Міністерство освіти і науки України

Національний університет «Києво-Могилянська академія»

Факультет інформатики

Кафедра мультимедійних систем

Курсова робота

на тему: **«СТВОРЕННЯ ВЕБ-ЗАСТОСУНКУ НА ОСНОВІ ТЕХНОЛОГІЇ PROGRESSIVE WEB APP»**

Виконала студентка 3-го року навчання,

Спеціальності 121 Інженерія програмного забезпечення

Ветрикуш Дар’я Андріївна

Керівник Борозенний С.О.,

ст. викладач

Кваліфікаційна робота захищена з оцінкою _____

Секретар ЕК _____

«___» _____2025 р.

Київ-2025

Календарний план виконання роботи

№	Назва етапу	Термін виконання
1	Отримання завдання на курсову роботу	04.10.2024
2	Огляд технічної літератури за темою роботи	05.02.2025
3	Проектування бази даних та архітектури веб-застосунку	10.02.2025
4	Розробка веб-застосунку	14.04.2025
5	Написання текстової частини курсової роботи	30.04.2025
6	Створення презентації для доповіді	03.05.2025

Ветрикуш Д. А. _____

Борозенний С. О. _____

«____»_____

Зміст

Анотація

У курсовій роботі розглянуто процес створення сучасного веб-застосунку кулінарної тематики з використанням технології PWA (Progressive Web App). Розроблений застосунок надає користувачам можливість реєстрації, публікації рецептів, перегляду публікацій інших користувачів, а також взаємодії з контентом у вигляді лайків, коментарів та підписок.

Особливу увагу приділено реалізації ключових функцій PWA. Застосунок підтримує офлайн-доступ до основної частини функціоналу. Реалізовано фонову синхронізацію дій користувача, таких як лайки, коментарі та підписки за рахунок автоматичної обробки одразу після відновлення Інтернет-з’єднання. Інтегровано push-сповіщення, які інформують користувача про нові взаємодії з його контентом. Крім того, застосунок може бути встановлений на пристрій, що максимально наближує користувацький досвід до нативних додатків.

Результатом роботи є зручний та інтуїтивно зрозумілий веб-застосунок, який демонструє переваги PWA як гібридного рішення між класичними веб-сайтами та мобільними застосунками.

Вступ

В умовах стрімкого розвитку цифрових технологій та зростання вимог користувачів до зручності та функціональності онлайн-сервісів в будь-яких умовах, перед розробниками постає завдання забезпечення якісного та уніфікованого користувацького досвіду на всіх платформах та пристроях. Традиційний підхід, який часто передбачає розробку як веб-версії, так і окремих додатків для кожної операційної системи, є ресурсоємним та потребує значних витрат часу та коштів. У цьому контексті з розвитком веб-технологій особливої актуальності набуває технологія Progressive Web App (PWA), яка пропонує ефективну альтернативу, поєднуючи гнучкість веб-платформи з функціональністю, характерною для нативних мобільних додатків.

PWA дозволяють створювати веб-застосунки, які можуть встановлюватися на головний екран пристрою безпосередньо з браузера, працювати в умовах відсутності або нестабільного інтернет-з’єднання, надсилати push-сповіщення для залучення користувачів та використовувати багато інших сучасних веб-API, забезпечуючи таким чином користувацький досвід, максимально наближений до нативних додатків.

Метою курсової роботи є дослідження функціональних можливостей технології PWA та їхнє практичне застосування для створення веб-застосунку, який би задовольняв сучасні вимоги до зручності, доступності та функціональності. Процес створення застосунку буде розглянуто на прикладі кулінарної тематики, яка є вдалим вибором для демонстрації всього спектру можливостей PWA, включаючи доступ та взаємодію з контентом в офлайн-режимі, фонову синхронізацію даних та надсилання сповіщень про нові публікації та оновлення. Запропоноване рішення демонструє потенціал PWA для побудови сучасних застосунків без необхідності дублювання розробки для мобільних платформ, водночас забезпечуючи високу якість користувацького досвіду.

Робота складається з трьох розділів. В першому розділі проаналізовано існуючі рішення та з огляду на виявлені недоліки визначено основні функціональні вимоги до PWA-застосунку. В другому розділі обґрунтовано переваги використання технології Progressive Web App у порівнянні з традиційною розробкою веб-сайтів та нативних мобільних додатків. Детально розглянуто теоретичні основи технології Progressive Web App, її основні принципи та ключові компоненти. Третій розділ присвячено обґрунтуванню вибору стеку технологій, де також наведений детальний опис реалізації кожної з ключових функцій PWA у застосунку.

Розділ 1. Аналіз предметної області

1.1. Аналіз наявних рішень

Серед великої кількості кулінарних платформ можна виокремити кілька найпопулярніших рішень, які вже мають широку аудиторію, проте мають недоліки з точки зору користувацького досвіду.

1.1.1. Allrecipes

Allrecipes є одним із найпопулярніших кулінарних порталів у світі. Платформа надає доступ до широкого функціоналу, зокрема додавання рецептів, взаємодії з контентом у вигляді оцінок, відгуків, збереження тощо. Крім того, застосунок можна встановити на робочий стіл або мобільний пристрій, що робить його більш доступним для користувачів. Однак вся функціональність платформи повністю залежить від інтернет-з'єднання. Перегляд контенту без доступу до мережі неможливий. Для багатьох користувачів, зокрема тих, хто використовує додаток в умовах обмеженого доступу до мережі, це створює певні незручності.

1.1.2. Tasty

Ще одним популярним кулінарним сервісом є Tasty. Платформа пропонує привабливий і зручний інтерфейс, доступ до великої кількості відеорецептів, а також мобільний застосунок. Водночас більшість розширеного функціоналу, включно з особистим кабінетом, можливістю зберігати рецепти або залишати відгуки, доступна виключно в мобільному додатку. Відповідно, користувачам веб-версії може бути невідомо про ці можливості. У додатку реалізовано часткову офлайн-підтримку, а саме доступний перегляд деяких рецептів, однак

переглянути збережені рецепти в офлайн-режимі неможливо, що значно обмежує функціонал.

1.1.3. Smachno.ua

Smachno.ua — один із найпопулярніших україномовних кулінарних сайтів. Платформа містить велику базу рецептів з фотографіями та інструкціями, а його головною перевагою є наявність часткової офлайн-підтримки: переглянуті рецепти залишаються доступними без підключення до Інтернету. Однак платформа не надає можливості збереження, відповідно, знайти вподобаний рецепт пізніше може бути складно для користувача. Крім того, Smachno.ua не підтримує встановлення застосунку на пристрій, і доступ до платформи можливий лише через браузер. Таким чином, застосунок забезпечує лише базовий функціонал перегляду рецептів і не дозволяє персоналізувати досвід користувача.

1.2. Постановка завдання

Аналіз існуючих рішень показує, що навіть найпопулярніші кулінарні платформи мають помітні обмеження в зручності використання, зокрема в умовах слабкого або нестабільного інтернет-з’єднання. Саме тому метою цієї курсової роботи є створення веб-застосунку для обміну кулінарними рецептами з максимально якісним користувацьким досвідом.

Запропонований застосунок підтримуватиме офлайн-доступ до більшості функціоналу, необхідного в повсякденному користуванні, включно з можливістю перегляду збережених рецептів. Дії користувача, такі як лайки, коментарі чи підписки, автоматично зберігатимуться локально та синхронізуватимуться після відновлення з’єднання, що забезпечить безперервність користування. Застосунок надсилатиме push-сповіщення про нову активність (коментарі, лайки та

підписки), а також надаватиме можливість встановлення застосунку на головний екран мобільного пристрою чи робочий стіл.

Для досягнення поставленої мети буде використано комплексний підхід, що включає всі ключові можливості, які надає технологія Progressive Web App, що дозволяє поєднати зручність класичного веб-сайту з функціональністю нативного мобільного додатку. При цьому немає потреби створювати окремий додаток для App Store чи Google Play, адже все працюватиме в межах єдиного веб-рішення.

Розділ 2. Огляд технології Progressive Web App

2.1. Визначення та суть

Progressive Web App (PWA) – це сучасний підхід до розробки веб-застосунків, який поєднує переваги традиційних веб-сайтів і нативних мобільних додатків.

PWA створюється за допомогою стандартних веб-технологій, таких як HTML, CSS і JavaScript та можливостей сучасних веб-браузерів, але забезпечує користувацький досвід, подібний до нативних додатків, включаючи можливість офлайн-доступу, обробки запитів у фоновому режимі, встановлення на пристрій та інтеграцію з функціями операційної системи. Це дозволяє розробникам створювати універсальні застосунки, які працюють на різних платформах і пристроях, використовуючи єдину кодову базу, що значно спрощує процес розробки та обслуговування застосунків, оскільки немає потреби створювати окремі версії для різних операційних систем.

В основі PWA лежить принцип прогресивного покращення (progressive enhancement). Це означає, що застосунок спроектований так, щоб спочатку надавати базову необхідну функціональність, доступну в будь-якому браузері, а додаткові можливості інтегрувати поступово за наявності їхньої підтримки з боку браузера користувача. Наприклад, якщо браузер не підтримує API як-от Service Worker, PWA буде працювати як звичайний веб-сайт без офлайн-режиму та push-сповіщень, тоді як у сучасному браузері користувач отримає весь спектр можливостей PWA. Цей підхід забезпечує доступність та функціональність веб-застосунку для найширшої аудиторії та оптимальний користувацький досвід на кожній платформі.

2.2. Порівняння нативних додатків і класичних веб-застосунків

Порівняння нативних додатків та класичних веб-застосунків надає можливість виявити фундаментальні відмінності між цими підходами до розробки програмного забезпечення. Обидва мають свої переваги та недоліки, які впливають на вибір технології залежно від завдання та ресурсів проєкту.

Нативні додатки розробляються для конкретної операційної системи, наприклад, Android або iOS, з використанням відповідних засобів. Такі додатки встановлюються через офіційні магазини (Google Play, App Store) і після інсталяції зазвичай сприймаються користувачем як повноцінна частина пристрою.

Вони мають доступ до ресурсів операційної системи (файлової системи, камери, мікрофону, геопозиції тощо), можуть працювати у фоновому режимі та підтримують повноцінну або часткову офлайн-функціональність. Такі застосунки можуть інтегруватися з іншими програмами та системними функціями, наприклад, для поширення даних між додатками, обробки медіа тощо. Крім того, за умови надання дозволу користувач може отримувати push-сповіщення на пристрій, навіть коли додаток не активний. Це одна з найважливіших переваг нативних застосунків, що сприяє повторному залученню аудиторії.

Однак створення застосунку під конкретну платформу має низку обмежень. В першу чергу це потреба окремої розробки для кожної платформи, що суттєво збільшує витрати на реалізацію та підтримку. Публікація оновлень вимагає повторної збірки та перевстановлення на пристрої. Крім того, процес розповсюдження через магазини пов’язаний з політиками, обмеженнями й перевірками відповідних платформ. Це може затримувати випуск оновлень і часто призводить до ситуації, коли не всі користувачі оновлені до останньої версії одночасно.

Класичні веб-застосунки у свою чергу є більш універсальними. Вони працюють у браузерах на будь-яких пристроях і не потребують інсталяції, що забезпечує легкий доступ для користувача. Однією з ключових переваг є використання єдиної кодової бази для всіх платформ, що значно знижує витрати

на розробку та підтримку. Зазвичай, достатньо реалізації адаптивного дизайну, щоб інтерфейс підлаштовувався під розміри пристрою. Крім того, веб-застосунки оновлюються миттєво без участі користувача, адже зміни застосовуються одразу після повторного завантаження сторінки. Веб-застосунки також мають перевагу у розповсюдженні за рахунок механізмів пошукової оптимізації, легкості у публікації та відсутності необхідності проходити численні перевірки.

Втім, класичні веб-застосунки довгий час поступалися нативним додаткам у таких аспектах, як офлайн-функціональність, інтеграція з операційною системою, а також фонові процеси. Крім того, за відсутності можливості push-сповіщень класичний веб-сайт менш ефективний у «поверненні» користувачів, адже для цього він може покладатися хіба що на email або рекламу. Загалом, до появи PWA, веб-застосунки фактично були лише «місцем, яке користувач відвідує», а не частиною пристрою, яку він встановлює.

Progressive Web App поєднує у собі переваги обох підходів. Від нативних додатків PWA переймає офлайн-підтримку, роботу у фоновому режимі, push-сповіщення, доступ до деяких системних функцій у межах можливостей браузера, а також можливість інсталяції безпосередньо з браузера. Процес встановлення зазвичай займає кілька секунд, оскільки фактично PWA - це кешований веб-сайт, що займає набагато менше місця, ніж нативний додаток.

Водночас PWA зберігає головні переваги вебу, такі як єдина кодова база, миттєве оновлення, можливість розповсюдження як через магазини, так і напряму через Інтернет. Завдяки кешуванню останніх переглянутих даних та синхронізації після відновлення з’єднання, PWA забезпечує безперервний користувацький досвід. При цьому, за оцінками, створення PWA обходиться на 60–70% дешевше, ніж розробка відповідного нативного додатку.

Попри свої переваги, технологія PWA створює і певні виклики для розробників, які варто враховувати при її впровадженні. Насамперед це проблема сумісності. Хоча більшість сучасних браузерів підтримують основні функції PWA, їхня реалізація в деяких випадках є неповною, особливо на пристроях

Apple. Наприклад, Safari на iOS не підтримує push-сповіщення для PWA в більшості версій, обмежує інтеграцію з системою та не дозволяє встановлення. Окремий виклик становить низький рівень обізнаності користувачів про можливість встановлення веб-сайтів як додатків, тому в багатьох випадках необхідно вручну інструктувати користувача. Також необхідно враховувати, що кожна нова версія операційної системи або браузера може змінити поведінку PWA, тому постійне ретельне тестування на різних пристроях і платформах є обов'язковим. До того ж завжди необхідно забезпечувати альтернативні рішення для недоступних функцій.

Таким чином, Progressive Web Apps займають проміжне положення між двома підходами й тим самим забезпечують баланс продуктивності, наближаючись до нативних додатків у різноманітності функціоналу і значно перевершуючи класичні веб-сайти у стабільності та швидкодії. Однак ефективне впровадження PWA потребує уважного врахування специфіки середовища виконання та роботи з обмеженнями, властивими окремим платформам.

2.3. Компоненти Progressive Web App

Прогресивний веб-застосунок (PWA) реалізується як комплекс взаємопов'язаних веб-технологій, кожна з яких виконує специфічну функцію.

Ключовими складовими PWA є:

- Web App Manifest, який містить метаінформацію про застосунок, необхідну для його встановлення;
- Service Worker, який керує кешуванням даних та взаємодією з мережею за допомогою сучасних веб-API.

Варто зауважити, що всі функціональні можливості PWA працюють виключно в межах захищених з'єднань (HTTPS). Це необхідна вимога для безпеки передачі даних, оскільки PWA використовують веб-API, які можуть бути небезпечними без захищеного з'єднання.

2.3.1. Web App Manifest

Web App Manifest – це спеціальний конфігураційний JSON-файл, що визначає, як застосунок буде відображатися на пристрої користувача після встановлення. Саме наявність маніфесту є обов'язковою умовою для того, щоб браузер сприймав застосунок, як такий, що може бути інстальований користувачем.

Для підключення маніфесту потрібно у <head>-секції HTML-документа додати тег <link rel="manifest" href="manifest.json">. Якщо веб-застосунок складається з кількох сторінок, кожна з них повинна містити посилання на маніфест.

Файл маніфесту містить JSON-об'єкт із набором параметрів, які описують зовнішній вигляд і поведінку застосунку, зокрема:

- name - повне ім'я застосунку, яке відображається при запуску;
- short_name - коротке ім'я, що відображається під іконкою на головному екрані;
- start_url - URL, який відкриється при запуску застосунку;
- display - режим відображення, що визначає, чи застосунок відкривається як веб-сторінка, чи у відокремленому вікні (standalone, fullscreen, minimal-ui, browser);
- icons - масив об'єктів з різними розмірами зображень, які браузер використовує як піктограму застосунку;
- background_color - колір фону екрану завантаження;
- theme_color - основний колір інтерфейсу застосунку, наприклад, панелі інструментів.

Для коректної роботи в браузерах на базі Chromium (Google Chrome, Microsoft Edge, Samsung Internet) обов'язковими є такі поля:

- name або short_name;

- icons (набір іконок у форматі PNG, серед яких обов’язково мають бути зображення розміром 192x192 та 512x512 пікселів);
- start_url;
- display;
- prefer_related_applications (має бути відсутнім або встановленим у false, адже якщо встановити true, браузер запропонує користувачу встановити пов’язаний нативний додаток, а не PWA).

Приклад мінімального валідного файлу Web App Manifest:

```
{
  "name": "Recipegram: Progressive Web App",
  "short_name": "Recipegram",
  "start_url": "/index.html",
  "display": "standalone",
  "background_color": "#f9faef",
  "theme_color": "#476730",
  "icons": [
    {
      "src": "icons/icon-192.png",
      "sizes": "192x192",
      "type": "image/png"
    },
    {
      "src": "icons/icon-512.png",
      "sizes": "512x512",
      "type": "image/png"
    }
  ]
}
```

Рисунок 1. Приклад файлу Web App Manifest

Після підтвердження встановлення на пристрої користувача з’являється ярлик, тобто іконка PWA серед інших додатків. Браузер створює запис у системі, що веде до веб-застосунку. Однак фактично не інсталюється жоден .apk чи .exe. Це досі веб-додаток, HTML, CSS, JavaScript та інші ресурси якого продовжують завантажуватись з сервера або з кешу через Service Worker.

Після інсталяції PWA запускається у власному вікні без звичних елементів браузера, зокрема адресного рядка та кнопок навігації. За замовчуванням це режим display: standalone або fullscreen. Завдяки Service Worker застосунок може оперативно завантажувати сторінки з кешу та працювати офлайн, а при

наступних оновленнях програми користувачу достатньо оновити сторінку в браузері, і нові файли автоматично застосуються при наступному запуску.

2.3.2. Service Worker

Service Worker - це спеціалізований тип web worker-а, який працює у потоці, окремому від основного JavaScript-коду веб-застосунку. Він не має доступу до DOM і не блокує користувацький інтерфейс, що дозволяє виконувати ресурсоємні завдання без впливу на продуктивність застосунку. Завдяки цим властивостям Service Worker є ключовою технологією у структурі прогресивного веб-застосунку.

У типовій архітектурі PWA логіка додатку умовно поділяється на дві частини:

- Головний застосунок, який складається файлів HTML, CSS та JavaScript та обробляє взаємодію з користувачем;
- Service Worker, який працює у фоновому режимі та відповідає за обробку мережевих запитів, кешування, синхронізацію та push-сповіщення.

Service Worker функціонує як посередник між застосунком і мережею. Механізм роботи Service Worker базується на перехопленні подій, які виникають у браузері або надходять із сервера. Він є event-driven, тобто активується лише у відповідь на конкретні події, такі як fetch (мережеві HTTP-запити), sync (фонова синхронізація), push (отримання повідомлень) тощо.

Комунікація між Service Worker та основним застосунком відбувається асинхронно через механізм обміну повідомленнями (postMessage), оскільки жоден з потоків не має прямого доступу до середовища іншого.

За замовчуванням Service Worker прив’язується до визначеної області дії (scope) домену застосунку і може обслуговувати лише ті сторінки, URL яких підпадають під цю область. Наприклад, якщо Service Worker розташовано за адресою /app/sw.js, він контролюватиме запити лише в межах шляху /app/.

Для підключення Service Worker, його потрібно зареєструвати у головному скрипті застосунку. Ось, як це можна зробити:

```
if ('serviceWorker' in navigator) {
  navigator.serviceWorker.register('/service-worker.js');
}
```

Рисунок 2. Приклад реєстрації Service Worker

Цей виклик ініціює життєвий цикл Service Worker, що складається з наступних етапів:

- Реєстрація (registration) - браузер завантажує скрипт і готує його до виконання.
- Встановлення (install) - викликається подія install, яка виконується лише один раз для кожної версії воркера. На цьому етапі зазвичай відбувається попереднє кешування необхідних ресурсів.
- Активація (activate) - після успішного встановлення Service Worker починає обслуговування сторінок в межах своєї області дії. На цьому етапі часто відбувається очистка старих кешів, якщо до цього існувала інша версія Service Worker.
- Очікування (idle) - якщо певний час воркер неактивний, або виконує обробку надто довго, він припиняє роботу і «прокидається» лише при виникненні нової події (fetch, push, sync тощо).
- Оновлення (update) - браузер періодично перевіряє файл Service Worker на наявність змін. Якщо доступна нова версія, вона завантажується у фоновому режимі і проходить етап встановлення, але не замінює поточний активний Service Worker, доки відкрита хоча б одна вкладка, що використовує стару версію. Цей механізм гарантує, що в межах певної області дії одночасно активною є лише одна версія скрипта.

До основних функціональних можливостей Service Worker належать:

- Кешування ресурсів - збереження файлів HTML, CSS, JS, зображень та відповідей API локально;
- Офлайн-доступ - завантаження збережених ресурсів навіть без з’єднання з мережею;
- Фонова синхронізація (Background Sync) - збереження дій користувача локально з подальшою їхньою обробкою після відновлення з’єднання;
- Push-сповіщення - показ повідомлень з сервера навіть при закритому застосунку.

Важливо зазначити, що через потенційну можливість перехоплення трафіку Service Worker вимагає використання захищеного з’єднання. Він працює тільки при доступі через HTTPS. Винятком є лише локальна розробка на localhost.

2.3.3. Кешування

Кешування надає PWA можливість зберігати ресурси додатка в браузері на пристрої користувача та отримувати їх без потреби звернення до мережі. Реалізація кешування ґрунтується на Service Worker, який «слухає» fetch-події та може вирішити, звідки повернути відповідь: з кешу, якщо відповідь вже була збережена, чи з мережі. Це забезпечує два основні призначення кешування:

- Підтримка офлайн-роботи;
- Підвищення продуктивності інтерфейсу.

Основним недоліком кешування є питання актуальності даних. Збережені в кеші дані можуть застаріти, якщо оновилася інформація на сервері. Для вирішення цієї проблеми у сучасних PWA застосовуються різні стратегії кешування для різних типів ресурсів, зокрема для статичних файлів інтерфейсу та динамічних даних.

Серед найбільш поширених стратегій кешування можна виокремити:

1. Cache-first (спочатку кеш).

При кожному запиті відбувається перевірка, чи збережено ресурс у кеші. Якщо закешована версія знайдена, вона негайно повертається користувачу, якщо ні, здійснюється мережевий запит, а отримана відповідь передається користувачу і записується в кеш. Cache-first зазвичай застосовується для збереження незмінних або рідко оновлюваних ресурсів, наприклад, структури сторінок, стилів, іконок тощо.

2. Stale-while-revalidate (кеш з фоновим оновленням).

   Застосунок одразу повертає закешовану версію ресурсу, паралельно звертаючись до мережі для отримання актуальних даних. Таким чином, користувач миттєво отримує запитаний ресурс, в той час як оновлені дані завантажуються на фоні й стануть доступними при наступному зверненні. Основна мета цієї стратегії полягає в забезпеченні високої швидкодії, як у cache-first, але з додатковою актуалізацією даних.

3. Network-first (спочатку мережа).

   При кожному запиті пріоритет надається отриманню даних з мережі. Якщо ж запит завершився неуспішно, використовується закешована відповідь, якщо така присутня. Стратегія забезпечує максимальну актуальність даних, де кеш виступає резервним варіантом і надає перевагу за відсутності інтернет-з'єднання.

У сучасних PWA рекомендується комбінувати стратегії для різних потреб. Вибір залежить від конкретних вимог до швидкодії та актуальності кожного виду ресурсів.

Одним із механізмів кешування у PWA є Cache API - інтерфейс, який надає можливість зберігати пари запит-відповідь у стійкому між сесіями сховищі браузера. Cache API містить методи для додавання записів до кешу, їх отримання та вилучення. Крім того, розробник має можливість розбивати кеш на іменовані сховища для різних типів даних або призначень, наприклад для статичних ресурсів, для зображень тощо.

Варто зазначити, що інтерфейс Cache API доступний як у середовищі Service Worker, так і в головному JavaScript-потоці застосунку. Він працює асинхронно, і, відповідно, не блокує основний потік.

Як правило, Cache API використовується в середовищі Service Worker для реалізації двох ключових функцій PWA: попереднє кешування (pre-caching) і динамічне кешування.

При встановленні нового Service Worker зазвичай виконується попереднє завантаження та кешування базових статичних ресурсів застосунку, зокрема файлів HTML, CSS та JavaScript, іконок, шрифтів, тобто так званої «оболонки» застосунку (application shell). Ці файли рідко змінюються, але є критичними для миттєвого запуску інтерфейсу.

Надалі, в процесі роботи застосунку обробник fetch-подій може виконувати динамічне кешування, що полягає у перехопленні мережевих запитів і збереженні або оновленні отриманих відповідей в кеші. Такий механізм дозволяє поступово накопичувати в кеші ресурси, до яких звертається користувач, щоб наступні запити могли обслуговуватися швидше або без наявності зв’язку з мережею.

Ресурси, які зазвичай кешуються через Cache API - це передусім статичні файли та відповіді сервера, отримані через GET-запити. Інтерфейс добре підходить для ресурсів, які можна точно ідентифікувати за URL і повторно використати без змін. Натомість для даних, що часто оновлюються або потребують часткового зчитування або запису, доцільно використовувати інші механізми, які доповнюють можливості кешування, зокрема IndexedDB.

IndexedDB - це інтерфейс для збереження структурованих даних в браузері у формі бази даних. На відміну від простого кешування запитів, IndexedDB надає повноцінне сховище, що за структурою нагадує NoSQL, з можливістю створювати бази даних, таблиці (object stores) та зберігати в них JavaScript-об’єкти, масиви, рядки, двійкові великі об’єкти у форматі ключ-значення. Застосунок може виконувати вибіркове читання, оновлення та видалення записів, здійснювати фільтрацію або пошук за ключами чи індексами, тобто отримує

можливості, подібні до роботи зі стандартною базою даних. Цей API асинхронний, і тому добре підходить для роботи з великими обсягами даних і складними структурами. IndexedDB підтримується в усіх сучасних браузерах і може використовуватися як із контексту головної сторінки, так і з середовища Service Worker.

Типовий приклад застосування IndexedDB - це кешування відповідей API, які містять динамічний контент у форматі JSON, як-от списки товарів, коментарі, результати пошуку тощо. Замість того щоб зберігати та використовувати повноцінні HTTP-відповіді через Cache API, можна розпарсити відповідь і зберегти необхідні її складові в IndexedDB як окремі записи. Пізніше ці дані можна динамічно оновлювати й використовувати для відображення офлайн.

За загальною рекомендацію при розробці PWA слід розмежувати використання Cache API та IndexedDB за призначенням. Файли інтерфейсу варто кешувати засобами Cache API для швидкого завантаження, тоді як дані додатка доцільніше зберігати в IndexedDB. Важливо, що доступний обсяг пам’яті в обох сховищах є обмеженим і може відрізнятися в різних браузерах. При перевищенні ліміту браузер може самостійно очистити найстаріші дані, тому при використанні цих інтерфейсів в розробці варто контролювати обсяг накопичених даних й періодично видаляти непотрібні записи.

2.3.4. Фонова синхронізація

Фонова синхронізація (Background Sync) – це механізм, який працює в контексті Service Worker та дозволяє відкласти виконання мережевих запитів до моменту відновлення у користувача стабільного інтернет-з’єднання. Його основне призначення – гарантувати надсилання важливих даних на сервер, навіть якщо на час ініціації цих дій пристрій був в режимі офлайн. Це мінімізує ризик втрати даних та гарантує безперервність користувацького досвіду, адже застосунок залишається функціональним без мережі, а накопичені дії синхронізуються автоматично при відновленні зв’язку.

Технічно Background Sync реалізується через Background Synchronization API. Для реєстрації завдання синхронізації застосунок викликає метод registration.sync.register(tag) інтерфейсу SyncManager, де tag - це унікальний ідентифікатор запланованого завдання. Цей виклик ініціює очікування на відновлення мережі, після чого браузер згенерує подію sync у Service Worker з відповідним тегом.

Для збереження відкладених запитів зазвичай використовується IndexedDB, яка дозволяє зберігати структуровані дані. У контексті Background Sync зазвичай це:

- URL запиту;
- HTTP-метод (як правило, POST, PUT, PATCH або DELETE);
- тіло запиту (наприклад, JSON-представлення форми);
- HTTP-заголовки;
- мітка часу або унікальний ідентифікатор.

Ці дані зберігаються у відповідному сховищі в IndexedDB, а при активації sync-події зчитуються воркером для відтворення повноцінного HTTP-запиту до сервера без потреби втручання користувача.

Фонову синхронізацію доцільно використовувати для таких сценаріїв, як публікація та оновлення даних, взаємодія з контентом або інші дії, які не обов’язково потребують негайного виконання, але потребують успішного завершення.

2.3.5. Push-сповіщення

Push-сповіщення - це технологія, що дозволяє веб-застосункам отримувати повідомлення від сервера, навіть якщо сам застосунок не активний у браузері користувача. Сповіщення відображаються через системну службу повідомлень операційної системи, тому виглядають і поводяться подібно до сповіщень звичайних додатків. У контексті прогресивних веб-застосунків push-сповіщення

забезпечують механізм доставки актуальної інформації на пристрій користувача подібно до нативних мобільних додатків, що значно підвищує рівень залученості та «повернення» користувачів.

В сучасних браузерах реалізовано дві тісно пов’язані технології для роботи з push-сповіщеннями, а саме Push API та Notification API. Push API відповідає за доставку даних від сервера до браузера, тобто за отримання push-сповіщень застосунком у фоновому режимі. Notification API у свою чергу відповідає за відображення отриманих сповіщень на пристрої користувача.

Щоб забезпечити доставку повідомлень, кожен сучасний браузер реалізує власний push-сервер - спеціальний посередник між сервером застосунку та браузером користувача. Саме цей push-сервер приймає запити з сервера PWA і відповідає за їхню доставку до браузера.

Механізм роботи push-сповіщень відбувається в кілька основних етапів:

1. Реєстрація Service Worker.

   Push-повідомлення потребують попередньо зареєстрованого Service Worker. Він діє у фоновому потоці і здатен отримувати push-події, навіть якщо жодна вкладка веб-застосунку не відкрита.

2. Запит дозволу користувача.

   Щоб розпочати використання push-сповіщень, веб-застосунок повинен отримати дозвіл користувача на показ повідомлень. Для цього використовується метод Notification.requestPermission(). Це обов’язкова умова, адже браузер блокує спроби підписки без попереднього схвалення користувача.

3. Підписка на push-сервер.

   Після отримання дозволу веб-застосунок викликає метод pushManager.subscribe() з інтерфейсу Push API. Цей виклик:

   - створює унікальний endpoint URL для користувача;
   - генерує пару криптографічних ключів, які використовуватимуться для шифрування повідомлень;
   - реєструє користувача на власному push-сервері браузера.

Після цього браузер повертає об’єкт PushSubscription з такими полями:

- endpoint - адреса push-сервера браузера;
- keys - публічні ключі для шифрування повідомлень.

Цей об’єкт далі надсилається на сервер застосунку, який зберігає його для подальшого використання при надсиланні повідомлень користувачу.

Важливо, що щоб ідентифікувати застосунок, підписка повинна включати VAPID-ключ (Voluntary Application Server Identification), який генерується розробником і дозволяє браузерному push-серверу перевірити, що запит надійшов від довіреного джерела. Цей публічний ключ передається під час виклику subscribe(), а приватний зберігається на сервері та використовується для підпису повідомлень.

4. Надсилання push-сповіщення.

Щоб повідомити користувача про нову подію, наприклад про новий коментар під його постом, сервер формує повідомлення, шифрує його з використанням збережених ключів із PushSubscription та надсилає запит до push-сервера браузера. Push-сервер зберігає повідомлення і, якщо пристрій користувача онлайн, негайно надсилає його через браузер, а в іншому випадку повідомлення додається в чергу.

5. Обробка push-події у Service Worker.

Коли надходить нове повідомлення, браузер активує push-подію, а Service Worker її оброблює за допомогою власного метода showNotification(), що відповідає за показ сповіщення користувачу.

Користувач також може взаємодіяти зі сповіщенням, наприклад, натиснути на нього. Це викликає іншу подію у Service Worker - notificationClick. Її обробка може надати можливість відкрити вкладку з визначеним URL або виконати іншу дію, залежно від потреб застосунку.

Розділ 3. Реалізація веб-застосунку

3.1. Функціональні можливості застосунку

Для практичного застосування функцій технології Progressive Web App було створено веб-застосунок кулінарної тематики. Серед основних можливостей застосунку:

- реєстрація та автентифікація користувачів;
- додавання, редагування та видалення рецептів з фото;
- перегляд рецептів інших користувачів з можливістю фільтрації за інгредієнтами та пошуку за назвою;
- перегляд профілів інших користувачів;
- додавання коментарів, лайків, підписок;
- офлайн доступ до перегляду рецептів;
- фонова синхронізація лайків, коментарів та підписок;
- push-сповіщення про нові коментарі, лайки та підписки;
- встановлення застосунку на пристрій;

3.2. Огляд використаних технологій

Для реалізації зазначеного функціоналу було обрано стек PERN (PostrgeSQL, Express.js, React, Node.js). Цей стек охоплює повний набір технологій для розробки повноцінного застосунку з клієнт-серверною архітектурою, що забезпечує єдине середовище розробки на мові JavaScript і відповідає вимогам сучасних веб-застосунків. До комонентів стеку PERN входять:

- PostrgeSQL - реляційна система керування базами даних (СКБД) з відкритим кодом. Вона добре підходить для моделювання взаємопов’язаних сутностей, а також підтримує всі необхідні механізми

для забезпечення цілісності даних, високого рівня продуктивності та масштабованості. За час свого існування PostgreSQL зарекомендувала себе як надійна система й стала своєрідним стандартом серед реляційних СКБД.

- Express.js - веб-фреймворк JavaScript для роботи з Node.js. Він призначений для швидкого й легкого створення веб-сервісів і RESTful API, надаючи зручні засоби маршрутизації, підключення проміжного програмного забезпечення та обробки HTTP-запитів і відповідей. У межах стеку PERN Express.js відповідає за реалізацію бекенд-логіки застосунку.
- React - найбільш популярна й широко застосовна JavaScript-бібліотека для створення користувацьких веб-інтерфейсів. React використовується для розробки фронтенду у вигляді односторінкового застосунку (Single Page Application) та використовує компонентно-орієнтований підіхід у побудові UI, що забезпечує високу зручніть розробки, а також максимально динамічний та швидкий інтерфейс.
- Node.js - середовище виконання JavaScript-коду поза браузером. Воно слугує платформою для запуску сервера на Express.js, дозволяючи використовувати єдину мову JavaScript при розробці як фронтенду, так і бекенду.

Архітектура застосунку була побудована за принципом клієнт-серверної взаємодії. Фронтенд на React, та бекенд на Express.js функціонують як окремі сервіси, а обмін даними між ними відбувається через RESTful API у форматі JSON. Такий підхід забезпечує чіткий поділ логіки представлення та обробки даних, а також спрощує масштабування, повторне використання API та тестування й відлагодження окремих компонентів.

При розробці клієнської частині застосунку було обрано сучасний інструмент Vite для збірки фронтенду, який забезпечує швидке завантаження проєкту під час розробки та оптимізовану збірку. Vite використовує ES-модулі та забезпечує менший час очікування при розробці в порівнянні з класичними збирачами, як-от Webpack. Крім того, Vite надає офіційний плагін vite-plugin-

pwa, який дозволяє значно полегшити та оптимізувати реалізацію основних функцій PWA.

У клієнтській частині застосунку для виконання HTTP-запитів було використано бібліотеку axios - популярну JavaScript-бібліотеку для роботи з API, яка забезпечує зручний синтаксис, підтримку обробки помилок, інтерсептори запитів і відповідей, а також автоматичну трансформацію даних.

Оскільки за замовчуванням axios використовує XMLHttpRequest, який не підтримується Service Worker-ом, у конфігурації клієнта було використано fetch-адаптер:

```
import axios from "axios";

const api : AxiosInstance = axios.create({
  baseURL: `${import.meta.env.VITE_API_BASE_URL}/api/`,
  adapter: "fetch",
  withCredentials: true,
});

export default api; Show usages  Daria Vetrykush
```

Рисунок 3. Налаштування fetch-адаптера для axios

Це дозволяє виконувати всі запити axios через інтерфейс fetch(), який підтримується Service Worker-ом.

3.2.1. Робота з базою даних

Для побудови ефективної та узгодженої структури бази даних, у процесі розробки було створено ER-модель, що відображає всі сутності та зв’язки між ними. Ця модель слугувала основою для побудови моделей, визначення зв’язків та формування бізнес-логіки застосунку.

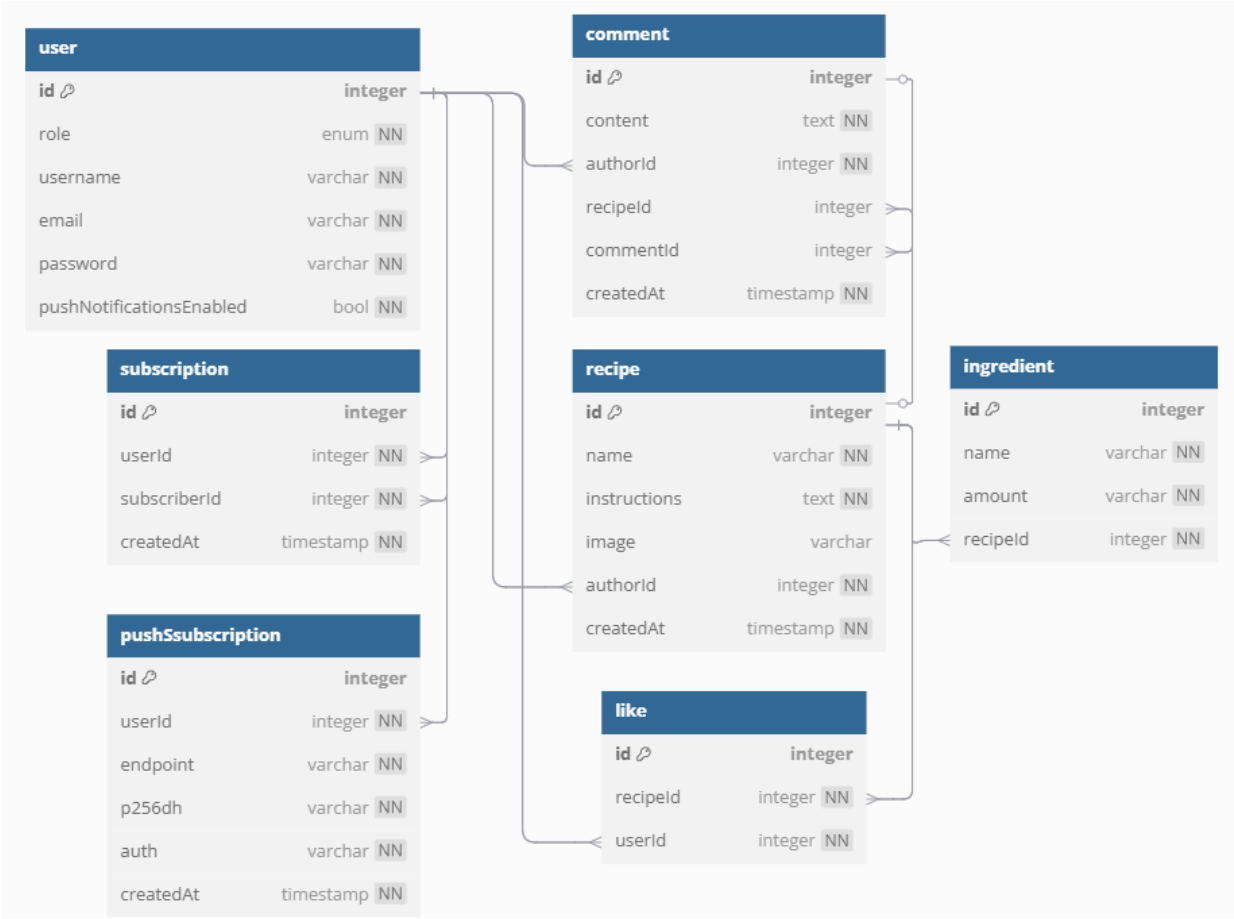

Рисунок 4. ER-модель

Для взаємодії із PostgreSQL було використано ORM-бібліотеку Sequelize. Вона надає можливість об'єктно-реляційного відображення даних, що дозволяє описувати структуру таблиць та зв’язки між ними у вигляді моделей JavaScript.

Sequelize підтримує основні типи зв’язків між сутностями: «один до одного» (hasOne), «один до багатьох» (hasMany), «багато до багатьох» (belongsToMany), що дає змогу легко моделювати реляційні структури, наприклад, «користувач - багато рецептів», «рецепт - багато коментарів» тощо. Бібліотека забезпечує вбудовану валідацію атрибутів моделей, що дозволяє перевіряти дані до їхнього збереження, а також підтримує scopes - механізм для повторного використання типових умов запиту.

Крім того, Sequelize надає гнучкість у формуванні запитів, адже є можливість використовувати як методи високого рівня, наприклад, findAll, create, update, так і виконувати «сирі» SQL-запити через метод sequelize.query() при

потребі оптимізації або нестандартної логіки. Завдяки цьому ORM поєднує простоту й декларативність з високим контролем над виконанням запитів.

До того ж використання ORM підвищує безпеку застосунку, знижуючи ризик помилок і типових вразливостей, наприклад, за рахунок автоматичного захисту від SQL-ін'єкцій.

Для керування змінами структури бази даних у проєкті було використано механізм міграцій Sequelize. Міграції дозволяють версіонувати схему бази даних, забезпечуючи контрольоване внесення змін без втрати даних.

Міграції в Sequelize - це JavaScript-файли, які експортують дві функції: up для застосування змін і down для їхнього скасування. Sequelize також автоматично створює таблицю SequelizeMeta, яка відстежує, які міграції були застосовані. Це надає можливість контрольованого та безпечного оновлення структури бази даних, а також спільної роботи над проєктом без конфліктів у схемі.

3.2.2. Автентифікація та авторизація

Для реалізації механізму автентифікації та авторизації в застосунку було використано бібліотеку Passport.

Passport - це гнучка бібліотека для Node.js, яка спрощує реалізацію автентифікації запитів у веб-застосунках. Вона підтримує понад 500 стратегій автентифікації, включаючи локальну, JWT, OAuth та інші. Її використання є досить простим. Необхідно вказати обрану стратегію, а Passport у свою чергу надає відповідні інструменти для обробки результатів автентифікації.

У веб-застосунку було використано стратегію passport-jwt. Користувач проходить автентифікацію за допомогою електронної пошти та паролю, після чого на основі унікального ідентифікатора (user.id) і ролі (user.role) генерується токен JSON Web Token. Процес генерації токена здійснюється за допомогою бібліотеки jsonwebtoken. Він зберігається у HTTP-only cookie з визначеним

терміном дії, що підвищує безпеку, запобігаючи доступу до нього з боку клієнтського JavaScript-коду.

Для обробки автентифікації passport-jwt вилучає токен із cookie, перевіряє його дійсність та ідентифікує користувача. При наявності валіного токена, об’єкт користувача додається до запиту. У випадку недійсного або простроченого токена система повертає відповідне повідомлення про помилку авторизації. Захист маршрутів реалізовано через проміжну функцію, яка реалізовує вище описаний алгоритм.

3.2.3. Обробка та валідація отриманих даних на сервері

Для забезпечення цілісності та безпеки даних у веб-застосунку було впроваджено серверну валідацію за допомогою бібліотеки express-validator. Цей інструмент надає гнучкий механізм для перевірки та очищення вхідних даних у запиті, зокрема параметрів та тіла запиту.

Основні можливості express-validator:

- Validation - перевірка даних на відповідність заданим критеріям, наприклад обов’язковість поля, його тип, формат тощо.
- Sanitazing - очищення вхідних даних від потенційно небезпечних символів для убезпечення застосунку від атак типу XSS.

Крім того, бібліотека надає можливість створення власних функцій для специфічних перевірок або обробки даних.

Для централізованої обробки результатів валідації було створено спеціальну функцію, яка застосовується після всіх валідаторів, та у разі наявності помилок передає відповідне повідомлення клієнту.

3.2.4. Робота з зображеннями

У веб-застосунку реалізовано функціональність завантаження зображень рецептів за допомогою бібліотеки express-fileupload. Ця бібліотека забезпечує обробку вхідних файлів у форматі multipart/form-data та надає зручний інтерфейс для їх збереження на сервері.

Для обробки зображень express-fileupload підключається як проміжне програмне забезпечення, що дозволяє отримувати файли запитів через об’єкт req.files.

При розробці було прийняте рішення реалізувати збереження зображень рецептів безпосередньо на сервері. Такий підхід забезпечує повний контроль над файлами, не перевантажує базу даних великими обсягами даних та не потребує інтеграції зі сторонніми хмарними сервісами, що спрощує реалізацію без втрати функціональності.

Завантажені файли зберігаються у файловій системі на сервері, а в базу даних записується лише ім’я файлу у вигляді згенерованого унікального UUID. Це дозволяє оптимізувати запити та мінімізує ризик конфліктів імен під час збереження.

З міркувань безпеки та ефективного використання ресурсів, у застосунку реалізовано наступні обмеження:

- Максимальний розмір файлу обмежено до 5 МБ, що дозволяє уникнути перевантаження файлової системи та знижує ризики, пов’язані з DoS-атаками через великі об'єми даних.
- Типи файлів обмежено до зображень у форматах .jpeg, .jpg та .png, що виключає можливість завантаження шкідливих файлів.

3.3. Впровадження функцій Progressive Web App

Для інтеграції технології PWA у клієнтській частині застосунку було використано офіційне розширення vite-plugin-pwa для Vite. Воно забезпечує

автоматизацію додавання необхідних елементів PWA, зокрема Web App Manifest, Service Worker та механізмів кешування ресурсів.

Vite-plugin-pwa працює на основі Workbox - офіційного набору бібліотек та інструментів від Google для спрощення розробки Service Worker-ів. Workbox надає API вищого рівня, чим дозволяє автоматизувати багато складних аспектів роботи з Service Worker API.

Основні компоненти Workbox включають:

- workbox-core - базові інструменти для роботи із Service Worker;
- workbox-routing - модуль для налаштування правил обробки мережевих запитів;
- workbox-strategies - набір готових реалізацій стратегій кешування;
- workbox-expiration - модуль для налаштовування обмежень розміру кешу та автоматичного видалення старих записів;
- workbox-precaching - модуль для попереднього кешування ресурсів;
- workbox-background-sync - інструменти для реалізації фонової синхронізації запитів;
- workbox-recipes - набір готових реалізацій поширених сценаріїв.

Vite-plugin-pwa підтримує дві основні стратегії генерації Service Worker-а з Workbox:

- generateSW - плагін викликає метод Workbox generateSW, який на основі визначеної розробником конфігурації автоматично генерує готовий файл воркера, що містить правила попереднього та динамічного кешування. Цей режим доцільно використовувати, якщо потрібне просте кешування без додаткових особливих сценаріїв;
- injectManifest - розробник самостійно створює файл воркера, де визначає власну логіку обробки подій, а плагін викликає метод Workbox injectManifest, який вбудовує у цей файл автоматично згенерований перелік ресурсів для попереднього кешування (precache manifest).

У проєкті було використано стратегію injectManifest, що дозволило максимально кастомізувати поведінку Service Worker відповідно до вимог застосунку. Конфігурація плагіна була реалізована у файлі vite.config.js. У ній були визначені наступні параметри:

- registerType: «autoUpdate» - вказує, що Service Worker має автоматично оновлюватися при зміні контенту;
- strategies: «injectManifest» - обрана стратегія генерації воркера;
- injectManifest.swSrc: «src/sw.js» - шлях до початкового Service Worker-файлу;
- injectManifest.globPatterns - список шаблонів файлів, які підлягають автоматичному кешуванню при інсталяції Service Worker;
- manifest - опис параметрів Web App Manifest, які будуть використані для генерації відповідного файлу при збірці.

```
export default defineConfig( config: {  no usages  Daria Vetrykush *
  plugins: [
    react(),
    VitePWA( userOptions: {
      registerType: "autoUpdate",
      strategies: "injectManifest",
      srcDir: "src",
      injectManifest: {
        swSrc: "src/sw.js",
        globPatterns: [
          "**/*.html",
          "**/*.js",
          "**/*.css",
          "**/*.ico",
          "**/*.png",
          "**/*.jpg",
        ],
      },
      manifest: {
        name: "Recipegram: Progressive Web App",
        short_name: "Recipegram",
        description:
          "A progressive web app for sharing your favorite recipes
        theme_color: "#476730",
        background_color: "#f9faef",
        display: "standalone",
        start_url: "/",
        scope: "/",
        icons: [
          {
            src: "pwa-64x64.png",
            sizes: "64x64",
            type: "image/png",
          },
```

Рисунок 5. Файл конфігурації Vite vite.config.json

3.3.1. Реалізація кешування

У проєкті реалізація кешування відбувається на двох рівнях:

- статичні ресурси - через Cache API з використанням Workbox;
- динамічні дані - через IndexedDB.

Для обробки кешування статичних файлів у Service Worker застосовано набір інструментів Workbox, зокрема:

- precacheAndRoute(self.__WB_MANIFEST) під час встановлення ServiceWorker автоматично кешує ресурси, які було вказано у globPatterns конфігурації vite-plugin-pwa;
- cleanupOutdatedCaches() очищає застарілі дані з кешу;
- googleFontsCache() з модуля workbox-recipes автоматично кешує шрифти з Google;

Крім того, реалізовано кешування зображень рецептів з використанням стратегії CacheFirst та обмеженням кількості й часу збереження в кеші через ExpirationPlugin для уникнення переповнення сховища кеша. При виникненні помилок в якості відповіді на запит надсилається стандартне зображення.

Для збереження динамічного контенту (рецепти, користувачі, інгредієнти тощо) реалізовано окреме кешування з використанням бібліотеки idb, яка є обгорткою над інтерфейсом IndexedDB й надає більш зручний та сучасний синтаксис.

Була створена окрема база даних, у якій визначено такі objectStore:

- recipes, favorites, users, ingredients - зберігання сутностей за id або name;
- profile - персоналізовані дані користувача;
- currentUser - дані автентифікації користувача для використання в режимі офлайн;
- appData - службова інформація (наприклад, прапор firstVisit);
- deferredQueue - відкладені запити для фонової синхронізації.

Операції кешування було реалізовано через утиліти saveToIDB, saveArrayToIDB, getFromIDB, getAllFromIDB, getPagedArrayFromIDB, deleteFromIDB, clearIDBStore, які забезпечують отримання даних з можливістю пагінації, оновлення або злиття даних з вже існуючими, а також вилучення даних.

Після успішної автентифікації користувача виконується початкове кешування персональних даних, а саме:

- профіль користувача;
- рецепти користувача з детальною інформацією;
- вподобані рецепти користувача з детальною інформацією.

Це забезпечує доступ до найважливішої інформації навіть без з'єднання з мережею та без потреби попереднього звернення до цих даних при наявності з'єднання.

Загальна логіка кешування динамічних даних через IndexedDB реалізує підхід, аналогічний стратегії NetworkFirst, тобто застосунок намагається отримати найактуальніші дані з API й оновлює їх в кеші, а при виникненні помилки за відсутності з'єднання одразу звертається до кешу. Таким чином, досягається як максимальна актуальність контенту, так і стійкість до нестабільного з'єднання.

Для сторінок зі списками рецептів або користувачів було використано кастомний React-хук usePaginatedData, який при наявності з'єднання зберігає або оновлює отримані дані у відповідному сховищі, а в офлайн-режимі отримує дані з кеша через утиліту getPagedArrayFromIDB, що зберігає можливість пагінації, та фільтрує дані за заданими користувачем критеріями.

На сторінці перегляду детальної інформації про рецепт при онлайн-завантаженні кеш оновлюється у всіх сховищах, де може міститись цей рецепт (favorites, profile, recipes) для підтримки узгодженості закешованих даних, а в режимі офлайн, відображається перша знайдена закешована версія серед цих сховищ.

3.3.2. Реалізація фонової синхронізації

У проєкті реалізовано підтримку Background Sync для відкладеного виконання деяких запитів до серверу, зокрема POST та DELETE для створення та видалення коментарів і лайків, а також DELETE для видалення рецептів, в умовах нестабільного або відсутнього інтернет-з'єднання.

Для цього було використано плагін BackgroundSyncPlugin з workbox-background-sync, в межах якого було створено чергу «defaultQueue» із максимальним часом утримання запитів 24 години. Також було додано обробник onSync() з кастомною логікою з урахуванням нюансів авторизації. Він поступово обробляє відкладені запити за допомогою queue.shiftRequest(). Якщо запит успішний, він виключається з черги. Якщо повертається статус 401 (Unauthorized), це означає, що на момент відновлення з'єднання користувач вже неавторизований, тому інформація про запит, зокрема url, HTTP-метод, headers та body, додатково зберігається до кастомної черги «deferredQueue» у IndexedDB для повторного надсилання одразу після успішної автентифікації. У випадку помилки мережі запит повертається в чергу за допомогою queue.unshiftRequest(entry).

```
const backgroundSyncPlugin : BackgroundSyncPlugin = new BackgroundSyncPlugin( name: "defaultQueue", options: {
  maxRetentionTime: 24 * 60,
  async onSync({ queue : Queue }) : Promise<void> {
    let entry;
    while ((entry = await queue.shiftRequest())) {
      try {
        const response : Response = await fetch(entry.request.clone());
        if (response.status === 401) {
          const cloned : Request = entry.request.clone();
          const body : string = await cloned.clone().text();

          const headers : {} = {};
          cloned.headers.forEach( callbackFn: (value : string , key : string ) : void => {
            headers[key] = value;
          });

          await addToDeferredQueue( entry: {
            url: cloned.url,
            method: cloned.method,
            headers,
            body,
          });
        }
      } catch (error) {
        console.error("Network error:", error);
        await queue.unshiftRequest(entry);
        break;
      }
    }
  },
});
```

Рисунок 6. Реалізація BackgroundSyncPlugin

Така реалізація гарантує, що жодна взаємодія користувача в офлайн-режимі не буде втрачена, а дані будуть синхронізовані автоматично.

3.3.3. Реалізація push-сповіщень

Для реалізації механізму push-сповіщень у застосунку було використано Push API та Notifications API у поєднанні з бібліотекою web-push на сервері. Бібліотека web-push - це Node.js-інструмент для надсилання push-сповіщень через Push API. Вона реалізує протокол Web Push та підтримує шифрування повідомлень згідно зі специфікацією. Зокрема, бібліотека дозволяє генерувати пари VAPID-ключів та надсилати зашифровані push-сповіщення на клієнтські підписки за збереженими enpoint.

У застосунку push-сповіщення надсилаються при наступних діях:

- Лайк рецепту - автор рецепту отримує повідомлення з ім’ям користувача, який поставив лайк, та назвою рецепта.
- Новий підписник - користувач отримує сповіщення з ім’ям нового підписника.
- Коментар або відповідь на коментар - автор рецепту або коментаря отримує відповідне повідомлення з контекстом та ім’ям користувача.

На бекенді реалізовано обробку push-підписок за допомогою Sequelize-моделі PushSubscription, яка зберігає такі параметри:

- endpoint - унікальна адреса push-сервера браузера;
- auth та p256dh - ключі шифрування повідомлень;
- userId - зв'язок із відповідним користувачем.

Підписки обробляються через DAO-функції:

- saveSubscription() створює або оновлює одну підписку;
- removeSubscription() видаляє одну підписку за заданими endpoint та userId;

- removeAllSubscriptions() видаляє всі підписки заданого користувача за userId;
- getSubscriptionsByUserId() повертає всі підписки заданого користувача за userId.

Для шифрування та автентифікації повідомлень використовується пара VAPID-ключів, які генеруються один раз і зберігаються на сервері в .env файлі. Приватний використовується в методі web-push sendNotification для підпису повідомлень, а публічний ключ передається клієнту перед створенням підписки для автентифікації.

Повідомлення надсилаються через функцію sendPushNotification(), яка отримує всі підписки заданого користувача через функцію getSubscriptionsByUserId(), використовує вказану вище функцію sendNotification з web-push і надсилає повідомлення з відповідним payload на кожен endpoint користувача.

Payload складається з таких полів:

- title – заголовок сповіщення;
- body – текст сповіщення;
- data.url – URL сторінки, на яку буде перенаправлено коритстувача при натисненні на сповіщення.

У клієнтській частині застосунку реалізовано окремий React-компонент PushNotificationToggle, що дозволяє користувачу керувати дозволом на отримання повідомлень. При увімкненні:

1. Запитується дозвіл на відображення сповіщень через Notification.requestPermission().
2. Отримується публічний VAPID-ключ з бекенду.
3. Виконується реєстрація підписки через serviceWorker.pushManager.subscribe() з такими параметрами:
    - userVisibleOnly: true – обов’язковий параметр з міркувань безпеки, який означає, що кожне надіслане push-сповіщення обов’язково повинне бути видимим для користувача;

- aplicationServerKey – публічний VAPID-ключ.

4. Підписка передається на сервер й зберігається в базі даних.

При відключенні підписка видаляється у браузері спеціальниим методом serviceWorker.pushManager.unsubscribe(), а на сервері видаляються всі збережені підписки користувача, тобто користувач перестає отримувати сповіщення в усіх браузерах та на всіх пристроях, з яких він міг відвідувати застосунок.

Користувацький вибір зберігається у полі pushNotificationsEnabled в моделі User на сервері, а клієнт отримує цю інформацію при кожному запиті на автентифікацію користувача.

Push-підписка також синхронізується автоматично при зміні стану автентифікації, для убезпечення від отримання «чужих» сповіщень різними користувачами, що відвідують застосунок з одного пристрою. При успішному вході, якщо в користувача увімкнено сповіщення, викликається createPushSubscription(), що створює нову підписку в браузері та на сервері. При виході з системи виконується deletePushSubscription(), що видаляє поточну підписку в браузері та на сервері.

Обробка отриманих сповіщень відбувається в Service Worker, який перехоплює події push та notificationClick. Після надходження нового повідомлення, обробник події push відображає отриманий payload у вигляді нативного сповіщення браузера. При натисканні користувачем на повідомлення активується подія notificationClick, обробник якої відкриває або фокусує відповідну сторінку застосунку у браузері користувача, наприклад при сповіщенні про нову підписку відкриває сторінку профіля нового підписника.

Висновок

У межах курсової роботи було проведено комплексне дослідження можливостей технології Progressive Web App та реалізовано повнофункціональний веб-застосунок кулінарної тематики з використанням відповідного підходу. Основна мета роботи, а саме створення сучасного застосунку, здатного забезпечити якісний користувацький досвід навіть в умовах нестабільного або відсутнього інтернет-з’єднання, була досягнута.

На основі проведеного аналізу існуючих кулінарних сервісів були виявлені типові обмеження, зокрема повна залежність від мережі, обмежена офлайн-функціональність або відсутність можливості персоналізації контенту. Запропонований у роботі застосунок вирішує ці проблеми завдяки поєднанню таких функцій, як кешування статичних та динамічних ресурсів, офлайн-доступ, фонова синхронізація дій користувача, підтримка встановлення на пристрій, а також push-сповіщення.

Для реалізації застосунку було обрано стек технологій PERN (PostgreSQL, Express.js, React, Node.js), який забезпечив ефективний розподіл відповідальності між клієнтською та серверною частинами, а також дозволив реалізувати архітектуру RESTful API. Технологія PWA була інтегрована за допомогою плагіну Vite PWA, Workbox та IndexedDB, що забезпечило гнучке управління кешем і підтримку фонових процесів.

Таким чином, результати роботи демонструють практичну застосовність і ефективність використання технології PWA для створення сучасних веб-застосунків. Застосунок відповідає ключовим вимогам до продуктивності, доступності, стабільності та взаємодії з користувачем, а отриманий результат може бути використаний як основа для подальшого розширення функціоналу.

Список використаних джерел

1. Офіційна сторінка веб-сайту Allrecipes. URL: https://www.allrecipes.com/
2. Офіційна сторінка веб-сайту Tasty. URL: https://tasty.co/
3. Офіційна сторінка веб-сайту Smachno.ua. URL: https://smachno.ua/ua/
4. What is a progressive web app? URL: https://developer.mozilla.org/en-US/docs/Web/Progressive_web_apps/Guides/What_is_a_progressive_web_app
5. Progressive Web Apps. URL: https://web.dev/learn/pwa/progressive-web-apps?continue=https%3A%2F%2Fweb.dev%2Flearn%2Fpwa%23article-https%3A%2F%2Fweb.dev%2Flearn%2Fpwa%2Fprogressive-web-apps
6. Web app manifest. URL: https://web.dev/learn/pwa/web-app-manifest
7. Making PWAs installable. URL: https://developer.mozilla.org/en-US/docs/Web/Progressive_web_apps/Guides/Making_PWAs_installable
8. Offline and background operation. URL: https://developer.mozilla.org/en-US/docs/Web/Progressive_web_apps/Guides/Offline_and_background_operation
9. Service Workers. URL: https://web.dev/learn/pwa/service-workers
10. Service Worker API. URL: https://developer.mozilla.org/en-US/docs/Web/API/Service_Worker_API
11. Caching. URL: https://developer.mozilla.org/en-US/docs/Web/Progressive_web_apps/Guides/Caching
12. Caching. URL: https://web.dev/learn/pwa/caching
13. Serving. URL: https://web.dev/learn/pwa/serving
14. IndexedDB API. URL: https://developer.mozilla.org/en-US/docs/Web/API/IndexedDB_API
15. Cache. URL: https://developer.mozilla.org/en-US/docs/Web/API/Cache
16. Offline data. URL: https://web.dev/learn/pwa/offline-data?continue=https%3A%2F%2Fweb.dev%2Flearn%2Fpwa%23article-https%3A%2F%2Fweb.dev%2Flearn%2Fpwa%2Foffline-data

17. Offline Storage for Progressive Web Apps. URL: https://medium.com/dev-channel/offline-storage-for-progressive-web-apps-70d52695513c#:~:text=For%20URL%20addressable%20resources%2C%20use,with%20a%20Promises%20wrapper
18. Background Synchronization API. URL: https://developer.mozilla.org/en-US/docs/Web/API/Background_Synchronization_API
19. Introducing Background Sync. URL: https://developer.chrome.com/blog/background-sync
20. Push API. URL: https://developer.mozilla.org/en-US/docs/Web/API/Push_API
21. Notifications API. URL: https://developer.mozilla.org/en-US/docs/Web/API/Notifications_API
22. How to make PWAs re-engageable using Notifications and Push. URL: https://developer.mozilla.org/en-US/docs/Web/Progressive_web_apps/Tutorials/js13kGames/Re-engageable_Notifications_Push
23. Using Push Notifications in PWAs: The Complete Guide. URL: https://www.magicbell.com/blog/using-push-notifications-in-pwas
24. Get Started with the PERN Stack: An Introduction and Implementation Guide. URL: https://medium.com/@ritapalves/get-started-with-the-pern-stack-an-introduction-and-implementation-guide-e33c55d09994
25. Офіційна документація Vite PWA. https://vite-pwa-org.netlify.app/
26. Axios. URL: https://www.npmjs.com/package/axios
27. Офіційна документація Sequelize. URL: https://sequelize.org/
28. Офіційна документація Passport. URL: https://www.passportjs.org/
29. Офіційна документація express-validator. URL: https://www.passportjs.org/
30. Express-fileupload. URL: https://www.npmjs.com/package/express-fileupload
31. Офіційна документація Workbox. URL: https://developer.chrome.com/docs/workbox
32. Web-push. URL: https://www.npmjs.com/package/web-push