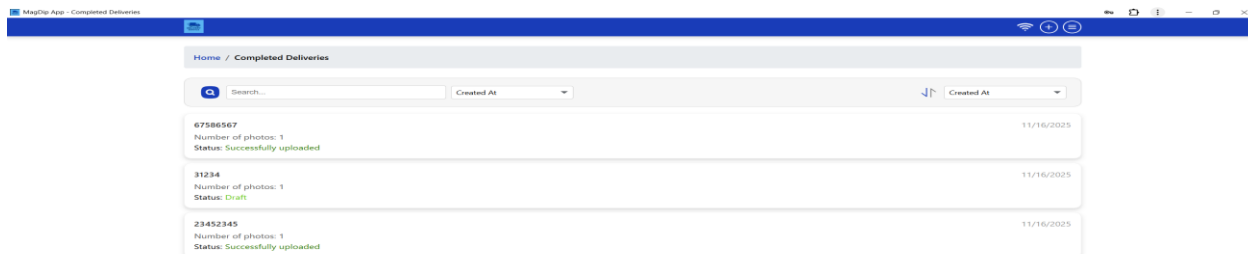


Рисунок 6.15 – Завантажений застосунок

Також при вході через браузер пропонується перейти у вже завантажену версію (рис. 6.16).

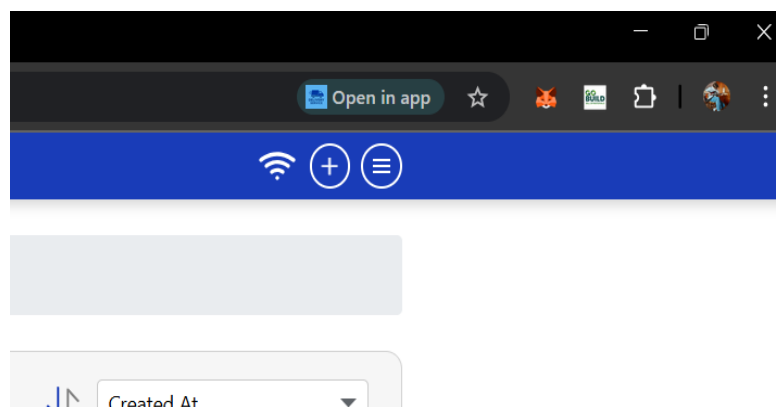


Рисунок 6.16 – Іконка входу у браузері

Такий сценарій використання створює комфортний та інтуїтивний користувацький досвід, притаманний нативним мобільним застосункам.

ВИСНОВКИ

У результаті виконання дипломної роботи було розроблено, реалізовано та досліджено прогресивний веб-застосунок (PWA), здатний працювати у повністю автономному режимі без підключення до мережі та забезпечувати узгоджену синхронізацію даних із сервером після відновлення з'єднання. Проведене дослідження підтвердило доцільність технологій PWA як сучасної альтернативи традиційним мобільним і десктопним рішенням.

Розроблена модель застосунку продемонструвала можливість реалізації стабільної оффлайн-роботи, використовуючи механізми локального сховища, сервіс-воркерів і відкладеної синхронізації. Тестування показало, що застосунок продовжує функціонувати навіть за повної втрати мережевого доступу, забезпечуючи збереження користувацьких даних і подальше їх узгодження після появи з'єднання. Це свідчить про підвищену стійкість системи до зовнішніх факторів і зменшення ризику втрати інформації.

Впровадження архітектури, побудованої на стеку React, TypeScript, Node.js, Express, Zustand і Workbox, дало змогу створити продуктивний, масштабований та гнучкий застосунок, який легко адаптується під різні платформи та пристрої без необхідності розробки окремих нативних версій. Застосування принципів MVC та MVVM забезпечило чітке розділення логіки, даних і представлення, що спрощує супровід та розширення системи.

Порівняльний аналіз із традиційними нативними рішеннями показав, що підхід PWA забезпечує менше споживання ресурсів, швидше завантаження, простоту оновлення та меншу вартість розробки при збереженні високої якості користувацького досвіду. Такі результати демонструють реальну економічну та технічну доцільність застосування PWA у бізнес-середовищах, внутрішніх корпоративних системах і масових користувацьких сервісах.

Розроблений вебзастосунок підтвердив, що технологія може стати ефективною основою для створення сучасних застосунків, орієнтованих на автономність, стабільність і кросс-платформеність. Отримані результати

мають практичне значення для побудови систем, які повинні функціонувати в умовах нестабільного зв'язку, обмеженого трафіку або необхідності швидкого розгортання без додаткових витрат на інфраструктуру.

Таким чином, виконана робота довела перспективність підходу PWA для створення автономних, швидких і надійних вебзастосунків.

ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАННЯ

1. Twitter Lite PWA Significantly Increases Engagement and Reduces Data Usage [Electronic resource]. – Mode of access: <https://web.dev/case-studies/twitter#:~:text=%2A%2065,decrease%20in%20bounce%20rate>.
2. Google Maps Statistics: Navigation, Business Integration, and More [Electronic resource]. – Mode of access: <https://sqmagazine.co.uk/google-maps-statistics>.
3. A one year PWA retrospective [Electronic resource]. – Mode of access: <https://medium.com/pinterest-engineering/a-one-year-pwa-retrospective-f4a2f4129e05>.
4. The State of Developer Ecosystem 2023 [Electronic resource]. – Mode of access : <https://www.jetbrains.com/lp/devecosystem-2023/>.
5. ECMAScript 2026 Language Specification [Electronic resource]. – Mode of access : <https://tc39.es/ecma262/>.
6. TypeScript Documentation [Electronic resource]. – Mode of access: <https://www.typescriptlang.org/docs/>.
7. Vite Documentation [Electronic resource]. – Mode of access: <https://vite.dev/>.
8. PNPM Documentation [Electronic resource]. – Mode of access: <https://pnpm.io/>.
9. Docker Documentation [Electronic resource]. – Mode of access: <https://docs.docker.com/>.
10. HTML Living Standard [Electronic resource]. – Mode of access: <https://html.spec.whatwg.org/>.
11. Less Documentation [Electronic resource]. – Mode of access: <https://lesscss.org/>.
12. React Documentation [Electronic resource]. – Mode of access: <https://react.dev/>.
13. Zustand Documentation [Electronic resource]. – Mode of access: <https://zustand.docs.pmnd.rs/getting-started/introduction>.

14. Workbox Documentation [Electronic resource]. – Mode of access: <https://developer.chrome.com/docs/workbox>.
15. Node Documentation [Electronic resource]. – Mode of access: <https://nodejs.org/docs/latest/api/>.
16. Express Documentation [Electronic resource]. – Mode of access: <https://expressjs.com/>.
17. Measuring the (Over)use of Service Workers for In-Page Push Advertising [Electronic resource]. – Mode of access: <https://arxiv.org/abs/2110.11241>.
18. What are Progressive Web Apps? [Electronic resource]. – Mode of access: <https://web.dev/articles/what-are-pwas#:~:text=Web%20applications%20are%20more%20capable,based%20image%20compressor>.
19. Що являє собою progressive web app [Електронний ресурс]. – Mode of access: <https://foxminded.ua/progressive-web-app/>.
20. Architectural Pattern of Progressive Web and Background Sync [Electronic resource]. – Mode of access: <https://www.semanticscholar.org/paper/Architectural-Pattern-of-Progressive-Web-and-Behl-Raj/8a3a154254578e2cbde707084d22bcc6a4f53125>.
21. Using IndexedDB for Offline-First Web Applications [Electronic resource]. – Mode of access: <https://dev.to/hexshift/using-indexeddb-for-offline-first-web-applications-33o0>.
22. Implementing Offline-First Web Apps for Remote Healthcare Monitoring [Electronic resource]. – Mode of access: https://www.researchgate.net/publication/394517782_Implementing_Offline-First_Web_Apps_for_Remote_Healthcare_Monitoring.
23. Offline first with progressive web apps [Electronic resource]. – Mode of access: <https://schoovaertswout.medium.com/offline-first-with-progressive-web-apps-part-3-3-background-sync-6766cda56947>.

24. Data Synchronization in PWAs: Offline-First Strategies [Electronic resource]. – Mode of access: <https://gtcsys.com/data-synchronization-in-pwas-offline-first-strategies-and-conflict-resolution>.

25. Case Study on Efficient Data Handling and Synchronization in Offline-First PWAs [Electronic resource]. – Mode of access: <https://ijsrcseit.com/CSEIT25112782>.

26. Mozilla Documentation [Electronic resource]. – Mode of access: <https://developer.mozilla.org/en-US/>.

27. Progressive Web Apps (PWAs): Enhancing User Experience through Modern Web Development [Electronic resource]. – Mode of access: https://www.researchgate.net/publication/385260558_Progressive_Web_Apps_PWAs_Enhancing_User_Experience_through_Modern_Web_Development.

ДОДАТОК А
Текст програми

A.1 Конфігурація PWA застосунку

```

import { defineConfig, loadEnv, type UserConfig } from "vite";import react from
"@vitejs/plugin-react";
import viteCompression from "vite-plugin-compression";
import { VitePWA } from "vite-plugin-pwa";
import mkcert from "vite-plugin-mkcert";import { writeFileSync } from "fs";
import { resolve } from "path";

const TERSER_ECMA_LEVEL = 2020;

export default defineConfig(({ mode }) => {
  const env = loadEnv(mode, process.cwd());

  const buildInfoPlugin = {
    name: "build-info-plugin",
    writeBundle() {
      const buildInfo = { buildVersion: new Date().toISOString() };
      writeFileSync(
        resolve(__dirname, "dist/build-info.json"),
        JSON.stringify(buildInfo, null, 2)
      );
    },
  };

  const config: UserConfig = {
    plugins: [
      react({
        babel: { plugins: ["babel-plugin-react-compiler"] },
      }),
      viteCompression({ algorithm: "gzip" }),
      viteCompression({ algorithm: "brotliCompress", ext: ".br" }),
      mkcert(),
      VitePWA({
        registerType: "autoUpdate",
        devOptions: { enabled: true },
        manifest: {
          name: env.VITE_APP_NAME,
          short_name: env.VITE_APP_SHORT_NAME,
          description: env.VITE_APP_DESCRIPTION,
          theme_color: env.VITE_THEME_COLOR,
          background_color: env.VITE_BACKGROUND_COLOR,
          display: "standalone",
          icons: [
            { src: "/android/android-launchericon-512-512.png", sizes: "512x512",
type: "image/png" },

```

```

    { src: "/android/android-launchericon-192-192.png", sizes: "192x192",
type: "image/png" },
    { src: "/android/android-launchericon-144-144.png", sizes: "144x144",
type: "image/png" },
    { src: "/android/android-launchericon-48-48.png", sizes: "48x48", type:
"image/png" },
    { src: "/windows11/Square44x44Logo.targetsize-16.png", sizes: "16x16",
type: "image/png" },
    { src: "/windows11/Square44x44Logo.targetsize-32.png", sizes: "32x32",
type: "image/png" },
    { src: "/windows11/Square44x44Logo.targetsize-72.png", sizes: "72x72",
type: "image/png" },
    { src: "/windows11/Square44x44Logo.targetsize-96.png", sizes: "96x96",
type: "image/png" },
    { src: "/windows11/Square44x44Logo.targetsize-256.png", sizes:
"256x256", type: "image/png" },
    { src: "/ios/180.png", sizes: "180x180", type: "image/png" },
    { src: "/ios/1024.png", sizes: "1024x1024", type: "image/png" },
  ],
  screenshots: [
    { src: "/preview/afri_pwa_desk.png", sizes: "1139x625", type:
"image/png", form_factor: "wide", label: "Home desktop" },
    { src: "/preview/afri_pwa_mobile.png", sizes: "640x1344", type:
"image/png", label: "Home mobile" },
  ],
},
workbox: {
  navigateFallback: "/index.html",
  skipWaiting: true,
  clientsClaim: true,
  globPatterns: ["**/*.js,css,html,woff2,png"],
  globIgnores: [
    "**/node_modules/**/*",
    "**/sw.js",
    "**/sync-notification-sw.js",
    "**/workbox-*.js",
    "**/manifest.webmanifest",
  ],
  runtimeCaching: [
    {
      urlPattern: /\.?(?:js|css|woff2|html|png)$/,
      handler: "StaleWhileRevalidate",
      options: { cacheName: "static-assets" },
    },
    {
      urlPattern: new RegExp(`^${env.VITE_API_URL}`),

```

```

    handler: "NetworkOnly",
    options: { cacheName: "api-assets" },
  },
  {
    urlPattern: /.*/,
    handler: "NetworkOnly",
    options: { cacheName: "other-assets" },
  },
],
ignoreURLParametersMatching: [/^utm_/, /^fbclid$/],
navigateFallbackDenylist: [new RegExp(`^${env.VITE_API_URL}/`)],
},
}),
buildInfoPlugin,
],

css: {
  preprocessorOptions: {
    less: { javascriptEnabled: true, math: "strict" },
  },
},

resolve: {
  alias: { "@": "/src", "@assets": "/src/assets" },
},

build: {
  target: "es2022",
  cssTarget: "chrome90",
  sourcemap: false,
  minify: "terser",
  terserOptions: {
    compress: {
      ecma: TERSER_ECMA_LEVEL,
      passes: 3,
      drop_console: true,
      drop_debugger: true,
      pure_funcs: ["console.log", "console.info"],
      hoist_vars: true,
      hoist_funs: true,
    },
    format: { comments: false, ecma: TERSER_ECMA_LEVEL, wrap_iife: true
  },
  mangle: { properties: { regex: /^_/, reserved: ["React", "ReactDOM"] } },
},
rollupOptions: {
  output: {

```

```

    chunkFileNames: "assets/js/[name]-[hash].js",
    entryFileNames: "assets/js/[name]-[hash].js",
    assetFileNames: "assets/[ext]/[name]-[hash].[ext]",
    manualChunks(id) {
      if (id.includes("node_modules")) return "vendor";
    },
  },
  preserveEntrySignatures: "strict",
},
chunkSizeWarningLimit: 500,
},

optimizeDeps: {
  include: ["react", "react-dom", "react-router-dom", "zustand", "dexie"],
  esbuildOptions: { target: "es2022", pure: ["console.log"] },
},
};

return config;
});

```

A.2 Error Boundary

```

import { Component, ErrorInfo, ReactNode } from "react";
import "./ErrorBoundary.less";

interface Props {
  children: ReactNode;
}

interface State {
  hasError: boolean;
  errorMessage?: string;
  errorDetails?: {
    name: string;
    message: string;
    stack?: string;
    componentStack?: string;
    timestamp: string;
  };
  consoleHistory: string[];
  updateMessage?: string;
  buildVersion: string;
}

function isPWAUpdateError(error: Error): boolean {
  const msg = error.message;

```

```

    return (
      error.name === "ChunkLoadError" ||
      /Loading chunk .* failed/.test(msg) ||
      /Failed to fetch dynamically imported module/.test(msg)
    );
  }

  const consoleBuffer: string[] = [];
  const originalConsoleError = console.error;
  const originalConsoleWarn = console.warn;

  console.error = (...args: any[]) => {
    consoleBuffer.push(`[ERROR] ${new Date().toISOString(): ${args.map(arg
=> String(arg)).join(' ')} `);
    if (consoleBuffer.length > 50) consoleBuffer.shift();
    originalConsoleError.apply(console, args);
  };

  console.warn = (...args: any[]) => {
    consoleBuffer.push(`[WARN] ${new Date().toISOString(): ${args.map(arg =>
String(arg)).join(' ')} `);
    if (consoleBuffer.length > 50) consoleBuffer.shift();
    originalConsoleWarn.apply(console, args);
  };

  export default class ErrorBoundary extends Component<Props, State> {
    static RELOAD_ATTEMPTS_KEY = "pwa_reload_attempts";

    constructor(props: Props) {
      super(props);
      this.state = {
        hasError: false,
        consoleHistory: [],
        buildVersion: "0",
      };
    }

    shouldComponentUpdate(_nextProps: Props, nextState: State) {
      if (this.state.hasError !== nextState.hasError) {
        return true;
      }
      if (this.state.updateMessage !== nextState.updateMessage) {
        return true;
      }
      return false;
    }
  }

```

```

async componentDidMount() {
  if (!import.meta.env.DEV) {
    try {
      const response = await fetch('/build-info.json');
      const buildInfo = await response.json();
      this.setState({ buildVersion: buildInfo.buildVersion });
    } catch (error) {
      this.setState({ buildVersion: "Failed to load build version" });
      console.error("Error loading build info:", error);
    }
  }
}

componentDidCatch(error: Error, errorInfo: ErrorInfo) {
  if (isPWAUpdateError(error)) {
    const attempts =
      parseInt(sessionStorage.getItem(ErrorBoundary.RELOAD_ATTEMPTS_KEY) ||
        "0", 10);
    if (attempts === 0) {
      sessionStorage.setItem(ErrorBoundary.RELOAD_ATTEMPTS_KEY,
        "1");
      this.setState({ updateMessage: "New version available. Reloading..." });
      window.location.reload();
    } else if (attempts === 1) {
      sessionStorage.setItem(ErrorBoundary.RELOAD_ATTEMPTS_KEY,
        "2");
      this.setState({ updateMessage: "Sorry, updating application. Retrying in
        3 seconds..." });
      setTimeout(() => window.location.reload(), 3000);
    } else {
      sessionStorage.removeItem(ErrorBoundary.RELOAD_ATTEMPTS_KEY);
      this.handleError(error, errorInfo);
    }
  } else {
    this.handleError(error, errorInfo);
  }
}

handleError(error: Error, errorInfo: ErrorInfo) {
  this.setState({
    hasError: true,
    errorDetails: {
      name: error.name || 'Unknown Error',
      message: error.message || 'No message available',
      stack: error.stack || 'No stack trace available',
    }
  });
}

```

```

        componentStack: errorInfo.componentStack || 'No component stack
available',
        timestamp: new Date().toISOString(),
    },
    consoleHistory: [...consoleBuffer],
  });
}

```

```

copyToClipboard = () => {
  const { errorDetails, consoleHistory, buildVersion } = this.state;
  const isOnline = typeof window !== "undefined" &&
window.navigator.onLine;

  const textToCopy = `
Application Error Occurred
Please try refreshing the page or contact support if the problem persists.

```

```

Build Version: ${buildVersion}

```

Error Details:

```

Type: ${errorDetails?.name}
Message: ${errorDetails?.message}
Timestamp: ${errorDetails?.timestamp}

```

Stack Trace:

```

${errorDetails?.stack}

```

Component Stack:

```

${errorDetails?.componentStack}

```

Recent Console Messages (\${consoleHistory.length}):

```

${consoleHistory.length > 0 ? consoleHistory.join('\n') : 'No console messages
captured'}

```

System Information:

```

Online: ${isOnline.toString()}
User Agent: ${navigator.userAgent}
Language: ${navigator.language}
URL: ${window.location.href}
.trim();

```

```

navigator.clipboard.writeText(textToCopy)
.then(() => {
  alert("Error details copied to clipboard!");
})
.catch((err) => {
  console.error("Failed to copy text: ", err);
  alert("Failed to copy error details.");
}

```

```

    });
  };

  render() {
    const { updateMessage, hasError, errorDetails, consoleHistory, buildVersion
  } = this.state;
    const isOnline = typeof window !== "undefined" &&
window.navigator.onLine;

    if (updateMessage) {
      return (
        <div className="error-boundary">
          <p>{updateMessage}</p>
        </div>
      );
    }

    if (hasError && errorDetails) {
      return (
        <div className="error-boundary">
          <h1 className="error-boundary__title">Application Error
Occurred</h1>

          <p className="error-boundary__info">
            Please try refreshing the page or contact support if the problem
persists.
          </p>

          <p className="error-boundary__info">
            <strong>Build Version:</strong> {buildVersion}
          </p>

          {isOnline && (
            <div className="error-boundary__iframe-container">
              <iframe
                src="https://chromedino.com/"
                title="Google Dino Game"
                className="error-boundary__iframe"
                scrolling="no"
              />
            </div>
            <p className="error-boundary__instructions">
              Press space to start. Press space to jump and press the down
arrow to dodge.
            </p>
          )}
        </div>
      );
    }
  }
}

```

```

    })

    <section className="error-boundary__section">
      <h2>Error Details</h2>
      <pre className="error-boundary__details">
        <div><strong>Type:</strong> {errorDetails.name}</div>
        <div><strong>Message:</strong> {errorDetails.message}</div>
        <div><strong>Timestamp:</strong>
{errorDetails.timestamp}</div>
      </pre>
    </section>

    <section className="error-boundary__section">
      <h2>Stack Trace</h2>
      <pre className="error-
boundary__stack">{errorDetails.stack}</pre>
    </section>

    <section className="error-boundary__section">
      <h2>Component Stack</h2>
      <pre className="error-boundary__component-
stack">{errorDetails.componentStack}</pre>
    </section>

    <section className="error-boundary__section">
      <h2>Recent Console Messages ({consoleHistory.length})</h2>
      <pre className="error-boundary__console">
        {consoleHistory.length > 0
        ? consoleHistory.join('\n')
        : 'No console messages captured'}
      </pre>
    </section>

    <section className="error-boundary__section">
      <h2>System Information</h2>
      <pre className="error-boundary__system">
        <div><strong>Online:</strong> {isOnline.toString()}</div>
        <div><strong>User Agent:</strong>
{navigator.userAgent}</div>
        <div><strong>Language:</strong> {navigator.language}</div>
        <div><strong>URL:</strong> {window.location.href}</div>
      </pre>
    </section>

    <p
      className="error-boundary__info error-boundary__copy-text"
      onClick={this.copyToClipboard}

```

```

        style={{ cursor: "pointer", textDecoration: "underline" }}
      >
        Click here to copy error details to clipboard
    </p>

```

A.3 index.html

```

<!doctype html>
<html lang="en">
<head>
  <meta charset="UTF-8" />
  <meta name="viewport" content="width=device-width, initial-scale=1.0" />
  <meta http-equiv="Cache-Control" content="no-store" />
  <link rel="manifest" href="/manifest.webmanifest" />
  <link rel="icon" type="image/png" sizes="512x512" href="/android/android-launchericon-512-512.png" />
  <link rel="icon" type="image/png" sizes="192x192" href="/android/android-launchericon-192-192.png" />
  <link rel="icon" type="image/png" sizes="144x144" href="/android/android-launchericon-144-144.png" />
  <link          rel="icon"          type="image/png"          sizes="16x16"
href="/windows11/Square44x44Logo.targetsize-16.png" />
  <link          rel="icon"          type="image/png"          sizes="32x32"
href="/windows11/Square44x44Logo.targetsize-32.png" />
  <link          rel="icon"          type="image/png"          sizes="72x72"
href="/windows11/Square44x44Logo.targetsize-72.png" />
  <link          rel="icon"          type="image/png"          sizes="96x96"
href="/windows11/Square44x44Logo.targetsize-96.png" />
  <link          rel="icon"          type="image/png"          sizes="256x256"
href="/windows11/Square44x44Logo.targetsize-256.png" />
  <link rel="icon" type="image/png" sizes="48x48" href="/android/android-launchericon-48-48.png" />
  <link rel="apple-touch-icon" sizes="180x180" href="/ios/180.png" />

```

```

<link rel="apple-touch-icon" sizes="1024x1024" href="/ios/1024.png" />

<meta name="theme-color" content="#ffffff" />
<meta name="apple-mobile-web-app-status-bar-style" content="default" />

<title>MagDip - Delivery App</title>
<script type="module" src="/dvw-polyfill.js"></script>
</head>
<body>
<div id="root"></div>

<script type="module" src="./src/index.tsx"></script>
</body>
</html>

```

A.4 sync-notification-sw.js

```

const ICON = '/android/android-launchericon-192-192.png';
const BADGE = '/android/android-launchericon-48-48.png';
const TAG_PERIODIC = 'pending-count-check';
const TAG_SYNC = 'send-notification';
self.addEventListener('install', event => {
  event.waitUntil(
    caches.open('notification-assets')
      .then(c => c.addAll([ICON, BADGE]))
      .then(() => self.skipWaiting())
  );
});
self.addEventListener('activate', event => {
  event.waitUntil(self.clients.claim());
});
self.addEventListener('notificationclick', event => {
  event.notification.close();

```

```

    event.waitFor(client.openWindow(event.notification.data.url));
  });
  self.addEventListener('sync', event => {
    if (event.tag !== TAG_SYNC) return;
    event.waitFor(showQueuedNotification());
  });
  self.addEventListener('periodicsync', event => {
    if (event.tag !== TAG_PERIODIC) return;
    event.waitFor(showQueuedNotification());
  });
  async function showQueuedNotification(extra = "") {
    const cache = await caches.open('sync-queue-metadata');
    const res = await cache.match('/pending-count');
    if (!res) return;
    const count = parseInt(await res.text(), 10);
    if (Number.isNaN(count) || count <= 0) return;
    await self.registration.showNotification(
      `${count} Pending Items`,
      {
        body: ` You have ${count} pending items. Visit the app to sync them!`,
        icon: ICON,
        badge: BADGE,
        vibrate: [200, 100, 200],
        data: { url: '/' }
      }
    );
  }
}

```

ДОДАТОК Б
Слайди презентації



Дипломна робота на тему:

Дослідження та розробка оффлаїн- пріоритетного прогресивного вебзастосунку

Виконав:
студент 2 курсу, групи КНТ-124м
Керівник: к.т.н., доцент

Максим ТРІНКАЛЬ
Ольга ГЛАДКОВА



1

Рисунок Б.1 – Слайд 1

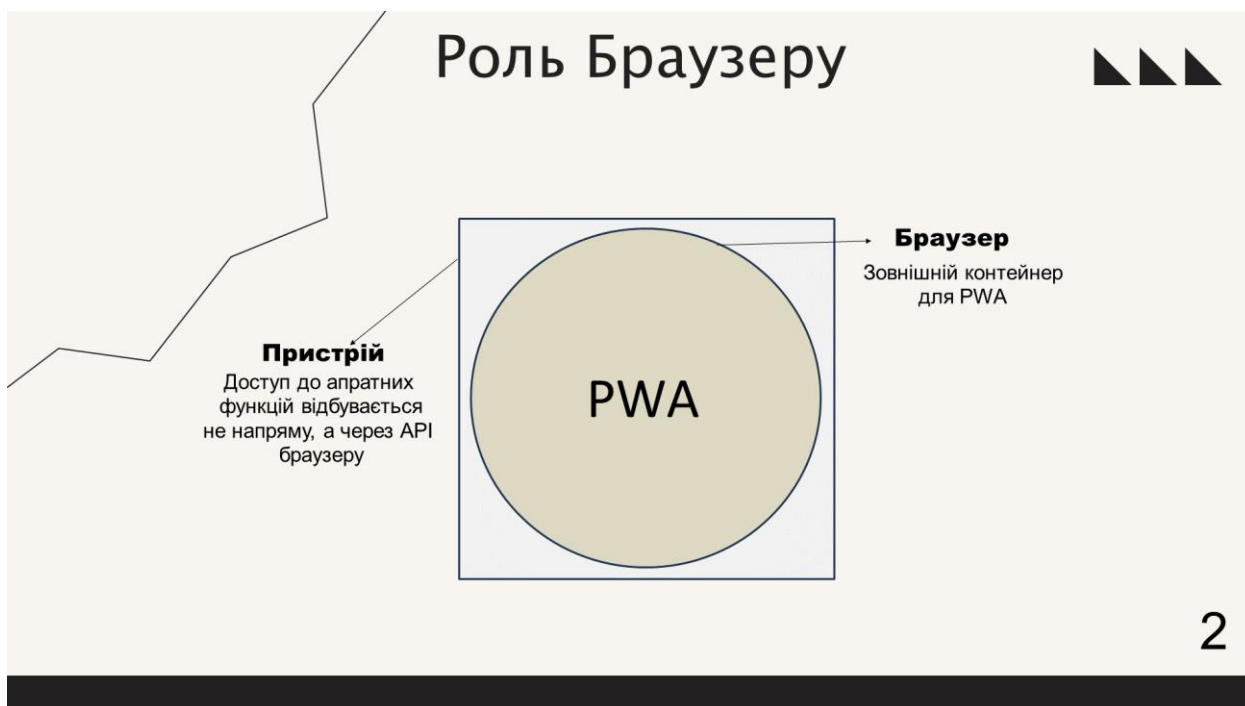


Рисунок Б.2 – Слайд 2

Чому саме PWA рішення ?



Офлайн-режим

Кур'єри працюють у зонах із поганим покриттям



Кросс-платформеність

Працює на Android, iOS, Windows, macOS без додаткових витрат.



Автоматичні оновлення

Без Google Play або App Store — важливо для корпоративних систем.



Миттєве відкриття

Без встановлення великих обсягів даних — працює навіть на старих пристроях.



Уніфікований UX

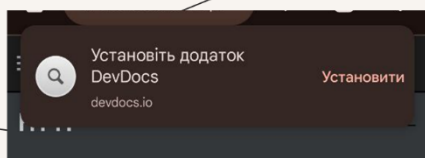
Однаковий UX на всіх пристроях спрощує навчання персоналу.



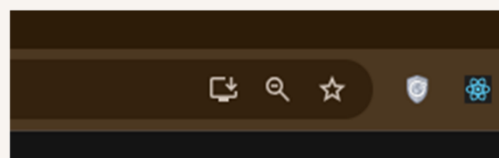
3

Рисунок Б.3 – Слайд 3

Встановлення PWA



Пропозиція встановлення у Chrome на Android

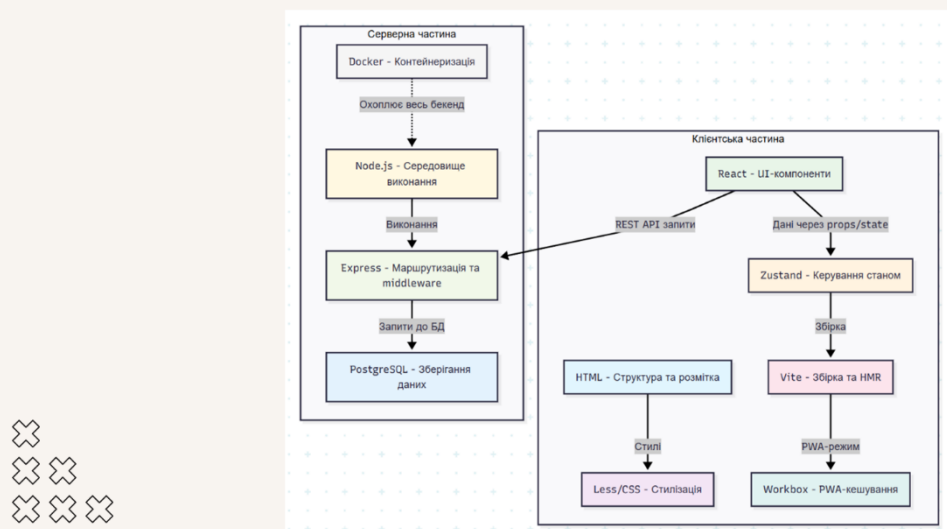


Іконка встановлення у Chrome на комп'ютері

4

Рисунок Б.4 – Слайд 4

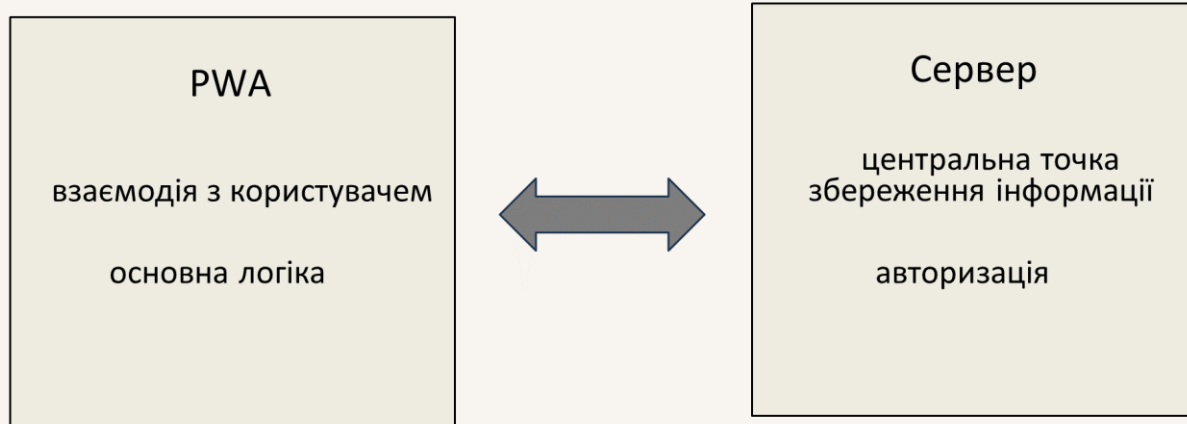
Технології розробки



5

Рисунок Б.5 – Слайд 5

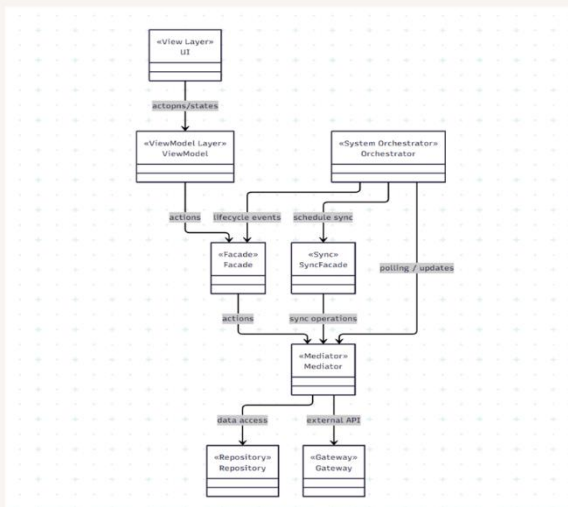
Розподіл відповідальності між сервером і застосунком



6

Рисунок Б.6 – Слайд 6

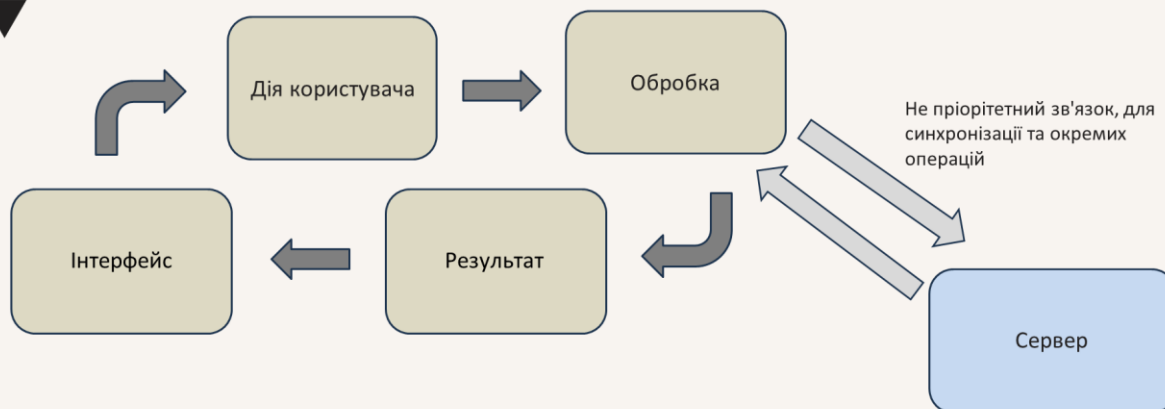
Архітектура PWA застосунку



7

Рисунок Б.7 – Слайд 7

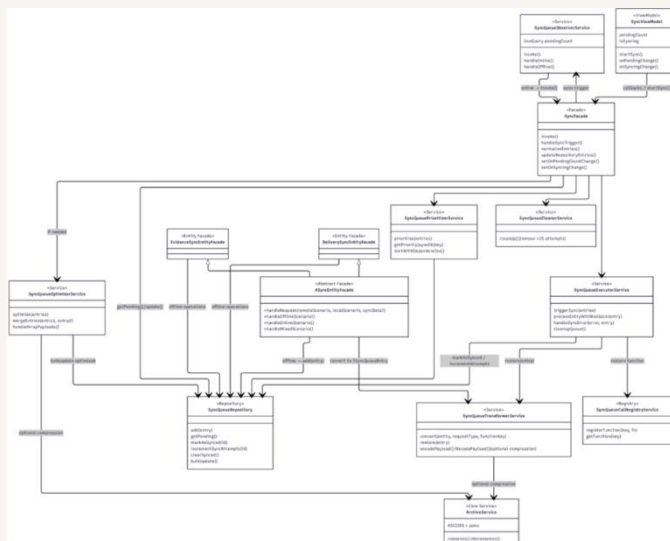
Оффлайн пріорітетний підхід



8

Рисунок Б.8 – Слайд 8

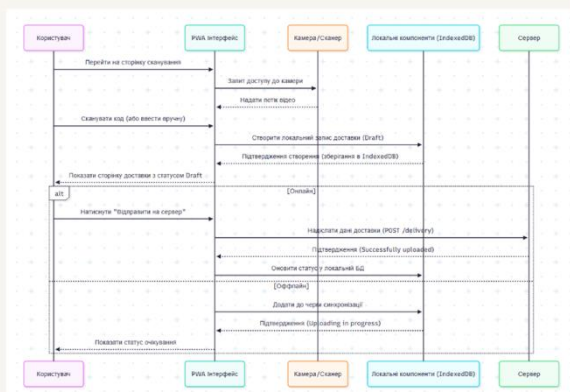
Система синхронізації даних



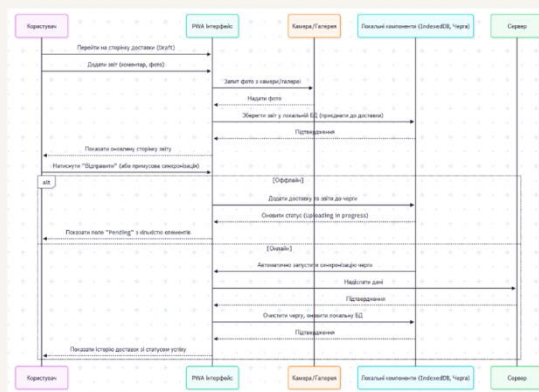
9

Рисунок Б.9 – Слайд 9

Сценарії користування



Сценарій нового сканування



Сценарій додавання звіту

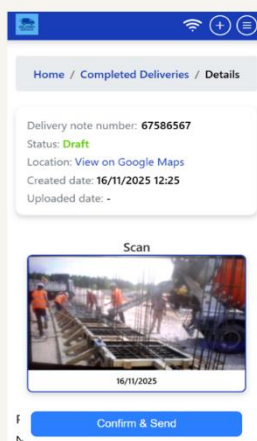
10

Рисунок Б.10 – Слайд 10

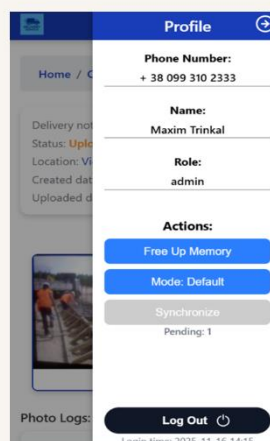
Інтерфейс користувача



Сторінка сканування



Сторінка доставки



Меню користувача та поле пендінг



11

Рисунок Б.11 – Слайд 11

Проблеми під час розробки



12

Рисунок Б.12 – Слайд 12

Висновки

- Створений застосунок підтверджує стабільний досвід навіть у складних сценаріях
- Низька вартість та швидкість розробки відкривають нові можливості для бізнесу
- PWA підхід забезпечує швидкість, мобільність та універсальність
- PWA поєднують найкращі властивості нативних застосунків та веб-доступності
- PWA — перспективне майбутнє кросплатформеної розробки



Рисунок Б.13 – Слайд 13