Міністерство освіти і науки України
НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ «КИЄВО-МОГИЛЯНСЬКА АКАДЕМІЯ»
Кафедра мультимедійних систем

СТВОРЕННЯ ВЕБ-ЗАСТОСУНКІВ НА ОСНОВІ ТЕХНОЛОГІЇ PROGRESSIVE WEB APPLICATIONS

Текстова частина до курсової роботи
за спеціальністю «Інженерія програмного забезпечення»

Студент: Українець Дмитро Сергійович
Інженерія програмного забезпечення, курс 3
Науковий керівник: кандидат технічних наук
Олецький Олексій Віталійович

Київ 2025

Зміст

Анотація 4
Вступ 5
1 Аналіз технологій для створення PWA 7
1.1 Історія виникнення та суть концепції Progressive Web Applications 7
1.2 Основні характеристики та переваги PWA 9
1.3 Порівняння PWA з іншими підходами 11
1.4 Компоненти та браузерні API, що використовуються в PWA 13
2 Проєктування та вибір стеку технологій 15
2.1 Компоненти та браузерні API, що використовуються в PWA 15
2.2 Огляд і вибір інструментів для реалізації клієнтської частини 16
2.3 Огляд і вибір інструментів для реалізації серверної частини 17
2.4 Безпека в PWA 19
3 Реалізація застосунку Spend Tracker 21
3.1 Загальний опис функціональності застосунку 21
3.1.1 Транзакції 21
3.1.2 Бюджет 22
3.1.3 Аналітика 23
3.1.4 Сповіщення 23
3.1.5 Офлайн-режим 24
3.1.6 Авторизація та реєстрація 24
3.1.7 Категорії 25
3.1.8 Налаштування 26
3.1.9 Користувач 26
3.2 Реалізація офлайн-режиму з використанням Service Worker 27
3.3 Реалізація локального збереження даних у IndexedDB 31
3.4 Push-сповіщення та фонова синхронізація 33
3.4.1 Реєстрація підписки на сповіщення 35
3.4.2 Збереження підписки на сервері 35
3.4.3 Надсилання повідомлень із сервера 36
3.4.4 Обробка сповіщень у браузері 36
3.4.5 Фонова синхронізація транзакцій та підписок 36
4 Тестування та оцінка результатів 37
4.1 Тестування функціональності 37
4.1.1 Установка на пристрої 37
4.1.2 Перевірка роботи офлайн 37
4.1.3 Робота push-сповіщень у різних браузерах 38
4.2 Оцінка зручності користування 38
4.2.1 Адаптивність 38
4.2.2 UX-аналітика 38
4.3 Порівняння з подібними рішеннями 39

Висновки 41
Список використаної літератури 43
Додаток А 45
Додаток Б 52
Додаток В 53
Додаток Г 55
Додаток Д 58

Анотація

Курсова робота присвячена створенню веб-застосунку на основі технології Progressive Web Applications (PWA), яка поєднує переваги веб- і нативних застосунків.

У межах роботи реалізовано функціональний фінансовий трекер витрат, що дозволяє користувачу додавати, переглядати та аналізувати особисті доходи і витрати. Застосунок підтримує офлайн-режим, фонову синхронізацію даних, push-сповіщення, кешування сторінок і ресурсів.

Для розробки використано стек технологій: Thymeleaf та Tailwind CSS для графічного інтерфейсу, JavaScript та Service Worker для клієнтської частини, а також та MySQL для серверної частини.

У текстовій частині проєкту розглянуто ключові аспекти архітектури PWA, особливості реалізації офлайн-функціональності, обробки сповіщень та безпеки. Отримані результати підтверджують ефективність застосування PWA для створення зручних і доступних веб-застосунків, які здатні функціонувати незалежно від наявності підключення до мережі.

Ключові слова: Progressive Web Application, Service Worker, офлайн-доступ.

Вступ

Сучасний розвиток інформаційних технологій характеризується зростанням ролі веб-застосунків, які набувають властивостей традиційних десктопних і мобільних застосунків. Однак значна частина веб-застосунків залежить від стабільності інтернет-з’єднання та доступу до мережі, що не завжди можливо забезпечити в умовах сучасного використання технологій та зважаючи на особливості розвитку певних регіонів світу. Вирішення цієї проблеми можливо через використання Progressive Web Applications (PWA), що поєднують можливості веб- та нативних застосунків, зокрема офлайн-доступ, push-сповіщення, швидку роботу та доступність без встановлення додаткових програмних засобів. Дослідження та практичне впровадження технології PWA є актуальним завданням, яке знаходить підтримку серед провідних технологічних компаній (Google, Microsoft, Twitter) і активно досліджується багатьма спеціалістами в галузі інформаційних технологій.

Метою даної курсової роботи є розробка та реалізація функціонального веб-застосунку у вигляді персонального фінансового трекера на основі технології Progressive Web Applications, що підтримує офлайн-режим, push-сповіщення та синхронізацію даних. Для досягнення поставленої мети визначено такі завдання:

а) Виконати аналіз технології Progressive Web Applications, визначивши її переваги та недоліки порівняно з традиційними веб- і нативними застосунками.

б) Систематизувати та обґрунтувати вибір технологічного стеку для реалізації клієнтської (Thymeleaf, Tailwind CSS, JavaScript) та серверної частин (Spring Boot, MySQL).

в) Розробити та описати архітектуру веб-застосунку, реалізувати підтримку офлайн-функціональності за допомогою Service Worker.

г) Реалізувати механізм надсилання push-сповіщень з використанням Web Push Protocol, серверної бібліотеки web-push та браузерних API (Push API, Notifications API).

ґ) Виконати комплексне тестування працездатності застосунку в різних умовах мережевого з’єднання, визначити шляхи його подальшої оптимізації та розширення функціональності.

Об’єктом дослідження є процес розробки та впровадження веб-застосунків на базі технології Progressive Web Applications.

Для виконання завдань курсової роботи застосовуються наступні методи: аналіз та систематизація технічної документації та веб-стандартів, експериментальне програмування, тестування веб-застосунку в умовах різної якості та наявності інтернет-з'єднання, використання інструментів для розробників Chrome DevTools, а також інтегроване середовище розробки IntelliJ IDEA.

В ході дослідження використовувалися матеріали та документація з таких ресурсів, як Magefan [1], MDN Web Docs [2], Google Web.dev [3,4], DAN.IT [5], Next.js [6], Ionic Framework [7], Smile Ukraine [8], Wezom [9], Chrome Developers [10], що допомогли структурувати технічні знання з даної теми.

Результати роботи можуть бути використані як готовий приклад реалізації PWA-застосунку для персонального фінансового обліку витрат та доходів, а також як методична основа для розробки інших подібних веб-застосунків, що мають забезпечувати стабільну роботу в умовах нестабільного, або ж відсутнього інтернет-з'єднання.

Аналіз технологій для створення PWA

1.1 Історія виникнення та суть концепції Progressive Web Applications

Поняття Progressive Web Applications (PWA) вперше було запропоноване інженером Google Алексом Расселом у 2015 році [11]. Він запропонував концепцію, яка дозволяє веб-застосункам працювати на мобільних пристроях з тією ж швидкістю, доступністю та функціональністю, що й нативні застосунки. Основна ідея полягала в тому, щоб зробити веб-додатки «прогресивними» – тобто такими, що поступово покращуються залежно від можливостей пристрою користувача.

На початку 2010-х років значною популярністю користувались нативні мобільні застосунки, розповсюджувані через магазини App Store і Google Play. Проте їх розробка вимагала створення окремих версій для Android та iOS, що збільшувало вартість проєктів. Крім того, доступність нативного застосунку залежала від платформи і потребувала схвалення в маркетплейсі. У відповідь на ці обмеження з'явилися гібридні технології (Cordova, Ionic), але вони не повністю відповідали очікуванням користувачів.

У цьому контексті PWA стала ефективною альтернативою: вона поєднує зручність використання та відкритість вебу з перевагами нативних застосунків. Progressive Web Apps можуть встановлюватися на головний екран пристрою, працювати офлайн, надсилати сповіщення, і при цьому не потребують встановлення з магазинів застосунків [2, 3, 4].

Сутність концепції полягає в тому, що PWA – це звичайний веб-застосунок, який використовує набір сучасних веб-API (Service Worker, Web App Manifest, Cache API тощо), щоб забезпечити:

а) Автономність (робота без інтернету);

б) Високу швидкість взаємодії;

в) Можливість встановлення на пристрій;

г) Підтримку push-сповіщень;

ґ) Безпеку завдяки HTTPS.

Однією з переваг PWA є відсутність необхідності у встановленні з маркетплейсів – користувач може просто натиснути «Додати на головний екран» у браузері. Застосунок буде відкриватися у власному вікні без адресного рядка, створюючи враження нативної програми [3].

Також важливим фактором стало те, що Google активно підтримує розвиток PWA, інтегруючи відповідні можливості в Chrome та надаючи детальну документацію [3, 4]. Схожі ініціативи були підтримані Mozilla Firefox та Microsoft Edge. У Safari підтримка PWA з'явилася дещо пізніше.

Переваги та недоліки технології PWA зведено в таблиці 1.1.

Таблиця 1.1 – Порівняння PWA-застосунків з класичними рішеннями

Характеристика	**PWA-застосунок**	**Нативний застосунок**	**Веб-застосунок**
Встановлення на пристрій	Так (через браузер)	Так (через маркетплейси)	Ні
Підтримка офлайн-режиму	Так	Так	Ні
Push-сповіщення	Так	Так	Ні
Доступ через браузер	Так	Ні	Так
Публікація в App Store/Google Play	Не обов'язкова	Обов'язкова	Не потрібна
Швидкість роботи	Висока	Висока	Залежить від інтернету
Доступ до апаратних ресурсів	Обмежений	Повний доступ	Ні
Оновлення застосунку	Автоматично через браузер	Через маркетплейс	Автоматично через сервер
Вартість розробки	Низька	Висока	Низька
Кросплатформеність	Так	Ні	Так

Таким чином, концепція Progressive Web Applications виникла як відповідь на потребу зробити веб-застосунки більш адаптивними до сучасного середовища користувача, забезпечивши функціональність нативних додатків без втрати універсальності вебу. Її поява стала новим етапом у розвитку веб-технологій, зробивши можливим створення застосунків нового покоління.

1.2 Основні характеристики та переваги PWA

Progressive Web Applications мають ряд ознак, які визначають їхню сутність та відрізняють від звичайних веб- і нативних застосунків. У 2015 році Google запропонував базовий набір критеріїв, за якими веб-застосунок можна вважати «прогресивним» [4]. До них належать:

а) Надійність (Reliability) — застосунок завжди відкривається і працює, незалежно від якості інтернет-з’єднання. Це забезпечується завдяки використанню Service Worker та Cache API.

б) Швидкість (Performance) — взаємодія з інтерфейсом має бути миттєвою. Висока швидкість досягається завдяки кешуванню та оптимізованому завантаженню ресурсів.

в) Залучення (Engagement) — PWA підтримує push-сповіщення, встановлюється на пристрій, працює в автономному вікні без адресного рядка, створюючи ефект нативного застосунку.

Також важливою характеристикою є принцип Progressive Enhancement — застосунок має функціонувати базово в усіх браузерах, але використовувати додаткові можливості, якщо браузер їх підтримує. Це гарантує широку доступність [2, 3].

До основних переваг PWA, які були виявлені на основі аналізу загальнодоступних документів Google Web.dev [3], MDN [2] і ряду професійних оглядів [5, 9], належать:

а) Можливість роботи в офлайн-режимі — завдяки Service Worker застосунок може працювати без доступу до мережі, зберігаючи кешовані дані або локальне сховище (IndexedDB).

б) Установка на головний екран — PWA може бути доданий до домашнього екрана, запускатися у повноекранному режимі без адресного рядка, що створює ефект нативної програми.

в) Push-сповіщення — дає змогу залучати користувачів навіть тоді, коли застосунок не відкрито. Сервер надсилає push-повідомлення, а Service Worker перехоплює та виводить повідомлення.

г) Автоматичне оновлення — завдяки Service Worker, оновлення застосунку можуть відбуватися у фоновому режимі, коли користувач під’єднується до мережі.

ґ) Незалежність від маркетплейсів — немає потреби у розміщенні в App Store або Google Play, чи інших маркетплейсах, що спрощує публікацію у відкритий доступ.

д) Безпека — всі PWA працюють через HTTPS, що гарантує захист даних користувача від атак типу «людина посередині» [12].

е) Легка підтримка — PWA – це єдиний код, який працює в багатьох браузерах і на багатьох пристроях (десктоп, мобільні телефони, планшети тощо).

Переваги PWA можуть мати значний вплив на бізнес-процеси. За даними аналітики Google, після впровадження PWA компанії Starbucks, Pinterest, Trivago, Twitter та Uber спостерігали зростання залученості користувачів, пришвидшення завантаження і навіть зменшення розміру застосунку у десятки разів порівняно з нативними аналогами [3, 5].

Таблиця 1.2 – Успішні результати впровадження PWA-застосунків

Компанія	**Результат впровадження PWA**
Starbucks	Зменшення розміру застосунку з 148 МБ до 233 КБ, офлайн-функціональність, удвічі більше замовлень через веб у порівнянні з нативним додатком [13].
Pinterest	Збільшення часу взаємодії на 40%, зростання рекламних доходів на 44%, збільшення кількості користувачів на 60% [13].
Trivago	Збільшення повторних візитів користувачів на 150%, push-сповіщення сприяли підвищенню залученості [14].
Twitter (X)	Twitter Lite став головною мобільною версією Twitter, що споживає на 70% менше даних [15].
Uber	Застосунок працює навіть при 2G і запускається менше ніж за 3 секунди [13].

1.3 Порівняння PWA з іншими підходами

Щоб повною мірою оцінити переваги технології Progressive Web Applications, важливо розглянути її у порівнянні з традиційними веб-застосунками та нативними мобільними застосунками. Кожен із цих підходів має свої переваги та недоліки, і вибір між ними залежить від цілей, бюджету, очікуваної аудиторії та технічних вимог проєкту [3, 4]:

а) Нативні застосунки створюються під конкретну платформу — Android або iOS — з використанням відповідних мов програмування та інструментів (наприклад, Kotlin/Java для Android, Swift/Objective-C для iOS). Вони мають повний доступ до функціональності пристрою (камера, Bluetooth, GPS, файлове сховище тощо) і забезпечують найкращу продуктивність. Однак вимагають тривалого процесу розробки, підтримки кількох кодових баз та обов’язкової публікації в App Store/Google Play з дотриманням вимог модерації.

б) Класичні веб-застосунки — це динамічні або статичні сайти, які запускаються в браузері. Вони не потребують встановлення, швидко розгортаються та оновлюються, мають єдину кодову базу. Проте втрачають у зручності використання: не мають офлайн-режиму, не інтегруються в операційну систему пристрою, не підтримують push-сповіщення та часто повільніші у роботі порівняно з нативними додатками.

в) Progressive Web Applications об'єднують переваги попередніх двох підходів. Вони працюють у браузері, але можуть бути встановлені як застосунки, мати офлайн-режим, підтримувати push-сповіщення та запускатись у повноекранному режимі. Розробка PWA здійснюється один раз для всіх платформ, і не потребує схвалення маркетплейсів. Основні обмеження стосуються доступу до деякої функціональності пристрою (наприклад, Bluetooth, повноцінний файловий доступ тощо).

Для наочного відображення відмінностей можна використати порівняльну таблицю 1.1, де показано ключові характеристики трьох типів застосунків за критеріями: офлайн-підтримка, продуктивність, доступ до апаратних ресурсів, вимоги до встановлення, оновлення, вартість розробки, кросплатформеність тощо.

1.4 Компоненти та браузерні API, що використовуються в PWA

Технологія Progressive Web Applications базується на використанні ряду сучасних веб-API, які дозволяють створити ряд функціональних можливостей, наближених до нативних застосунків. Ці компоненти реалізуються на рівні браузера, і для їх ефективного використання необхідне HTTPS-з'єднання та підтримка відповідного API з боку браузера. Схему взаємодії компонентів PWA між собою зображено на рисунку 1.1. Серед основних компонентів PWA виділяють наступні [3]:

а) Service Worker — це фоновий скрипт, який виконується окремо від основного потоку веб-сторінки. Він дає змогу перехоплювати мережеві запити, кешувати ресурси, реалізовувати офлайн-доступ, push-сповіщення, синхронізацію даних у фоновому режимі тощо. Service Worker є основою для реалізації більшості PWA-функцій (додаток А).

б) Web App Manifest — JSON-файл, який містить метаінформацію про застосунок: назву, іконки, колір теми, початкову сторінку, режим відображення, орієнтацію екрана тощо. Цей файл необхідний для встановлення PWA на головний екран пристрою (додаток Б).

в) Cache API — інтерфейс для зберігання запитів та відповідей HTTP, що дає змогу повторно використовувати ресурси при наступних зверненнях або в офлайн-режимі. Зазвичай використовується спільно з Service Worker для створення сценаріїв кешування (наприклад, стратегія «cache first»).

г) IndexedDB — асинхронне сховище даних у браузері, яке дозволяє зберігати структуровану інформацію (об'єкти, масиви тощо) у вигляді ключ-значення. Застосовується для зберігання локальних даних, які синхронізуються з сервером після відновлення з'єднання. У PWA використовується для офлайн-записів, налаштувань, push-підписок.

ґ) Push API — дозволяє веб-застосунку отримувати push-повідомлення з сервера, навіть коли сторінка не активна. Працює в парі з Service Worker.

Використовується для надсилання сповіщень з бекенду на основі підписки користувача.

д) Notifications API — відповідає за відображення push-повідомлень на пристрої користувача. Після отримання повідомлення через Push API, Service Worker викликає метод showNotification, який показує повідомлення.

е) Background Sync API — дозволяє Service Worker виконувати певні дії (наприклад, надсилання форми або синхронізацію) після відновлення інтернет-з’єднання. Це зручно для обробки офлайн-транзакцій, черг дій, запізнілих push-підсумків тощо.

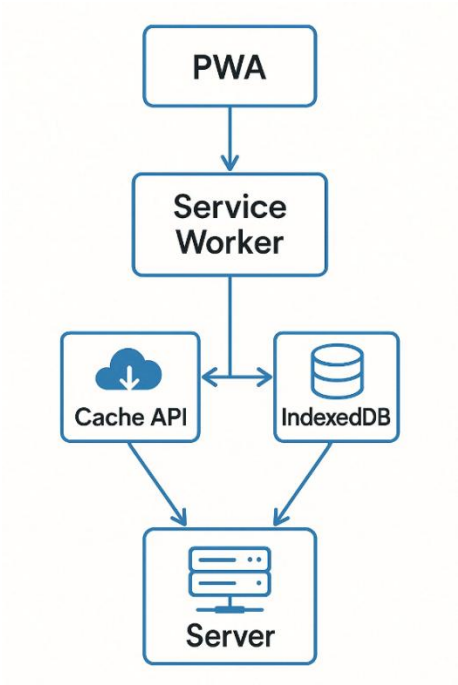

Рисунок 1.1 – Схема взаємодії компонентів PWA

Проєктування та вибір стеку технологій

Компоненти та браузерні API, що використовуються в PWA

Перед початком розробки PWA-застосунку важливо визначити загальну архітектуру системи, яка б забезпечувала ефективну взаємодію між клієнтською та серверною частинами, а також підтримку офлайн-функціональності та інших ключових характеристик PWA, що були наведені у попередніх розділах. Зважаючи на функціональні вимоги до застосунку (облік транзакцій, робота з бюджетом, аналітика, сповіщення тощо), було обрано клієнт-серверну архітектуру з розділенням відповідальності.

Архітектура базується на класичному підході «розділення шарів», де клієнтська частина відповідає за взаємодію з користувачем, а серверна — за обробку даних, авторизацію, логіку сповіщень та зберігання інформації в базі даних. Такий підхід дає змогу масштабувати окремі компоненти, забезпечує безпеку, легкість у тестуванні та адаптивність до різних пристроїв.

Загальна схема застосунку виглядає наступним чином:

а) Клієнт (браузер) — HTML-сторінки згенеровані за допомогою Thymeleaf, стилізація через Tailwind CSS, логіка — JavaScript з підтримкою IndexedDB, Cache API та Service Worker.

б) Сервер (Spring Boot) — REST API для обробки запитів, push-підписки, аутентифікація користувачів, логіка сповіщень, підключення до бази даних MySQL.

в) База даних — MySQL з реляційною структурою, зокрема таблицями для користувачів, транзакцій, категорій, push-підписок тощо.

У розробці також використовується принцип «прогресивного покращення» [16]: застосунок працює у базовому режимі в будь-якому сучасному браузері, але використовує розширену функціональність, якщо браузер підтримує відповідні API.

Оскільки застосунок орієнтований на доступність, стабільну роботу в різних умовах мережі та просту інтеграцію з серверною логікою на Spring

Boot, було вирішено використати серверний рендеринг із шаблонізатором Thymeleaf. Це дало змогу легко реалізувати кешування, забезпечити SEO-дружність, а також мінімізувати залежність від зовнішніх JavaScript-фреймворків. Водночас, на стороні клієнта активно використовується JavaScript для реалізації динамічної взаємодії, обробки форм, роботи з IndexedDB, завантаження ресурсів у фоновому режимі, реєстрації Service Worker та взаємодії з API браузера.

2.2 Огляд і вибір інструментів для реалізації клієнтської частини

Розробка клієнтської частини PWA-застосунку потребує вибору інструментів, які б забезпечили ефективну генерацію сторінок, стилізацію графічного інтерфейсу користувача, взаємодію з браузерними API, а також роботу в офлайн-режимі.

а) Thymeleaf — це серверний шаблонізатор для Java, який широко використовується для генерації HTML-сторінок на боці сервера. Його перевагою є природна інтеграція зі Spring Boot, підтримка логіки на рівні шаблонів, можливість передачі даних через модель та простота адаптації під динамічний контент. Використання Thymeleaf дозволяє уникнути зайвої складності, пов’язаної з клієнтськими фреймворками, при цьому забезпечує високий рівень контролю над HTML-структурою сторінок.

б) Tailwind CSS — сучасний CSS-фреймворк, що дозволяє стилізувати компоненти без написання власних CSS-файлів, що значно пришвидшує розробку графічного інтерфейсу користувача. Його основними перевагами є гнучкість, підтримка адаптивного дизайну, зменшення повторень стилів і зручність при створенні інтерфейсу з мобільною орієнтацією. Tailwind добре підходить ддя розробки PWA-застосунків завдяки легкості кастомізації та малій вазі кінцевого CSS після збирання.

в) JavaScript — використовується для реалізації клієнтської логіки, зокрема:

- динамічне завантаження контенту;
- обробка подій;
- робота з IndexedDB;
- реєстрація Service Worker (додаток В);
- взаємодія з браузерними API;

г) IndexedDB — браузерна база даних NoSQL для збереження структурованих даних. У застосунку вона використовується, зокрема для тимчасового зберігання транзакцій у режимі офлайн. Після відновлення з’єднання дані синхронізуються з сервером (додаток Г).

ґ) Service Worker — один із ключових компонентів PWA-застосунку. Скрипт, який виконується у фоновому режимі, дозволяє перехоплювати мережеві запити, кешувати сторінки та обробляти push-повідомлення. У реалізації застосунку Service Worker відповідає за кешування основних сторінок, скриптів і стилів, а також за підписку та відображення сповіщень (додаток А).

д) Web App Manifest — JSON-файл, який задає основну метаінформацію про застосунок (назва, іконки, колір теми тощо) та дозволяє браузеру відображати пропозицію встановлення PWA. Він інтегрується у <head> головної сторінки (або у кожній сторінці) й підтримується всіма сучасними браузерами (додаток Б).

2.3 Огляд і вибір інструментів для реалізації серверної частини

Серверна частина застосунку реалізована за допомогою фреймворку Spring Boot, який є одним із найпопулярніших рішень для створення веб-застосунків на мові програмування Java. Його перевагами є висока швидкодія, розширюваність, зручність конфігурації, підтримка RESTful API

та інтеграція з системами безпеки й базами даних. У проєкті Spring Boot забезпечує обробку HTTP-запитів, автентифікацію користувачів, логіку опрацювання транзакцій, роботу з підписками на push-сповіщення та щоденну генерацію повідомлень.

Для зберігання даних використовується реляційна база MySQL, яка обрана за її стабільність, поширеність, підтримку складних SQL-запитів, наявність засобів резервного копіювання і відновлення, а також сумісність із Spring Data JPA. У структурі бази даних передбачені таблиці для користувачів, транзакцій, бюджетів, категорій, підписок на сповіщення, журналу подій (логів) та статусу синхронізацій.

Основні компоненти серверної частини:

а) Контролери — класи PushController, TransactionController, UserController та інші обробляють HTTP-запити, що надходять від клієнтської частини. Вони відповідають за перевірку авторизації користувача, обробку вхідних даних та передачу запитів до відповідних сервісів для виконання бізнес-логіки.

б) Сервіси — логіка застосунку зосереджена в сервісах, таких як TransactionService, PushSubscriptionService тощо. Тут реалізовано обчислення підсумків витрат та доходів, формування повідомлень, перевірку доступу та збереження інформації до БД.

в) Репозиторії — через Spring Data JPA такі репозиторії, як TransactionRepository, PushSubscriptionRepository, UserRepository та інші здійснюють доступ до бази даних.

г) Безпека — застосунок реалізує автентифікацію за email та паролем, обмежує доступ до API залежно від ролі користувача, захищає маршрути через фільтри безпеки. Нові користувачі реєструються з роллю USER, а адміністратори мають окремий інтерфейс, в якому мають змогу переглядати дані профілю користувачів та змінювати роль та ім’я користувача (додаток Д).

ґ) Підтримка розкладу — щоденне надсилання підсумку витрат та доходів за попередній день реалізовано через анотацію @Scheduled, яка запускається у визначений час за допомогою планувальника Spring.

д) Надсилання push-сповіщень — реалізовано за допомогою бібліотеки web-push, яка використовує VAPID-ключі для взаємодії з браузерними push-сервісами. Сервіс PushSubscriptionService формує повідомлення, кодує їх і надсилає користувачам.

Безпека в PWA

Забезпечення безпеки у Progressive Web Applications є критично важливим, оскільки такі застосунки мають доступ до чутливих даних користувача, можуть зберігати інформацію у браузері, працювати у фоновому режимі та надсилати сповіщення. Для забезпечення конфіденційності, цілісності даних і захисту від типових веб-атак у застосунку реалізовано кілька рівнів захисту, зокрема налаштування Spring Security в додатку Д:

а) HTTPS як обов’язкова умова — усі PWA обов’язково мають працювати через захищене з’єднання HTTPS. Це гарантує шифрування трафіку, захист від атак типу «людина посередині» [20] та дозволяє використовувати API, які недоступні по HTTP (наприклад, Service Worker, Push API, Cache API) [2]. Сертифікат SSL встановлено на сервері, що дозволяє безпечно обмінюватися даними між клієнтом і сервером.

б) Аутентифікація та авторизація користувачів — у застосунку використовується механізм входу за допомогою email та паролю. При реєстрації користувачу автоматично присвоюється роль USER. Доступ до окремих маршрутів, таких як адміністративна панель, обмежено лише для користувачів з роллю ADMIN. Це реалізовано, зокрема через Spring Security (додаток Д) та перевірки на рівні шаблонів Thymeleaf (наприклад, authorization.expression('hasRole("ADMIN")')).

в) Захист API і шифрування ключів — кожен запит до API перевіряється на наявність авторизації. Крім того, ключі push-підписок (VAPID auth і p256dh) зберігаються в базі даних у вигляді зашифрованих рядків, а не у відкритому вигляді. Це мінімізує ризик несанкціонованого доступу до підписок.

г) Захист даних у браузері — IndexedDB не містить критичних даних (наприклад, паролів), але використовується для зберігання транзакцій і черги дій у разі відсутності з’єднання. Дані з IndexedDB автоматично видаляються після синхронізації з сервером. Це дозволяє зменшити обсяг інформації, що зберігається локально.

ґ) Політика безпеки контенту — для захисту від атак типу XSS (вставка скриптів у сторінку) застосунок використовує політику CSP, яка обмежує джерела скриптів, стилів, зображень тощо.

Реалізація застосунку Spend Tracker

Загальний опис функціональності застосунку

У межах даної курсової роботи реалізовано PWA-застосунок під назвою Spend Tracker, який слугує для персонального обліку фінансів. Застосунок надає користувачеві можливість зручно фіксувати доходи та витрати, переглядати історію транзакцій, контролювати виконання бюджету, переглядати графіки аналітики та отримувати push-сповіщення з фінансовими підсумками.

Основні цілі застосунку:

а) надати користувачеві простий та інтуїтивний інтерфейс для управління персональними фінансами;

б) забезпечити офлайн-доступ до даних та основних дій;

в) реалізувати push-сповіщення для інформування про витрати/доходи;

г) продемонструвати можливості PWA у реальному прикладі.

Далі подано основні розділи функціональності демонстраційного застосунку, які були реалізовані в межах практичної частини курсової роботи.

3.1.1 Транзакції

Реалізована функціональність на сторінках «Огляд» та «Записи» на рисунку 3.1:

- додавання нової транзакції з вказанням суми, категорії, типу (дохід/витрата) та дати проведення транзакції;
- перегляд історії транзакцій зі швидким фільтруванням (за датою, типом, категорією та сумою);
- редагування та видалення транзакцій.

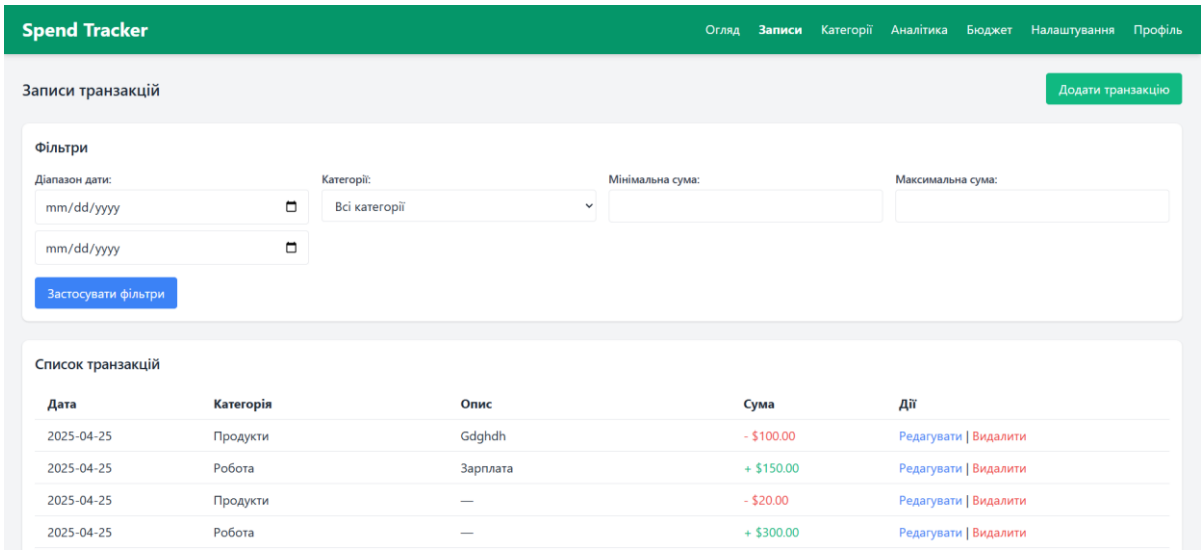

Рисунок 3.1 – Сторінка «Записи»

3.1.2 Бюджет

Реалізована функціональність на сторінках «Огляд» та «Бюджет» на рисунку 3.2:

– встановлення щомісячного бюджету користувача;
– автоматичне вирахування витрат за поточний місяць;
– відображення прогресу виконання бюджету;
– автоматичне розподілення витрат за категоріями за поточний місяць.

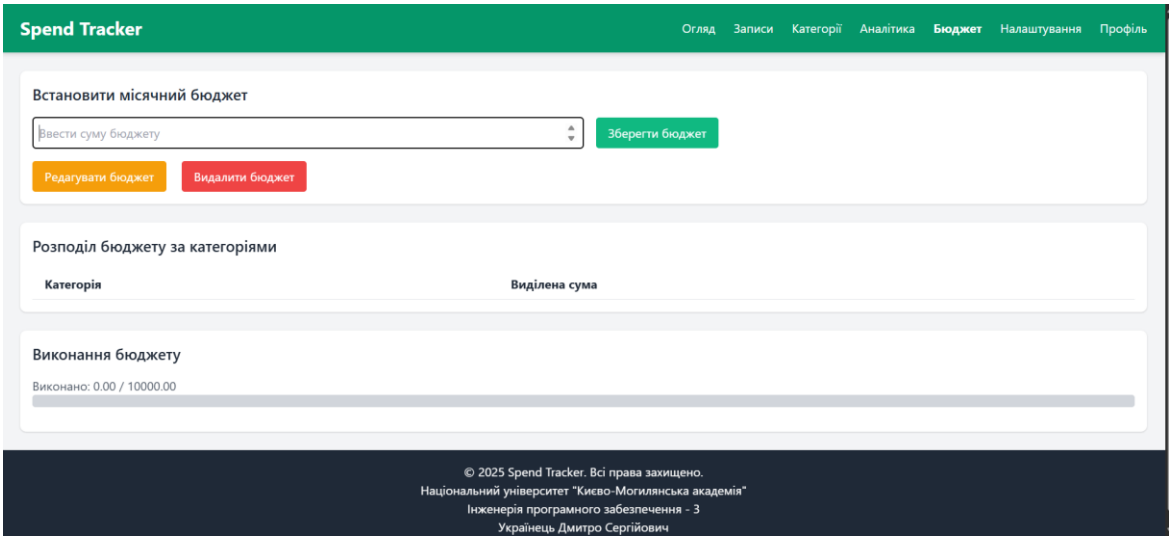

Рисунок 3.2 – Сторінка «Бюджет»

3.1.3 Аналітика

Реалізована функціональність на сторінках «Огляд» та «Аналітика» на рисунку 3.3:

– побудова секторного графіка витрат за категоріями;
– порівняння доходів і витрат по місяцях;
– порівняння аналітики витрат та доходів в межах кварталу, півріччя, року та за весь час.

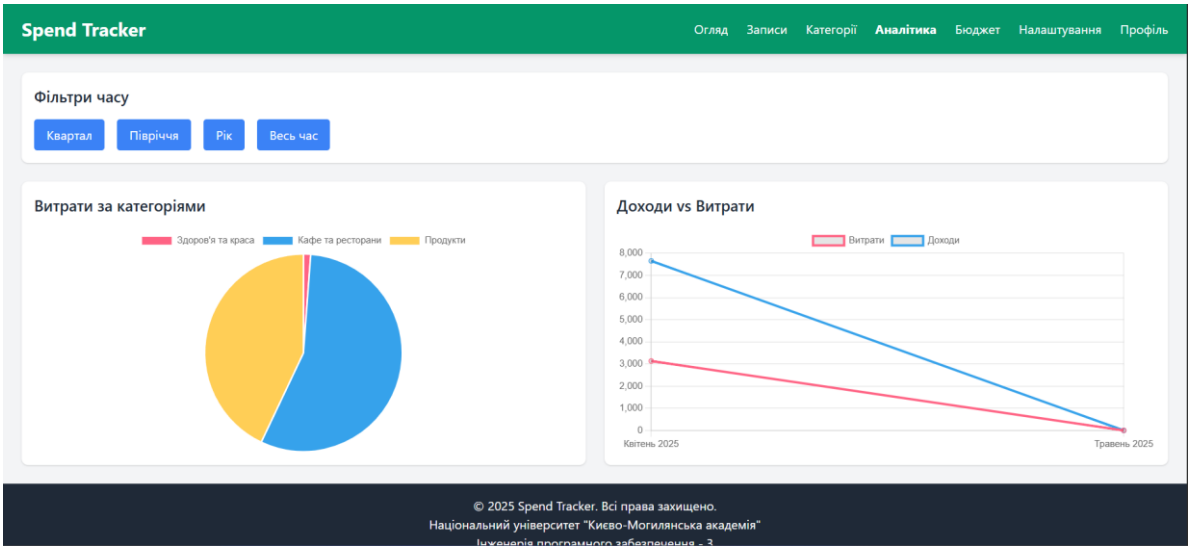

Рисунок 3.3 – Сторінка «Аналітика»

3.1.4 Сповіщення

Реалізована функціональність на сторінці «Налаштування» на рисунку 3.8:

– щоденне надсилання підсумку витрат та доходів за попередній день;
– ручне сповіщення через інтерфейс налаштувань;
– підтримка VAPID, фонової доставки повідомлень через Background Sync.

3.1.5 Офлайн-режим

Реалізована функціональність на рисунках 3.4, 3.5:

– збереження транзакцій та підписок у IndexedDB при відсутності мережі;
– подальша синхронізація з сервером при відновленні з’єднання;
– використання fallback-сторінки offline.html.

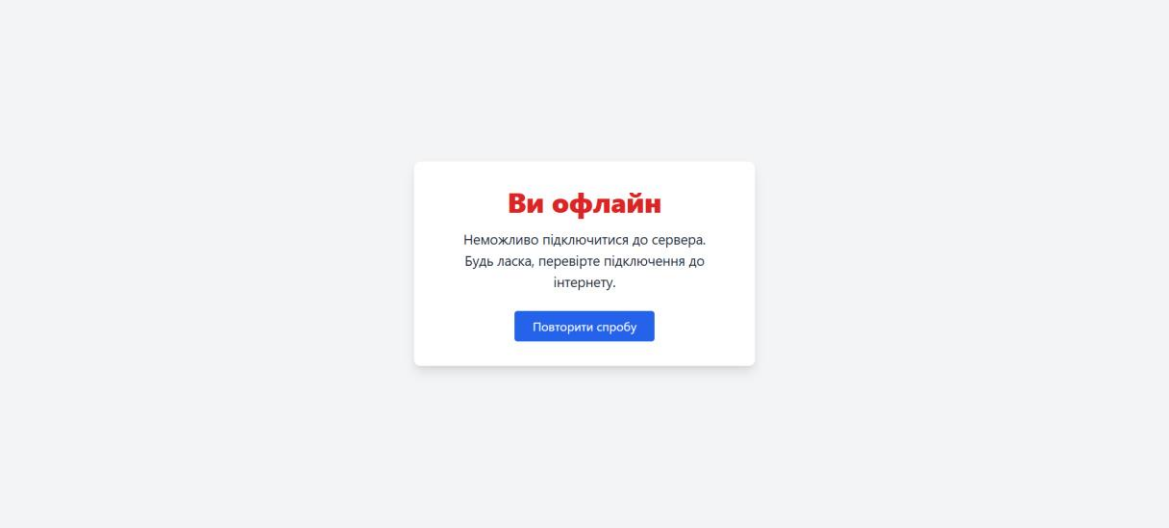

Рисунок 3.4 – Офлайн-сторінка

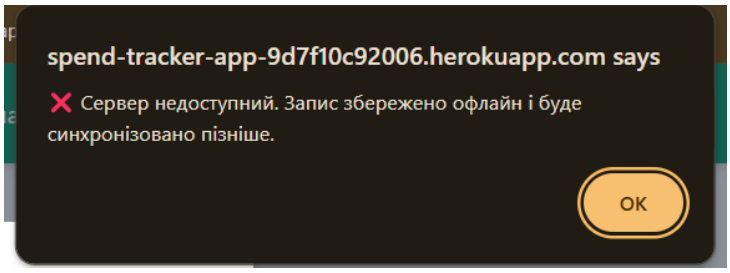

Рисунок 3.5 – Сповіщення про відсутність з’єднання з сервером

3.1.6 Авторизація та реєстрація

Реалізована функціональність на сторінках авторизації та реєстрації на рисунку 3.6:

– реєстрація нового користувача (роль USER);

– вхід з валідацією email/пароля;
– перевірка прав доступу через Spring Security та Thymeleaf.

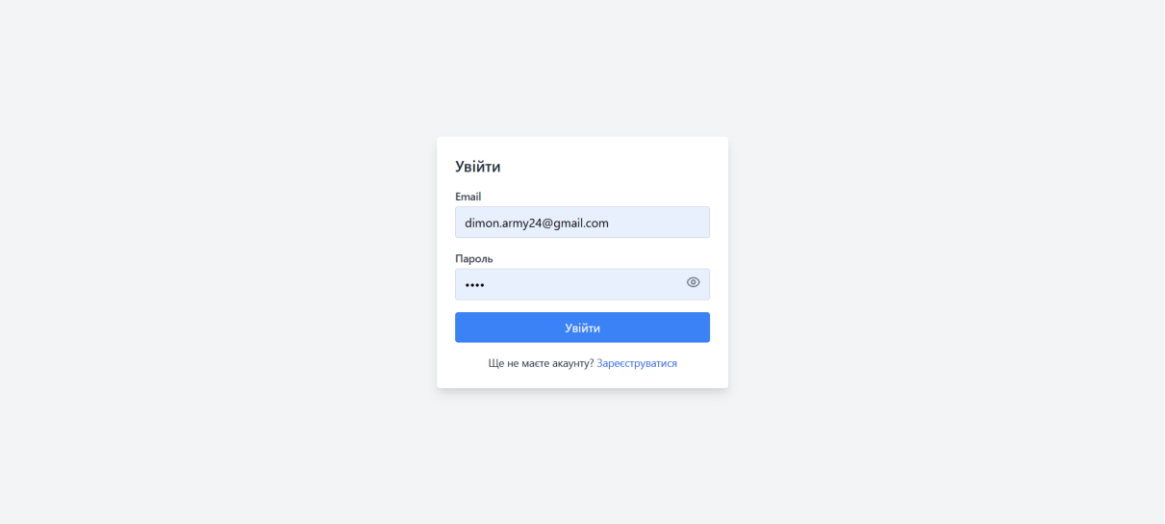

Рисунок 3.6 – Сторінка авторизації

3.1.7 Категорії

Реалізована функціональність на сторінці «Категорії» на рисунку 3.7:
– створення нової категорії з відповідним типом (дохід/витрати);
– редагування та видалення категорій;
– валідація при видаленні категорії на наявність транзакцій.

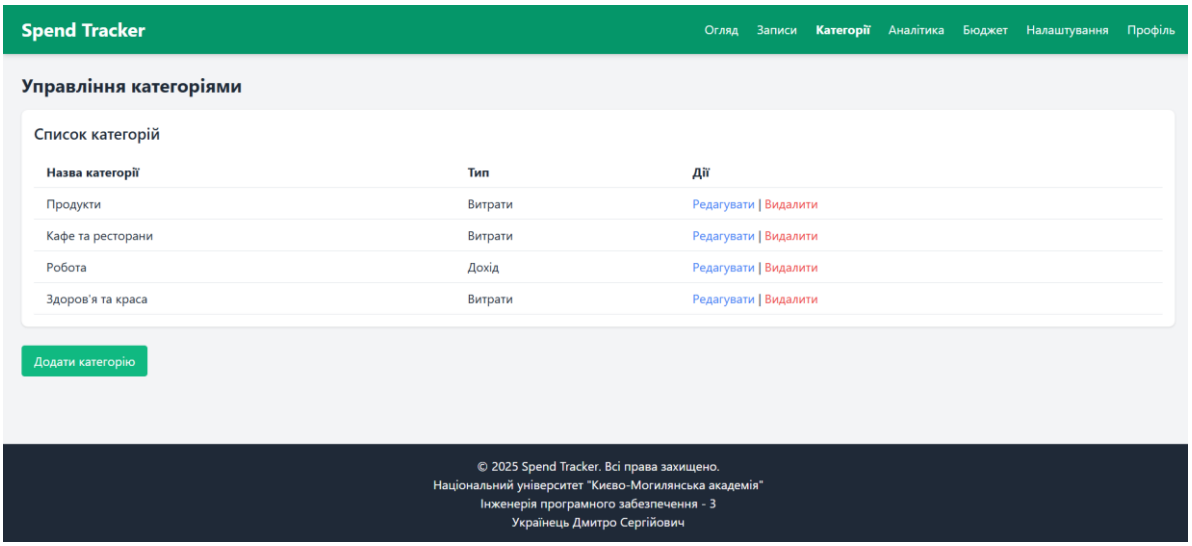

Рисунок 3.7 – Сторінка «Категорії»

3.1.8 Налаштування

Реалізована функціональність на сторінці «Налаштування» на рисунку 3.8:

– експорт транзакцій в CSV/Excel файли;
– імпорт транзакцій із CSV/Excel файлів;
– відображення статусу синхронізації з сервером.

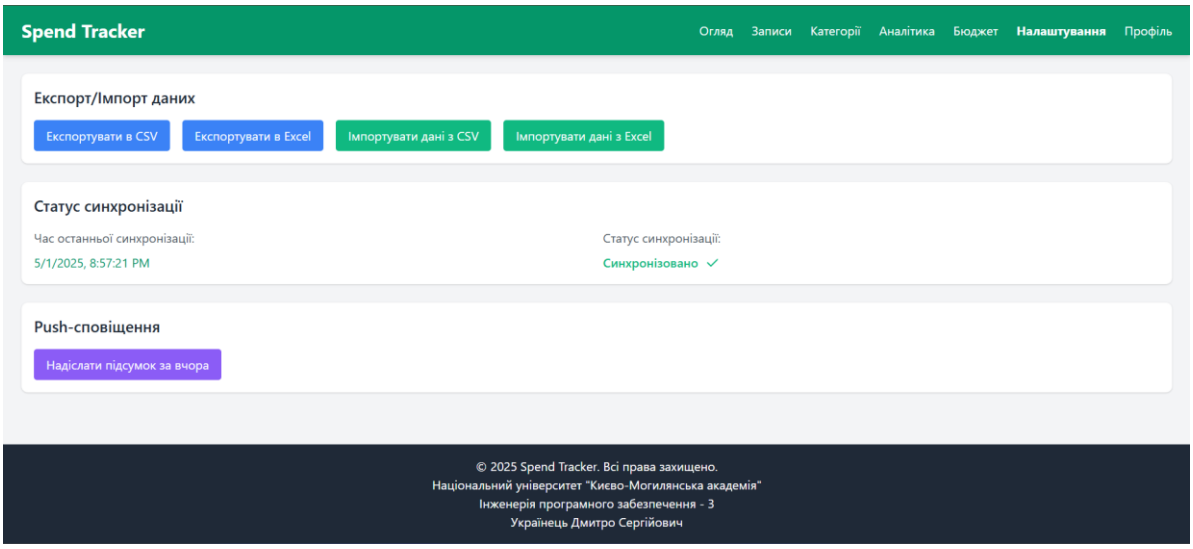

Рисунок 3.8 – Сторінка «Налаштування»

3.1.9 Користувач

Реалізована функціональність на сторінках «Профіль» та «Адмін-панель» на рисунку 3.9:

– реєстрація, авторизація користувача;
– редагування ім’я та пароля користувача;
– вихід з поточного аккаунта;
– можливість редагувати ім’я, та роль користувача (для користувача з роллю адміністратора через адмін-панель).

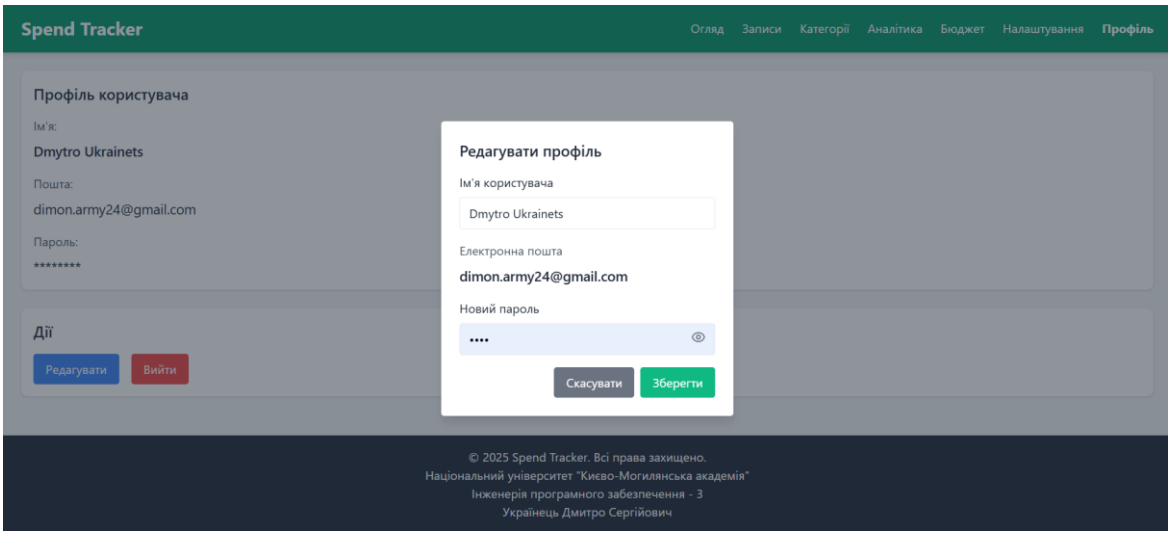

Рисунок 3.9 – Сторінка «Профіль» з модульним вікном редагування

Реалізація офлайн-режиму з використанням Service Worker

Однією з ключових характеристик PWA-застосунків є здатність працювати в умовах повної або часткової відсутності інтернет-з'єднання. Для реалізації цієї функції у застосунку Spend Tracker використовується Service Worker (додаток А) — спеціальний JavaScript-файл, який діє як проксі між браузером та мережею.

Після першого завантаження застосунку на головній сторінці браузер реєструє Service Worker за допомогою реєстратора (додаток В), як показано на рисунку 3.10, що дозволяє запускати фонові процеси, навіть якщо сторінка неактивна.

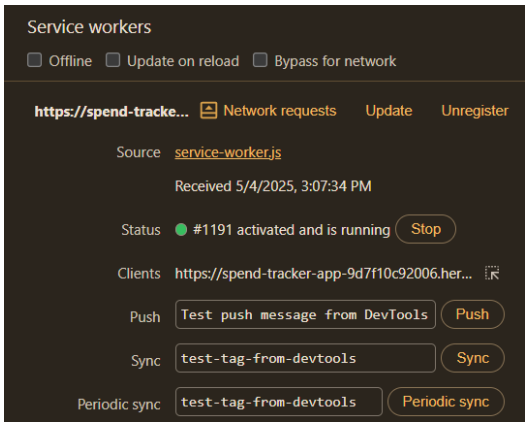

Рисунок 3.10 – Реєстрація Service Worker в браузері Google Chrome

На етапі встановлення Service Worker виконується кешування основних ресурсів, як показано в лістингу 3.1 — HTML-сторінок, стилів, скриптів, іконок, а також спеціальної офлайн-сторінки. Це дозволяє користувачеві повторно відкривати застосунок навіть без підключення до інтернету чи поганої якості мережі.

Лістинг 3.1

Кешування даних при встановленні Service Worker

```
self.addEventListener('install', event => {
    event.waitUntil(
        caches.open(CACHE_NAME).then(async cache => {
            for (const resource of PRECACHE_RESOURCES) {
                try {
                    await cache.add(resource);
                    console.log(`Cached: ${resource}`);
                } catch (err) {
                    console.error(`Failed to cache:
${resource}`, err);
                }
            }
        }).then(() => self.skipWaiting())
    );
});
```

Після реєстрації Service Worker виконує перехоплення всіх GET-запитів до мережі, як показано в лістингу 3.2. Якщо запит виконується успішно, відповідь зберігається у кеші для подальшого повторного використання. Це дозволяє забезпечити швидке завантаження даних при наступних зверненнях і знизити навантаження на мережу. У випадках, коли мережеве з’єднання відсутнє або запит завершується помилкою, Service Worker спершу намагається знайти відповідний ресурс у кеші. Якщо запит стосується навігації (тобто переходу на HTML-сторінку) і відповідь не знайдена, користувачеві повертається спеціальна офлайн-сторінка, яка пояснює причину недоступності ресурсу.

Лістинг 3.2

Перехоплення мережевих запитів у Service Worker

```
self.addEventListener('fetch', event => {
    if (event.request.method !== 'GET') return;

    event.respondWith(
        fetch(event.request)
            .then(response => {
                if (response.ok && response.type === 'basic' &&
event.request.url.startsWith('http')) {
                    const responseClone = response.clone();
                    event.waitUntil(
                        caches.open(CACHE_NAME).then(cache =>
cache.put(event.request, responseClone))
                    );
                }
                return response;
            })
            .catch(() => {
                return
caches.match(event.request).then(cachedResponse => {
                    if (cachedResponse) return cachedResponse;
                    if (event.request.mode === 'navigate') {
                        return caches.match(OFFLINE_URL);
                    }
                });
            })
    );
});
```

Щоб уникнути зберігання застарілих ресурсів, Service Worker також реалізує механізм оновлення кешу, як показано в лістингу 3.3. Під час активації нової версії видаляються старі версії кешу, які не входять до переліку актуальних. Це забезпечує, що застосунок завжди використовує найсвіжіші дані без перевантаження пам’яті пристрою.

Лістинг 3.3

Оновлення кешу під час активації нової версії Service Worker

```
self.addEventListener('activate', event => {
    event.waitUntil(
        Promise.all([
            self.clients.claim(),
            caches.keys().then(keys =>
                Promise.all(
                    keys.map(key => {
                        if (key !== CACHE_NAME) {
```

```
                                return caches.delete(key);
                            }
                        }).filter(Boolean)
                    )
                )
            ])
        );
    });
```

Офлайн-режим також включає fallback-механізм — при спробі доступу до не кешованої сторінки користувач бачить статичне повідомлення про втрату зв’язку з мережею (див. рисунок 3.5).

Реалізація локального збереження даних у IndexedDB

Для забезпечення повноцінної роботи застосунку в офлайн-режимі, важливо не лише кешувати інтерфейс та ресурси, але й тимчасово зберігати змінювані дані користувача — зокрема транзакції та підписки на сповіщення. У застосунку Spend Tracker для цього використовується браузерна база даних IndexedDB.

IndexedDB — це низькорівневе сховище у форматі «ключ-значення», яке працює асинхронно та дозволяє зберігати великі обсяги структурованих об'єктів без потреби в серверному підключенні. Застосунок створює власну базу даних у браузері користувача, як показано в лістингу 3.4, з такими таблицями — транзакції та черга push-підписок.

Лістинг 3.4

Створення власної IndexedDB у браузері користувача

```
function openLocalDB() {
    return new Promise((resolve, reject) => {
        const request = indexedDB.open('spend-tracker-db', 1);
        request.onupgradeneeded = event => {
            const db = event.target.result;
            if (!db.objectStoreNames.contains('pending-
transactions')) {
                db.createObjectStore('pending-transactions', {
autoIncrement: true });
            }
        };
        request.onsuccess = event =>
resolve(event.target.result);
        request.onerror = event => reject(event.target.error);
    });
}
```

Коли користувач створює нову транзакцію в офлайн-режимі, вона автоматично записується у локальну IndexedDB як показано на рисунку 3.11. Згодом, при відновленні підключення, Service Worker виконує синхронізацію цих даних із сервером, після чого запис у базі може бути видалений, як показано у лістингу 3.5.

Лістинг 3.5

Збереження транзакцій в IndexedDB

```
async function syncPendingTransactions() {
    const userId = await fetchCurrentUserId();
    const db = await openLocalDB();
    const tx = db.transaction('pending-transactions',
'readonly');
    const store = tx.objectStore('pending-transactions');
    const getAllRequest = store.getAll();
    getAllRequest.onsuccess = async () => {
        const records = getAllRequest.result;
        if (!records.length) {
            return;
        }
        for (const record of records) {
            try {
                const formattedRecord = {
                    amount: record.amount,
                    category: { id: record.category?.id ||
record.categoryId },
                    description: record.description ||
record.note,
                    date: record.date,
                    user: { id: record.user?.id || record.userId
}
                };
                const response = await
fetch('/api/transactions', {
                    method: 'POST',
                    headers: { 'Content-Type':
'application/json' },
                    body: JSON.stringify(formattedRecord)
                });
                const responseText = await response.text();
                if (!response.ok) {
                    console.error(`Сервер повернув статус
${response.status}: ${responseText}`);
                    throw new Error(`Сервер повернув статус
${response.status}`);
                }
            } catch (err) {
                console.error('Не вдалося синхронізувати
транзакцію:', err);
                return;
            }
        }
        const clearTx = db.transaction('pending-transactions',
'readwrite');
        clearTx.objectStore('pending-transactions').clear();
    };
    getAllRequest.onerror = (event) => {
```

```
        console.error('Помилка читання IndexedDB при sync:',
event.target.error);
    };
}
```

Завдяки цьому механізму:

а) Користувач не втрачає введені дані при втраті з'єднання.

б) Застосунок може працювати швидко та автономно без постійного звернення до API.

в) Зберігається послідовність дій у черзі синхронізації.

Окрім транзакцій, у IndexedDB також зберігаються дані push-підписок у випадках, коли не вдалося одразу надіслати їх на сервер. Це дозволяє зберегти підписку і надіслати її пізніше за допомогою Background Sync.

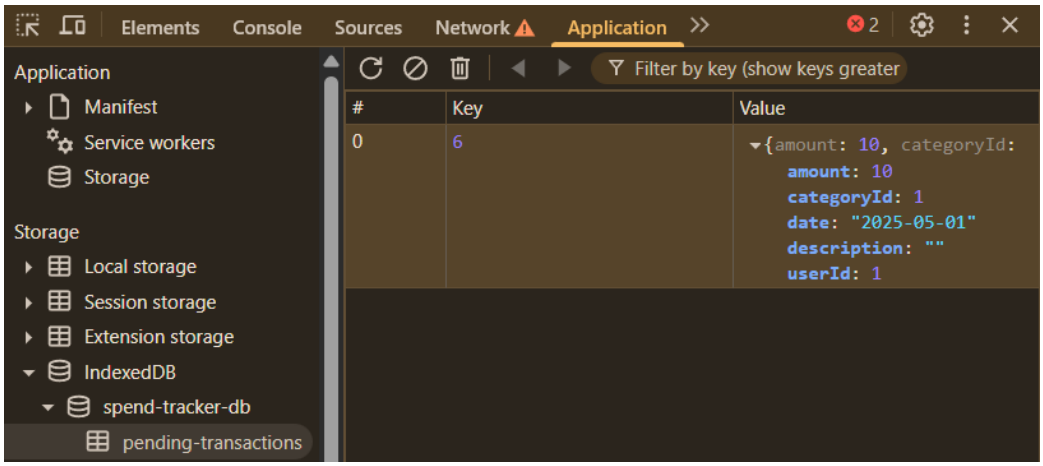

Рисунок 3.11 – Локальна IndexedDB із транзакціями

Push-сповіщення та фонова синхронізація

Одним із важливих елементів застосунку Spend Tracker є система push-сповіщень (рисунок 3.12), яка інформує користувача про підсумок витрат і доходів за попередній день. Повідомлення надсилаються навіть тоді, коли застосунок неактивний (лістинг 3.6 та 3.7), що дозволяє зберігати залученість користувача.

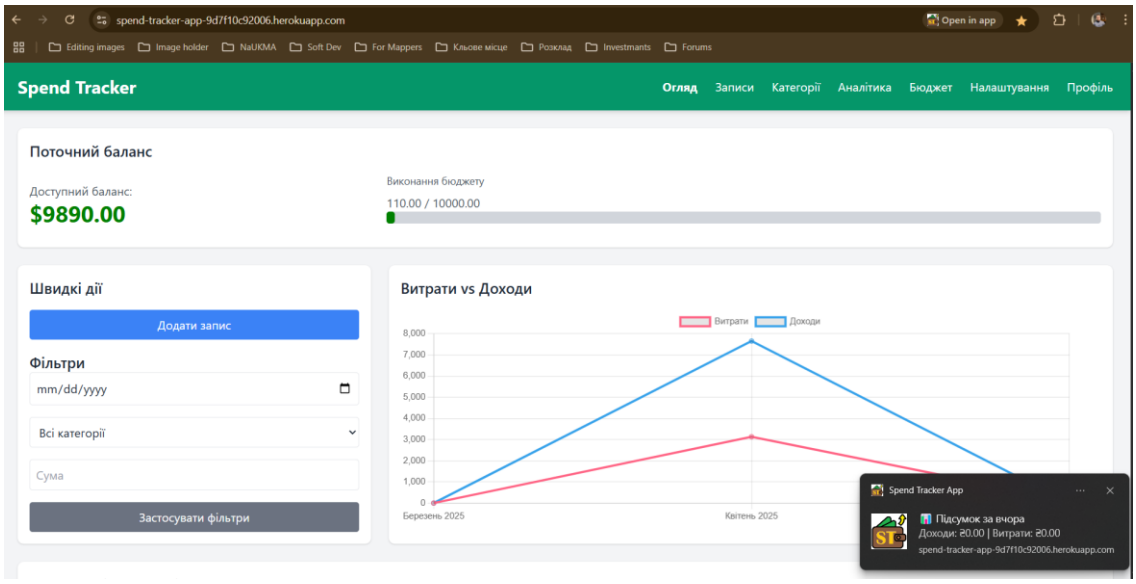

Рисунок 3.12 – Push-сповіщення про доходи та витрати за минулий день

Лістинг 3.6

Надсилання push-сповіщень

```
self.addEventListener('push', event => {
    let data = event.data?.json() || {};
    const title = data.title || 'Spend Tracker';
    const options = {
        body: data.body,
        icon: '/icons/icon-192x192.png',
        badge: '/icons/icon-192x192.png',
        data: data.url || '/'
    };
    event.waitUntil(
        self.registration.showNotification(title, options)
    );
});
```

Лістинг 3.7

Синхронізація та надсилання push-сповіщень

```
self.addEventListener('sync', event => {
    if (event.tag === 'sync-transactions') {
        event.waitUntil(syncPendingTransactions());
    } else if (event.tag === 'sync-push-summary') {
        event.waitUntil(sendPushSummaryLater());
    }
});
```

Реєстрація підписки на сповіщення

Після дозволу користувача браузер генерує спеціальний об'єкт підписки, який містить endpoint та криптографічні ключі. Ці дані зберігаються у локальній базі IndexedDB і надсилаються на сервер для подальшого збереження, як показано в лістингу 3.8. Якщо з’єднання відсутнє — підписка зберігається локально і буде надіслана пізніше за допомогою Background Sync.

Лістинг 3.8

Реєстрація підписки

```
            try {
                await fetch('/api/push/subscribe', {
                    method: 'POST',
                    headers: { 'Content-Type':
'application/json' },
                    body: JSON.stringify(subscription)
                });
                console.log('Підписку на push оновлено:',
subscription);
            } catch (err) {
                console.warn('Не вдалося надіслати підписку —
зберігаємо локально');
                await saveSubscriptionOffline(subscription);
                await registration.sync.register('sync-push-
subscription');
            }
```

Збереження підписки на сервері

Після отримання підписки серверна частина застосунку перевіряє користувача, формує об’єкт підписки та зберігає його в базу даних у зв’язку з відповідним обліковим записом користувача. Усі підписки мають унікальні ключі (endpoint, auth, p256dh) і зберігаються з позначенням часу створення.

Надсилання повідомлень із сервера

Для формування push-сповіщень (див. рисунок 3.12) сервер використовує бібліотеку web-push та VAPID-ключі. Щодня планувальник формує коротке повідомлення з підсумком витрат та доходів за вчорашній день і надсилає його усім підписаним користувачам. Якщо користувач має кілька пристроїв — повідомлення надсилаються на кожен.

Обробка сповіщень у браузері

Service Worker на боці клієнта приймає повідомлення, розшифровує його та виводить сповіщення на екран користувача. При натисканні на нього відкривається вікно застосунку з відповідною сторінкою. Користувач також має можливість надіслати сповіщення вручну через сторінку налаштувань.

Фонова синхронізація транзакцій та підписок

Якщо транзакцію або push-підписку не вдалося відправити на сервер у момент створення, вона зберігається в IndexedDB та ставиться в чергу для синхронізації. Після відновлення з’єднання Service Worker перехоплює подію sync і автоматично виконує передачу накопичених даних на сервер. Таким чином, користувачу не потрібно вручну повторювати дії.

Цей механізм забезпечує:

а) Гарантовану доставку важливих змін.

б) Надійність навіть у нестабільних мережах.

в) Асинхронну обробку даних без втрат.

4 Тестування та оцінка результатів

Тестування функціональності

Після завершення реалізації PWA-застосунку Spend Tracker було проведено тестування на відповідність ключовим вимогам до PWA-застосунків. Тестування охоплювало встановлення на пристрої, роботу в умовах відсутності інтернет-з'єднання та надсилання push-сповіщень у різних браузерах.

Установка на пристрої

Застосунок було протестовано на десктопному браузері Google Chrome та інших браузерах за замовчуванням на операційній системі Windows. У всіх випадках браузер розпізнає PWA та пропонує встановити застосунок на головний екран пристрою. Після встановлення запуск відбувається у відокремленому вікні без адресного рядка, що відповідає поведінці нативного застосунку. Іконка застосунку автоматично додається до головного екрана.

Перевірка роботи офлайн

Функціональність офлайн-режиму перевірялась шляхом повного відключення інтернет-з’єднання після початкового завантаження застосунку. Застосунок продовжує відображати кешовані сторінки, дозволяє додавати транзакції, які зберігаються в IndexedDB. При відновленні мережі транзакції синхронізуються з сервером автоматично або після оновлення сторінки.

Робота push-сповіщень у різних браузерах

Push-сповіщення було протестовано в браузерах Chrome, Firefox та Edge. У всіх цих браузерах функціональність працює коректно: користувач отримує сповіщення навіть при закритому застосунку, транзакції зберігаються офлайн тощо.

Оцінка зручності користування

Після технічного тестування функціональних можливостей було здійснено оцінку зручності користування застосунком Spend Tracker. Цей аспект є важливим для визначення того, наскільки інтерфейс інтуїтивно зрозумілий, доступний для користувача та адаптивний під різні пристрої.

Адаптивність

Інтерфейс застосунку було оптимізовано для використання на комп’ютерах та ноутбуках. Завдяки використанню Tailwind CSS і класів адаптивної верстки всі елементи інтерфейсу коректно розташовуються на сторінках. Поєднання стилізації, кольорової гами та шрифтів є досить приємним. Однак поки що застосунок не адаптований під використання на мобільних пристроях та планшетах. Цю функціональність слід покращити та доопрацювати в подальших версія застосунку.

4.2.2 UX-аналітика

У процесі власного тестування було виявлено, що користувацький досвід загалом є позитивним. Найбільш зручними виявилися такі рішення:

– форма додавання транзакцій розташована одразу на головній сторінці;
– секція аналітики дозволяє швидко та всебічно оцінити витрати по категоріях та за певний проміжок часу;

– push-сповіщення підвищують залученість без потреби постійно відкривати застосунок.

До покращень можна віднести:

– реалізацію реєстрації за допомогою сторонніх сервісів;
– можливість створювати спільні бюджети;
– встановлення бюджету на основі записів про доходи.

Оцінка зручності користування дозволяє підтвердити, що застосунок відповідає більшості очікуванням щодо PWA у частині доступності, простоти взаємодії та швидкого реагування на дії користувача.

Порівняння з подібними рішеннями

Для повноцінної оцінки розробленого застосунку Spend Tracker було проведено аналіз подібних рішень, зокрема таких як Google Keep і фінансовий модуль мобільного застосунку Monobank:

а) Google Keep — це хмарний сервіс для створення нотаток, який також підтримує офлайн-доступ, синхронізацію між пристроями та PWA-функціональність. Проте він не має спеціалізованих можливостей для обліку фінансів. Перевагою є висока швидкість роботи та гнучкість використання на будь-якому пристрої.

б) Monobank — популярний банківський застосунок із вбудованою системою аналітики витрат. Він має візуалізацію по категоріях, push-сповіщення, автоматичне визначення типу транзакції. Його перевага — інтеграція з реальними банківськими рахунками, однак обмеженням є те, що користувач не може створювати власні категорії або вводити дані вручну для персонального бюджету, який не пов'язаний з банком.

Сильні сторони застосунку Spend Tracker:

а) Повна автономність — дані не залежать від банку, зберігаються на сервері або локально;

б) Підтримка офлайн-режиму — транзакції можна додавати без мережі;

в) Гнучкість введення — користувач сам визначає категорії, типи та суми;

г) Інтерфейс без перевантаження функціями — фокус лише на доходах і витратах;

ґ) PWA-переваги — можна встановити як застосунок без Google Play чи App Store.

Обмеження у порівнянні з іншими

а) Відсутність автоматичної інтеграції з банківськими рахунками;

б) Менша глибина аналітики порівняно з банківськими застосунками;

в) Сповіщення доступні не на всіх платформах (наприклад, обмеження в iOS).

г) Відсутня реєстрація та авторизація за допомогою сторонніх сервісів таких як Google, Facebook, GitHub тощо ;

ґ) Відсутня адаптивна версія застосунку під мобільні пристрої та планшети.

Загалом застосунок показав хорошу технологічну завершеність та переваги в автономності й гнучкості, які можуть бути недоступні у складніших комерційних рішеннях.

Висновки

У ході виконання курсової роботи було досліджено ключові аспекти створення веб-застосунків на основі технології Progressive Web Applications, проведено аналіз їх архітектури, функціональних можливостей, безпекових вимог і практичного застосування.

У першому розділі було охарактеризовано концепцію PWA, розкрито її історію, основні характеристики, переваги над традиційними веб- та нативними застосунками, а також ключові API, що забезпечують прогресивну поведінку. Також було встановлено, що PWA дозволяє поєднувати переваги браузерної доступності з функціональністю мобільних додатків, забезпечуючи високу швидкість, автономність та зручність.

У другому розділі обґрунтовано вибір архітектури застосунку Spend Tracker, який реалізовано у вигляді клієнт-серверної системи з серверним рендерингом через шаблонізатор Thymeleaf. Проаналізовано інструменти клієнтської частини (Tailwind CSS, IndexedDB, Service Worker тощо) та серверної частини (Spring Boot та MySQL), а також засоби захисту, включно з використанням HTTPS, Spring Security, контроль доступу та політику CSP.

У третьому розділі описано реалізацію застосунку: роботу з транзакціями, бюджетом, аналітикою, push-сповіщеннями, offline-функціональністю, авторизацією тощо. Описано механізми кешування, локального збереження, фонової синхронізації, а також системи надсилання push-повідомлень із бекенду з використанням VAPID-ключів.

У четвертому розділі проведено тестування застосунку за ключовими критеріями PWA, оцінено його адаптивність, зручність користування та переваги порівняно з подібними рішеннями. Встановлено, що застосунок коректно працює в офлайн-режимі, підтримує встановлення на пристрої та забезпечує сповіщення через push.

Результати теоретичної частини курсової роботи свідчать про доцільність використання технології PWA для створення сучасних,

автономних і зручних веб-застосунків. Запропонована архітектура забезпечує гнучкість, масштабованість і простоту підтримки. У подальшому можливим напрямом розвитку може бути інтеграція з зовнішніми сервісами (наприклад, банківськими API), розширення можливостей аналітики, оптимізація для використання на екранах різного розміру, можливість авторизації та реєстрації за допомогою сторонніх сервісів, а також використання спільних бюджетів.

Таким чином, було створено концептуальну й технологічну основу для реалізації курсової роботи, в якій було поглиблено питання ефективності реалізації, продуктивності та стійкості застосунку.

Список використаної літератури

1. Progressive Web Apps vs Regular Websites [Електронний ресурс] // Magefan. – Режим доступу: https://magefan.com/ua/blog/progressive-web-apps-vs-regular-websites.
2. Progressive web apps [Електронний ресурс] // MDN Web Docs. – Режим доступу: https://developer.mozilla.org/en-US/docs/Web/Progressive_web_apps.
3. Explore Progressive Web Apps [Електронний ресурс] // Web.dev. – Режим доступу: https://web.dev/explore/progressive-web-apps.
4. Learn PWA [Електронний ресурс] // Web.dev. – Режим доступу: https://web.dev/learn/pwa.
5. Що таке PWA-додатки: розробка Progressive Web Application [Електронний ресурс] // DAN.IT. – Режим доступу: https://dan-it.com.ua/uk/blog/shho-take-pwa-dodatky-rozrobka-progressive-web-application/.
6. Progressive Web Apps [Електронний ресурс] // Next.js Documentation. – Режим доступу: https://nextjs.org/docs/app/guides/progressive-web-apps.
7. What Are Progressive Web Apps [Електронний ресурс] // Ionic Framework. – Режим доступу: https://ionicframework.com/docs/core-concepts/what-are-progressive-web-apps.
8. Вступ до PWA [Електронний ресурс] // Smile-Ukraine. – Режим доступу: https://smile-ukraine.com/ua/pwa/introduction.
9. PWA-додатки: можливості і переваги [Електронний ресурс] // Wezom. – Режим доступу: https://wezom.com.ua/ua/blog/pwa-prilozheniya-web-progressive-app.
10. Progressive Web Apps in DevTools [Електронний ресурс] // Chrome Developers. – Режим доступу: https://developer.chrome.com/docs/devtools/progressive-web-apps.

11. The Rise of Progressive Web Apps: Revolutionising the Mobile Experience [Електронний ресурс] // Koda web. – Режим доступу: https://www.kodaweb.co.nz/blog/rise-progressive-web-apps-revolutionising-mobile-experience
12. Атаки типу Man-In-The-Middle: що треба знати кожному [Електронний ресурс] // Imena.ua. – Режим доступу: https://www.imena.ua/blog/man-in-the-middle/
13. How To Build Progressive Web App: Best Examples & Tools For PWA [Електронний ресурс] // Agile Tech. – Режим доступу: https://agiletech.vn/build-progressive-web-app-examples-frameworks/
14. The next billion users: trivago embrace progressive web apps as the future of mobile [Електронний ресурс] // Think with Google. – Режим доступу: https://www.thinkwithgoogle.com/intl/en-emea/marketing-strategies/app-and-mobile/trivago-embrace-progressive-web-apps-as-the-future-of-mobile/
15. Twitter Lite PWA Significantly Increases Engagement and Reduces Data Usage [Електронний ресурс] // Web.dev. – Режим доступу: https://web.dev/case-studies/twitter
16. Progressive Enhancement [Електронний ресурс] // MDN Web Docs. – Режим доступу: https://developer.mozilla.org/en-US/docs/Glossary/Progressive_Enhancement

Додаток А

(обов’язковий)

Service Worker застосунку «Spend Tracker»

```
const CACHE_NAME = 'spend-tracker-cache-v2';
const OFFLINE_URL = '/offline.html';
const PRECACHE_RESOURCES = [
  '/',
  '/manifest.json',
  '/offline.html',
  '/login',
  '/register',
  '/main',
  '/records',
  '/categories',
  '/analytics',
  '/budget',
  '/settings',
  '/profile',
  '/admin',
  '/service-worker.js',
  '/assets/js/login.js',
  '/assets/js/main-chart.js',
  '/assets/js/main-budget.js',
  '/assets/js/main-records.js',
  '/assets/js/admin.js',
  '/assets/js/records.js',
  '/assets/js/settings.js',
  '/assets/js/profile.js',
  '/assets/js/categories.js',
  '/assets/js/analytics.js',
```

```
    '/assets/js/sw-register.js',
    '/assets/js/budget.js',
    '/assets/js/syncStatus.js',
    'https://cdn.jsdelivr.net/npm/tailwindcss@2.2.19/dist/tailwind.min.css',
    '/icons/icon-192x192.png',
    '/icons/icon-512x512.png',
    '/assets/favicon.ico'
];
self.addEventListener('install', event => {
    event.waitUntil(
        caches.open(CACHE_NAME).then(async cache => {
            for (const resource of PRECACHE_RESOURCES) {
                try {
                    await cache.add(resource);
                    console.log(`Cached: ${resource}`);
                } catch (err) {
                    console.error(`Failed to cache: ${resource}`, err);
                }
            }
        }).then(() => self.skipWaiting())
    );
});
self.addEventListener('activate', event => {
    event.waitUntil(
        Promise.all([
            self.clients.claim(),
            caches.keys().then(keys =>
                Promise.all(
                    keys.map(key => {
                        if (key !== CACHE_NAME) {
```

```
                    return caches.delete(key);
                }
            }).filter(Boolean)
        )
    )
  ])
);
});
self.addEventListener('fetch', event => {
  if (event.request.method !== 'GET') return;
  event.respondWith(
    fetch(event.request)
      .then(response => {
        if (response.ok && response.type === 'basic' &&
event.request.url.startsWith('http')) {
          const responseClone = response.clone();
          event.waitUntil(
            caches.open(CACHE_NAME).then(cache =>
cache.put(event.request, responseClone))
          );
        }
        return response;
      })
      .catch(() => {
        return caches.match(event.request).then(cachedResponse => {
          if (cachedResponse) return cachedResponse;
          if (event.request.mode === 'navigate') {
            return caches.match(OFFLINE_URL);
          }
        });
```

```
            })
    );
});
self.addEventListener('push', event => {
    let data = event.data?.json() || {};
    const title = data.title || 'Spend Tracker';
    const options = {
        body: data.body,
        icon: '/icons/icon-192x192.png',
        badge: '/icons/icon-192x192.png',
        data: data.url || '/'
    };
    event.waitUntil(
        self.registration.showNotification(title, options)
    );
});
self.addEventListener('notificationclick', event => {
    event.notification.close();
    event.waitUntil(
        clients.openWindow(event.notification.data)
    );
});
self.addEventListener('sync', event => {
    if (event.tag === 'sync-transactions') {
        event.waitUntil(syncPendingTransactions());
    } else if (event.tag === 'sync-push-summary') {
        event.waitUntil(sendPushSummaryLater());
    }
```

```
});
async function sendPushSummaryLater() {
    try {
        await fetch('/api/push/send', { method: 'POST' });
        console.log('Сповіщення надіслано пізніше через sync');
    } catch (err) {
        console.error('Не вдалося надіслати навіть через sync', err);
    }
}
async function syncPendingTransactions() {
    console.log('Синхронізація транзакцій з IndexedDB...');
    const userId = await fetchCurrentUserId();
    const db = await openLocalDB();
    const tx = db.transaction('pending-transactions', 'readonly');
    const store = tx.objectStore('pending-transactions');
    const getAllRequest = store.getAll();
    getAllRequest.onsuccess = async () => {
        const records = getAllRequest.result;
        if (!records.length) {
            console.log('Немає транзакцій для синхронізації');
            return;
        }
        console.log(`Знайдено ${records.length} транзакцій для надсилання...`);
        for (const record of records) {
            try {
                const formattedRecord = {
                    amount: record.amount,
                    category: { id: record.category?.id || record.categoryId },
```

```
            description: record.description || record.note,
            date: record.date,
            user: { id: record.user?.id || record.userId }
          };
          console.log('Формуємо транзакцію для синхронізації:', record);
          console.log('Відправляємо на сервер:', formattedRecord);
          const response = await fetch('/api/transactions', {
            method: 'POST',
            headers: { 'Content-Type': 'application/json' },
            body: JSON.stringify(formattedRecord)
          });
          const responseText = await response.text();
          if (!response.ok) {
            console.error(`Сервер повернув статус ${response.status}:
${responseText}`);
            throw new Error(`Сервер повернув статус ${response.status}`);
          }
          console.log('Надіслано транзакцію:', formattedRecord);
        } catch (err) {
          console.error('Не вдалося синхронізувати транзакцію:', err);
          return;
        }
      }
      const clearTx = db.transaction('pending-transactions', 'readwrite');
      clearTx.objectStore('pending-transactions').clear();
      console.log('Очищено локальні транзакції після синхронізації');
    };
    getAllRequest.onerror = (event) => {
```

```
        console.error('Помилка читання IndexedDB при sync:', event.target.error);
    };
}
function openLocalDB() {
    return new Promise((resolve, reject) => {
        const request = indexedDB.open('spend-tracker-db', 1);
        request.onupgradeneeded = event => {
            const db = event.target.result;
            if (!db.objectStoreNames.contains('pending-transactions')) {
                db.createObjectStore('pending-transactions', { autoIncrement: true });
            }
        };
        request.onsuccess = event => resolve(event.target.result);
        request.onerror = event => reject(event.target.error);
    });
}
async function fetchCurrentUserId() {
    try {
        const resp = await fetch('/api/users/me/id');
        if (!resp.ok) throw new Error('Помилка отримання userId');
        return await resp.json();
    } catch (err) {
        console.error('Неможливо отримати userId для синхронізації:', err);
        throw err;
    }
}
```

Додаток Б

(обов’язковий)

Web App Manifest застосунку «Spend Tracker»

```
{
  "id": "/",
  "name": "Spend Tracker App",
  "short_name": "Spend Tracker",
  "start_url": "/",
  "display": "standalone",
  "background_color": "#ffffff",
  "theme_color": "#6200ea",
  "description": "PWA для керування витратами",
  "icons": [
    {
      "src": "/icons/icon-192x192.png",
      "sizes": "192x192",
      "type": "image/png"
    },
    {
      "src": "/icons/icon-512x512.png",
      "sizes": "512x512",
      "type": "image/png"
    }
  ]
}
```

Додаток В

(обов’язковий)

Service Worker Register застосунку «Spend Tracker»

```
const vapidPublicKey = 'BMEH2H4-gHG-te-
a2hWQBBcRW463gJ8QHME6lly1HLwmSrR0eUKHbKhX2sAVnG9D3uQmVN
AVRoLt469P2MJL2-I';

if ('serviceWorker' in navigator && 'PushManager' in window && 'SyncManager'
in window) {
  window.addEventListener('load', async () => {
    try {
      const registration = await navigator.serviceWorker.register('/service-
worker.js');
      console.log('Service Worker зареєстрований:', registration);

      let subscription = await registration.pushManager.getSubscription();
      if (!subscription) {
        subscription = await registration.pushManager.subscribe({
          userVisibleOnly: true,
          applicationServerKey: urlBase64ToUint8Array(vapidPublicKey)
        });
      }

      try {
        await fetch('/api/push/subscribe', {
          method: 'POST',
          headers: { 'Content-Type': 'application/json' },
          body: JSON.stringify(subscription)
        });
        console.log('Підписку на push оновлено:', subscription);
```

```
        } catch (err) {
            console.warn('Не вдалося надіслати підписку — зберігаємо
локально');
            await saveSubscriptionOffline(subscription);
            await registration.sync.register('sync-push-subscription');
        }

    } catch (err) {
        console.error('Помилка реєстрації SW або підписки:', err);
    }
  });
}


function urlBase64ToUint8Array(base64String) {
  const padding = '='.repeat((4 - base64String.length % 4) % 4);
  const rawData = (base64String + padding).replace(/\-/g, '+').replace(/_/g, '/');
  return Uint8Array.from(window.atob(rawData), c => c.charCodeAt(0));
}
```

Додаток Г

(обов’язковий)

Робота з IndexedDB в застосунку «Spend Tracker»

```
window.openDB = async function () {
  return new Promise((resolve, reject) => {
    const request = indexedDB.open('spend-tracker-db', 1);
    request.onupgradeneeded = (event) => {
      const db = event.target.result;
      if (!db.objectStoreNames.contains('pending-transactions')) {
        db.createObjectStore('pending-transactions', { autoIncrement: true });
      }
    };
    request.onsuccess = (event) => resolve(event.target.result);
    request.onerror = (event) => reject(event.target.error);
  });
};

window.saveTransactionOffline = async function (txData) {
  console.log('Викликано saveTransactionOffline:', txData);
  const db = await window.openDB();
  return new Promise((resolve, reject) => {
    const tx = db.transaction('pending-transactions', 'readwrite');
    const store = tx.objectStore('pending-transactions');
    const request = store.add(txData);

    request.onsuccess = () => {
      console.log('Запис додано до IndexedDB');
    };

    request.onerror = (event) => {
```

```
            console.error('Помилка збереження в IndexedDB:', event.target.error);
            reject(event.target.error);
        };

        tx.oncomplete = () => {
            console.log('Транзакція завершена');
            resolve();
        };

        tx.onerror = (event) => {
            console.error('Помилка транзакції IndexedDB:', event.target.error);
            reject(event.target.error);
        };
    });
};

window.getAllPendingTransactions = async function () {
    const db = await openDB();
    const tx = db.transaction('pending-transactions', 'readonly');
    const store = tx.objectStore('pending-transactions');
    return store.getAll ? store.getAll() : new Promise((resolve, reject) => {
        const result = [];
        const cursorRequest = store.openCursor();
        cursorRequest.onsuccess = () => {
            const cursor = cursorRequest.result;
            if (cursor) {
                result.push(cursor.value);
                cursor.continue();
            } else {
                resolve(result);
```

```
            }
        };
        cursorRequest.onerror = reject;
    });
}

window.clearPendingTransactions = async function () {
    const db = await openDB();
    const tx = db.transaction('pending-transactions', 'readwrite');
    tx.objectStore('pending-transactions').clear();
    return tx.complete;
}
```

Додаток Д

(обов’язковий)

Фрагмент конфігурації безпеки застосунку «Spend Tracker»

```
@Bean
   public SecurityFilterChain securityFilterChain(HttpSecurity http) throws
Exception {http.csrf().disable()
            .authorizeHttpRequests(auth -> auth
                  .requestMatchers("/login", "/register",
                        "/css/**", "/js/**", "/assets/**",
                        "/manifest.json", "/service-worker.js", "/icons/**",
"/offline.html").permitAll()
                  .requestMatchers("/admin/**").hasRole("ADMIN")
                  .requestMatchers("/api/**").authenticated()
                  .anyRequest().authenticated())
            .formLogin(form -> form
                  .loginPage("/login")
                  .usernameParameter("email")
                  .passwordParameter("password")
                  .defaultSuccessUrl("/", true)
                  .successHandler(authSuccessHandler)
                  .failureHandler(customAuthenticationFailureHandler)
                  .permitAll())
            .logout(logout -> logout
                  .logoutUrl("/logout")
                  .logoutSuccessUrl("/login?logout")
                  .permitAll()
.headers(headers -> headers7.contentSecurityPolicy("default-src 'self'; script-src
'self' https://cdn.jsdelivr.net; style-src 'self' https://cdn.jsdelivr.net"))
            .userDetailsService(customUserDetailsService);
      return http.build();}
```