МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
ЗАПОРІЗЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ

МАТЕМАТИЧНИЙ ФАКУЛЬТЕТ
Кафедра програмної інженерії

КВАЛІФІКАЦІЙНА РОБОТА МАГІСТРА

на тему: **«РОЗРОБКА PWA-ЗАСТОСУНКУ ДЛЯ ВЕДЕННЯ ОБЛІКУ ПОДАТКІВ ІЗ ВИКОРИСТАННЯМ NODE.JS»**

Виконав: студент 2 курсу, групи 8.1213-іпз-1
спеціальності 121 інженерія програмного забезпечення
(шифр і назва спеціальності)
освітньої програми інженерія програмного забезпечення
(назва освітньої програми)
А.В. Бондаренко
(ініціали та прізвище)

Керівник доцент кафедри програмної інженерії, доцент, к.т.н. Лимаренко Ю.О.
(посада, вчене звання, науковий ступінь, прізвище та ініціали)

Рецензент професор кафедри комп'ютерних наук, професор, д.т.н. Шило Г.М.
(посада, вчене звання, науковий ступінь, прізвище та ініціали)

Запоріжжя – 2024

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ

ЗАПОРІЗЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ

Факультет математичний

Кафедра програмної інженерії

Рівень вищої освіти магістр

Спеціальність 121 інженерія програмного забезпечення
(шифр і назва)

Освітня програма інженерія програмного забезпечення

ЗАТВЕРДЖУЮ
Завідувач кафедри програмної інженерії, к.ф.-м.н., доцент

_____ Лісняк А.О.
(підпис)

“ ____ ” _____ 2024 р.

З А В Д А Н Н Я

НА КВАЛІФІКАЦІЙНУ РОБОТУ СТУДЕНТОВІ

Бондаренку Артуру Вікторовичу
(прізвище, ім’я та по-батькові)

1. Тема роботи Розробка PWA-застосунку для ведення обліку податків із використанням Node.js

керівник роботи Лимаренко Юлія Олександрівна, к.т.н., доцент
(прізвище, ім’я та по-батькові, науковий ступінь, вчене звання)

затверджені наказом ЗНУ від « 01 » березня 2024 року № 385-с

2. Строк подання студентом роботи 25.11.2024 р.

3. Вихідні дані до роботи 1. Постановка задачі.
2. Перелік літератури.

4. Зміст розрахунково-пояснювальної записки (перелік питань, які потрібно розробити)
1. Постановка задачі.
2. Основні теоретичні відомості.
3. Розробка програмного PWA-застосунку обліку податків.

5. Перелік графічного матеріалу (з точним зазначенням обов’язкових креслень)
презентація за темою доповіді

6. Консультанти розділів роботи

Розділ	Прізвище, ініціали та посада консультанта	Підпис, дата	
		Завдання видав	Завдання прийняв

7. Дата видачі завдання 07.03.2024 р.

К А Л Е Н Д А Р Н И Й П Л А Н

№	Назва етапів кваліфікаційної роботи	Строк виконання етапів роботи	Примітка
1.	Розробка плану роботи.	12.04.2024	
2.	Збір вихідних даних.	30.05.2024	
3.	Обробка методичних та теоретичних	21.06.2024	
	джерел.		
4.	Розробка першого та другого розділу.	27.08.2024	
5.	Розробка третього розділу.	31.10.2024	
6.	Оформлення та нормоконтроль	20.11.2024	
	кваліфікаційної роботи магістра.		
7.	Захист кваліфікаційної роботи.	11.12.2024	

Студент _____ А.В. Бондаренко
(підпис) (ініціали та прізвище)

Керівник роботи _____ Ю.О. Лимаренко
(підпис) (ініціали та прізвище)

Нормоконтроль пройдено

Нормоконтролер _____ А.В. Столярова
(підпис) (ініціали та прізвище)

РЕФЕРАТ

Кваліфікаційна робота магістра «Розробка PWA-застосунку для ведення обліку податків із використанням Node.js»: 58 с., 40 рис., 5 табл., 21 джерело.

БАЗА ДАНИХ, БЕКЕНД, ОБЛІК ПОДАТКІВ, ПРОГРЕСИВНИЙ ВЕБЗАСТОСУНОК, ФРОНТЕНД, JAVASCRIPT, NODE.JS, PWA, REST API, TYPESCRIPT.

Об’єкт дослідження – процес автоматизації обліку податків та його оптимізація за допомогою прогресивних вебзастосунків (PWA) та технології Node.js.

Предмет дослідження – використання прогресивного вебзастосунку для автоматизації податкового обліку.

Мета роботи: розробка PWA-застосунку для ведення обліку податків з використанням Node.js, що дозволить автоматизувати процеси збору, зберігання та аналізу податкової інформації.

Методи дослідження – об’єктно-орієнтоване програмування, методи проєктування користувацьких інтерфейсів, методи побудови API з використанням REST-протоколу та баз даних.

Результати дослідження: у роботі розроблено та впроваджено прогресивний вебзастосунок для ведення обліку податків. Застосунок забезпечує зручність у роботі для користувачів, високий рівень автоматизації обробки податкових даних.

SUMMARY

Master's qualifying paper "Development of a Progressive Web Application for Tax Record Keeping Using Node.js": 58 pages, 40 figures, 5 tables, 21 references.

BACKEND, DATABASE, FRONTEND, JAVASCRIPT, NODE.JS, PWA, REST API, TAX ACCOUNTING, TYPESCRIPT.

The object of the study is the process of automating tax accounting and optimizing it through the use of progressive web applications (PWA) and Node.js technology.

The subject of the study is the use of a progressive web application to automate tax accounting.

The aim of the study is to develop a PWA application for tax accounting using Node.js that will facilitate the automation of the collection, storage, and analysis of tax information.

The methods of research are object-oriented programming, modular testing, user interface design methods, and API construction methods using the REST protocol and databases.

The results of the study indicate that a progressive web application for tax accounting has been developed and implemented. The application provides convenience for users, a high level of automation for processing tax data.

ЗМІСТ

Завдання на кваліфікаційну роботу 2
Реферат 4
Summary 5
Скорочення та умовні познаки 7
Вступ 8
1 Аналіз сучасних технологій автоматизації обліку податків 10
1.1 Переваги та недоліки існуючих систем ведення обліку 10
1.2 Особливості впровадження та використання pwa-застосунків 12
1.3 Аналіз вимог pwa-застосунку 15
2 Проєктування архітектури pwa-застосунку 18
2.1 Обґрунтування вибору засобів розробки 18
2.2 Розробка архітектури та структури 21
2.3 Проєктування інтерфейсу 25
2.4 Проєктування бази даних та серверної частини 27
2.5 Офлайн-режим та кешування даних 31
2.6 Безпека 33
3 Реалізація та тестування pwa-застосунку 35
3.1 Реалізації інтерфейсу користувача 35
3.2 Реалізації серверної логіки 43
3.3 Тестування функціональності 48
Висновки 55
Перелік посилань 57

СКОРОЧЕННЯ ТА УМОВНІ ПОЗНАКИ

JSON	JavaScript Object Notation
ORM	Object Relational Mapping
PWA	Progressive web app
REST	Representational state transfer
SPA	Single page application
URL	Uniform Resource Locator

ВСТУП

В епоху цифрових технологій автоматизація процесів у бізнесі є ключовим елементом для забезпечення ефективності та прозорості. Одним із таких важливих процесів є ведення податкового обліку, яке вимагає точності та відповідності законодавству. Ведення податкової звітності може бути складним завданням, особливо для малого та середнього бізнесу, де відсутність доступу до професійних бухгалтерів може ускладнювати виконання всіх необхідних вимог. Тому виникає необхідність у розробці простих та доступних цифрових інструментів, які дозволять автоматизувати цей процес.

У сучасному світі вебзастосунки стали важливим інструментом для вирішення різних завдань, а прогресивні вебзастосунки (PWA) поєднують у собі зручність мобільних застосунків і доступність вебплатформ. Прогресивний вебзастосунок дозволяє користувачам отримувати доступ до функцій навіть без постійного підключення до Інтернету, що робить його ідеальним інструментом для підприємців, які часто працюють у різних умовах. Саме тому розробка PWA для обліку податків із використанням Node.js є актуальним і перспективним напрямом дослідження.

Мета – розробка PWA-застосунку для ведення обліку податків з використанням Node.js, що дозволить автоматизувати процеси збору, зберігання та аналізу податкової інформації

Досягнення поставленої мети передбачає реалізацію наступних завдань:

- проаналізувати сучасні технології обліку податків, визначити їх переваги та недоліки;
- спроєктувати архітектуру прогресивного вебзастосунку;
- реалізувати серверну частину застосунку з використанням Node.js;
- створити клієнтську частину з інтерактивним інтерфейсом;
- провести тестування вебзастосунку та оцінити його ефективність.

Об’єкт дослідження – процес автоматизації обліку податків та його оптимізація за допомогою прогресивних вебзастосунків (PWA) та технології Node.js.

Предмет дослідження – використання прогресивного вебзастосунку для автоматизації податкового обліку.

Методи дослідження – об’єктно-орієнтоване програмування, методи проєктування користувацьких інтерфейсів, методи побудови API з використанням REST-протоколу та баз даних.

Для досягнення результату було обрано мову TypeScript, яка дозволяє реалізувати серверну та клієнтську частини, забезпечуючи ефективну взаємодію між ними. Для зберігання інформації використовувалась база даних PostgreSQL, що зберігає дані у вигляді таблиць та зв’язків між ними, забезпечуючи масштабованість і зручність роботи з великими обсягами інформації.

Основні положення і результати кваліфікаційної роботи доповідались, обговорювалися та знайшли схвалення на П’ятнадцятій Всеукраїнській, Двадцять другій регіональній науковій конференції молодих дослідників «Актуальні проблеми математики та інформатики» (Запоріжжя, 25-26 квітня 2024 р.), за результатом якої опубліковано тези.

Кваліфікаційна робота складається зі вступу, трьох розділів, висновку, переліку посилань. Загальний обсяг роботи становить 58 сторінок. У переліку посилань 21 використане джерело.

Перший розділ присвячений дослідженню існуючих систем ведення обліку, проаналізовано основні переваги та недоліки. У другому розділі наведено обґрунтування вибору інструментів розробки та спроєктовано загальну архітектуру та структуру. В третьому розділі описано реалізацію прогресивного вебзастосунку та проведення тестування реалізованого функціоналу.

1 АНАЛІЗ СУЧАСНИХ ТЕХНОЛОГІЙ АВТОМАТИЗАЦІЇ ОБЛІКУ ПОДАТКІВ

1.1 Переваги та недоліки існуючих систем ведення обліку

Ведення податкового обліку є невід'ємною частиною діяльності будь-якого підприємства та індивідуального підприємця. Традиційно цей процес характеризується складністю, трудомісткістю та значними фінансовими витратами на залучення професійних бухгалтерів. Проте розвиток інформаційних технологій відкриває нові можливості для автоматизації податкового обліку, що дозволяє суттєво скоротити витрати часу та ресурсів, а також мінімізувати ризик виникнення помилок.

Автоматизація податкового обліку за допомогою спеціалізованого програмного забезпечення забезпечує низку переваг. Зокрема, вона спрощує процес підготовки звітності, зменшує ймовірність помилок при розрахунку податків, забезпечує надійне зберігання даних в електронному форматі та дозволяє отримувати детальні аналітичні звіти про фінансовий стан підприємства. У результаті, підприємства отримують можливість ефективніше управляти своїми фінансами та приймати обґрунтовані управлінські рішення.

Український ринок програмного забезпечення для автоматизації податкового обліку пропонує широкий спектр. Від потужних комплексних систем, таких як BAS, які охоплюють усі аспекти бухгалтерського та податкового обліку, до спеціалізованих програм, наприклад, M.E.Doc, що орієнтовані на електронний документообіг та звітність. Крім того, стрімко розвивається сегмент хмарних сервісів, які забезпечують гнучкий та доступний інструмент для ведення обліку [1].

У таблиці 1.1 зазначено основні переваги та недоліки найбільш популярних систем ведення обліку.

Таблиця 1.1 – Порівняння існуючих застосунків у сфері обліку

Назва технології	**Опис**	**Переваги**	**Недоліки**
BAS Автоматизація підприємства	Одна з найпоширеніших систем в Україні.	Висока функціональність. Широке використання.	Висока вартість ліцензії. Складність використання. Вимога до налаштування та супроводження.
M.E.Doc	Програмне забезпечення для електронної звітності.	Зручність у відправленні звітів. Можливість електронного підпису.	Обмежена функціональність для комплексного обліку та управління фінансами. Висока вартість користування.
Бухгалтерія онлайн	Хмарний сервіс для ведення обліку.	Зручний доступ до інформації з будь-якого пристрою та місця.	Обмежена функціональність порівняно з локальними програмами. Залежність від стабільного інтернет-з’єднання.

З огляду на вищезазначені проблеми, існує нагальна потреба в нових рішеннях, які б поєднували доступність, простоту використання та високу функціональність. Прогресивний вебзастосунок (PWA) для ведення податкового обліку може стати оптимальним рішенням. Розглянемо основні переваги.

Інтуїтивний інтерфейс: завдяки використанню сучасних технологій, таких як React, інтерфейс буде зрозумілим навіть для непрофесійних користувачів.

Доступність: PWA забезпечить доступність застосунку з будь-якого пристрою, незалежно від стабільності інтернет-з'єднання.

Широкий функціонал: включаючи автоматичне заповнення податкових декларацій, управління витратами та доходами, а також інтеграцію з іншими системами управління бізнесом.

Вартість: застосунок буде доступний за помірну плату, що робить його привабливим для малого бізнесу та індивідуальних підприємців.

Таким чином, автоматизація податкового обліку за допомогою прогресивного вебзастосунку може стати важливим кроком у підвищенні ефективності бізнесу в Україні, зниженні витрат та мінімізації ризиків, пов'язаних з людськими помилками та змінами в законодавстві.

1.2 Особливості впровадження та використання PWA-застосунків

Прогресивна вебпрограма (PWA) – це програма, яка створена з використанням технологій вебплатформи, але забезпечує таку ж взаємодію з користувачем, як і програма для певної платформи [2]. Це відносно нова технологія, яка набула широкої популярності в 2015 році.

Progressive Web Apps володіють функціями, які надають користувачам переваги, подібні до тих, що доступні в нативних додатках.

Прогресивний web-застосунок, отримує можливість працювати як web-сайт, в тому числі і в режимі офлайн, при повній втраті зв'язку з всесвітньою мережею інтернет. А також за потреби і в режимі нативного застосунку, тобто завантажуватись на пристрій, з своєю особистою назвою та іконкою на робочому столі, і відкриватись в своєму особистому вікні. Мати при цьому високу швидкодію за рахунок багатопоточності та стратегій кешування.

Окрім того, PWA підтримують можливість надсилання push-повідомлень, що дозволяє тримати користувачів у курсі нових подій навіть тоді, коли програма не відкрита.

У таблиці 1.2 наведено порівняння PWA з односторінковими програмами (SPA) і системними програмами, щоб краще зрозуміти їх функції та особливості.

Таблиця 1.2 – Порівняння PWA з SPA і системними програмами

Особливість	**Прогресивна вебпрограма (PWA)**	**Односторінкова програма (SPA)**	**Системна програма**
Розгортання	Веббраузер	Веббраузер	Play Store, App Store
Кросплатформна сумісність	Наявна	Наявна	Відсутня
Інсталювання	Наявне	Відсутнє	магазин додатків
Офлайн-доступ	Наявний	Відсутній	Обмежений
Push-повідомлення	Наявні	Відсутні	Наявні
Приклади	Pinterest, Starbucks, Twitter Lite	Gmail, Google Maps, Facebook	WhatsApp, Spotify, Pokemon Go

PWA-застосунки побудовані на стандартних вебтехнологіях – HTML, CSS та JavaScript, і мають три основні компоненти [3]. Розглянемо їх детальніше.

Клієнтська частина (frontend) – це інтерфейс, що працює у браузері користувача. Створений на HTML, CSS та JavaScript, він забезпечує візуальне подання, а також логіку взаємодії з користувачем. Клієнтська частина може бути побудована на будь-якому сучасному фреймворку (React, Vue, Angular тощо).

Серверна частина (backend) – відповідає за обробку запитів та передавання даних до клієнта. Бекенд можна реалізувати будь-якою мовою, наприклад PHP, Node.js або Python.

Файл маніфесту – JSON-файл з основними налаштуваннями програми (назва, іконка, колірна схема), що дозволяє встановлювати PWA на головний екран пристрою та працювати як нативний застосунок.

При запуску застосунку завантажується HTML, CSS та JavaScript-код, після чого ініціюється Service Worker – фоновий процес у браузері, який відповідає за кешування і підтримку офлайн-режиму. Він дозволяє заздалегідь завантажувати та зберігати ресурси, що дозволяє застосунку працювати навіть без підключення до Інтернету. Для зберігання даних використовується IndexedDB – об’єктне сховище, що підтримує збереження структурованих даних.

PWA працює за протоколом HTTPS для захисту даних між клієнтом та сервером, що також є необхідною умовою для використання Service Worker та доступу до браузерних API, таких як геолокація.

PWA широко використовуються великими компаніями, такими як Uber, Pinterest та Starbucks, адже вони пропонують швидкий та зручний доступ до сервісів без необхідності встановлювати додатки. Розглянемо основні переваги.

Підвищення конверсії – користувачі отримують швидкий доступ до сервісів, що може збільшити ймовірність покупки.

Економія на розробці – один код для всіх платформ, що скорочує витрати на створення окремих додатків.

Спрощення оновлень – оновлення відбуваються автоматично, без потреби завантажувати нові версії з App Store.

Кросплатформеність – доступ до аудиторії на iOS, Android та інших пристроях.

1.3 Аналіз вимог PWA-застосунку

Процес розробки програмного забезпечення потребує ретельного аналізу вимог, який є фундаментальним етапом для створення якісного, зручного та ефективного застосунку. Вимоги визначають набір функцій, які повинен виконувати майбутній продукт, а також встановлюють критерії продуктивності, безпеки, доступності та сумісності з іншими системами.

У контексті розробки PWA-застосунку для обліку податків, вимоги мають особливе значення, адже застосунок буде використовуватися для роботи з чутливою фінансовою інформацією, яка вимагає надійного захисту, точності розрахунків та можливості доступу в режимі офлайн. Загалом можна виокремити такі категорії вимог: функціональні, нефункціональні, інтеграційні, вимоги інтерфейсу.

Розглянемо **функціональні вимоги.** Для забезпечення базової функціональності PWA-застосунку з обліку податків, необхідно розглянути основні аспекти.

Реєстрація та автентифікація користувачів: застосунок має підтримувати систему реєстрації та автентифікації для захисту персональних даних користувачів, таких як інформація про податки. Використання JWT (JSON Web Tokens) для захищеного доступу до приватних даних.

Облік доходів та витрат: користувачі повинні мати можливість додавати дані про доходи, витрати та автоматично розраховувати податки. Це потребує гнучкої структури даних для зберігання записів, пов’язаних з фінансами, і алгоритмів для їх обробки.

Розрахунок податків: автоматизоване обчислення податків на основі наданих даних (доходів, витрат тощо) залежно від встановлених параметрів, таких як відсоток податку. Потрібно передбачити налаштування параметрів розрахунку податків, що змінюються залежно від законодавчих норм.

Історія операцій та звітність: зберігання історії фінансових операцій користувача та створення звітів за обраний період (щомісячні, щоквартальні,

щорічні звіти). Звіти мають бути доступні для перегляду та експорту у форматі PDF або Excel (рис. 1.1).

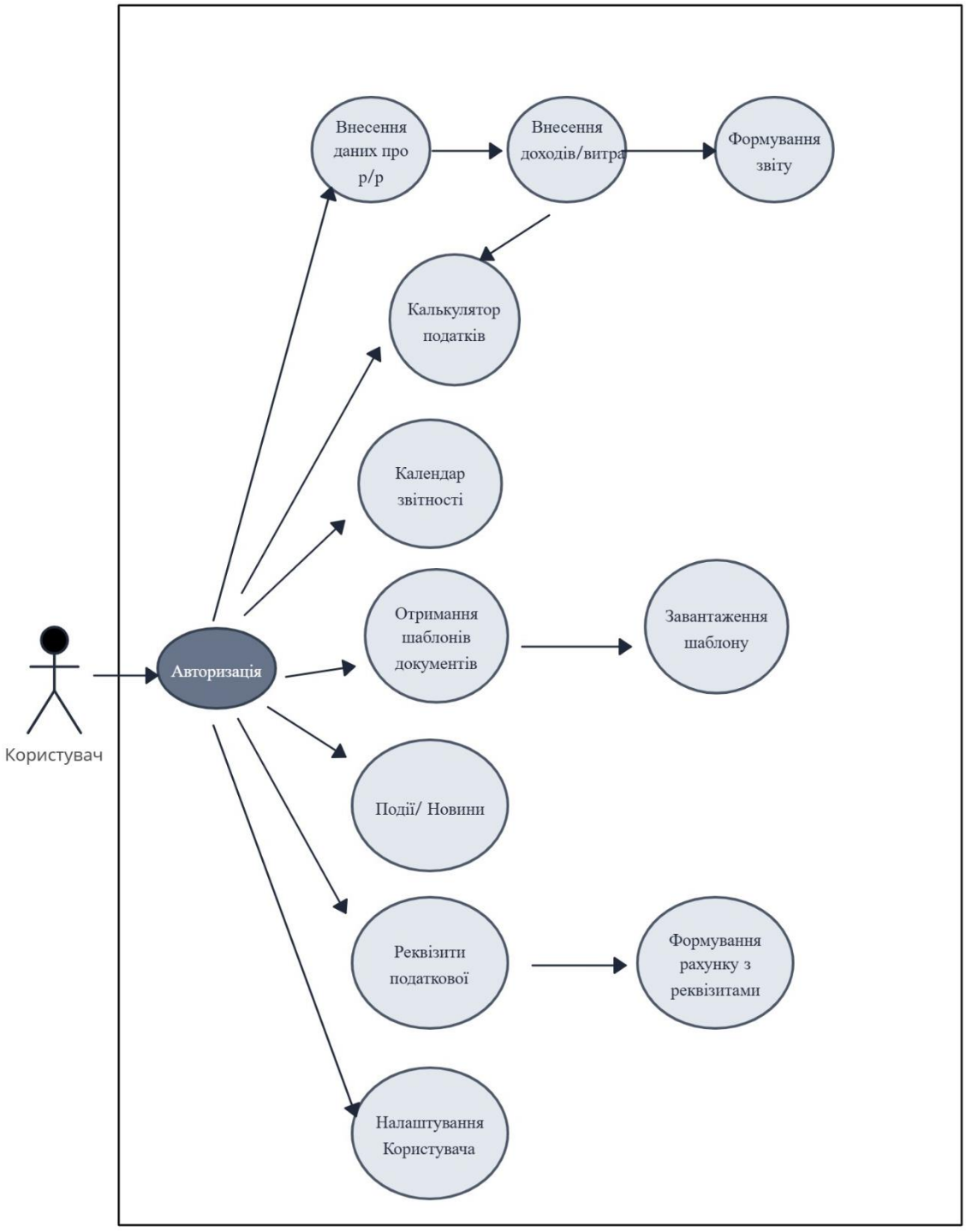


Рисунок 1.1 – Діаграма прецедентів системи

Розглянемо **нефункціональні вимоги.**

Продуктивність: PWA має працювати стабільно як у режимі онлайн, так і офлайн завдяки кешуванню ресурсів та даних за допомогою Service Worker.

Це гарантує доступність критичних функцій (перегляд історії, доступ до останніх введених даних) навіть при відсутності підключення до Інтернету.

Безпека: застосунок має відповідати стандартам безпеки, таким як захищене зберігання та обробка даних користувачів, захист від CSRF та XSS-атак, налаштування HTTPS.

Мобільність та адаптивність: застосунок має бути зручним для використання на різних платформах (настільні ПК, смартфони, планшети), з адаптивним дизайном для різних розмірів екранів.

Оновлюваність: PWA-застосунок повинен легко оновлюватися без необхідності перевстановлення. Для цього потрібно забезпечити автоматичне оновлення Service Worker, щоб користувачі завжди мали доступ до актуальних даних та функціональності.

Інтеграційні вимоги.

Інтеграція з API податкових сервісів: для актуалізації податкових ставок або інформації про податкові знижки можливе підключення до API податкових органів чи фінансових сервісів.

Інтеграція з системами обліку: для автоматичного імпорту або експорту даних.

Вимоги інтерфейсу.

Зручність у використанні: інтерфейс має бути інтуїтивно зрозумілим для користувачів з різним рівнем технічних знань, з простими і зрозумілими формами для введення даних.

Сповіщення та нагадування: реалізація сповіщень про терміни подання звітів або сплату податків для користувачів, що сприятиме підвищенню зручності роботи із застосунком.

2 ПРОЄКТУВАННЯ АРХІТЕКТУРИ PWA-ЗАСТОСУНКУ

2.1 Обґрунтування вибору засобів розробки

Першим кроком в розробці Progressive Web Apps є рішення, щодо фреймворку розробки інтерфейсу користувача. Незважаючи на те, що є кілька варіантів, бажано оцінити плюси і мінуси кожної платформи Progressive Web Apps, перш ніж вибрати найкращий.

Найпопулярніші фреймворки для прогресивних вебдодатків є AngularJS, React та VueJS [4].

У таблиці 2.1 наведено переваги та недоліки кожного з фреймворків.

Таблиця 2.1 – Порівняння фреймворків для створення інтерфейсів

Характеристика	**AngularJS**	**React**	**VueJS**
Спільнота програмістів	Середня	Велика	Мала
Простота вивчення та поріг входу	Високий	Низький	Середній
Швидкість розробки	Висока	Висока	Висока
Мова розробки	Typescript	Typescript	Typescript

Детально розглянувши 3 варіанти фреймворків досить важко обрати один, так як різниця між ними не суттєва, але React має найбільшу спільноту програмістів, через що полегшується пошук та виправлення помилок, також з переваг можна виділити простоту та зрозумілість написання програмного коду.

Для вибору серверної частини застосунку було проведено аналіз найбільш популярних мов програмування разом з фреймворками та узагальнено в таблиці 2.2 [5].

Головними перевагами Node.js є написання користувацької та серверної частин на одній мові, що значно прискорює розробку, а використання

асинхронної моделі обробки запитів в Node.js забезпечує високу продуктивність та масштабованість. Це особливо важливо для додатків, які обробляють велику кількість запитів одночасно [6].

Таблиця 2.2 – Порівняння серверних фреймворків

№ з/п	**Характеристика**	**NestJS**	**Spring Boot**	**Laravel**
1	Продуктивність	Висока завдяки асинхронній моделі	Висока завдяки Java	Висока, але синхронна
2	Масштабованість	Дуже висока	Дуже висока	Висока
3	Легкість у навчанні	Помірна, вимагає знання JavaScript/TypeScript	Висока складність через Java	Помірна, вимагає знання PHP
4	Гнучкість	Дуже висока	Висока	Помірна
5	Підтримка спільноти	Дуже велика	Велика	Велика
6	Безпека	Висока, завдяки сучасним підходам	Висока	Висока

Вибір бази даних є критично важливим аспектом при розробці будь-якого застосунку, особливо такого, як PWA-застосунок для ведення обліку податків. Розглянемо основні причини, чому було обрано PostgreSQL для цього проєкту.

Розширені функціональні можливості. PostgreSQL є однією з найбільш функціонально багатих реляційних баз даних.

Висока продуктивність. PostgreSQL забезпечує високу продуктивність при роботі з великими обсягами даних. Завдяки ефективним алгоритмам індексації та оптимізації запитів, ця база даних може швидко обробляти запити навіть у випадку великої кількості одночасних звернень.

Надійність та консистентність даних. Для обліку податків важливо забезпечити високу надійність та консистентність даних.

Масштабованість. PostgreSQL легко масштабується як вертикально (додаванням ресурсів до одного сервера), так і горизонтально (шардінг та реплікація). Це дозволяє базі даних ефективно рости разом з бізнесом, забезпечуючи необхідну потужність для обробки збільшеного обсягу даних та користувачів.

Підтримка JSON і JSONB. PostgreSQL підтримує типи даних JSON і JSONB, що дозволяє ефективно зберігати та обробляти напівструктуровані дані.

Безкоштовність та відкритий вихідний код. PostgreSQL є безкоштовним і має відкритий вихідний код. Це означає відсутність витрат на ліцензії та можливість модифікувати систему відповідно до специфічних потреб проєкту.

Активна спільнота та розвиток. PostgreSQL має одну з найбільших та найактивніших спільнот серед реляційних баз даних. Це забезпечує швидке вирішення проблем, регулярні оновлення та багатий набір додаткових інструментів та розширень, що полегшують розробку та адміністрування.

Таким чином, PostgreSQL є оптимальним вибором для бази даних у PWA-застосунку для ведення обліку податків завдяки своїм розширеним можливостям, високій продуктивності, надійності, масштабованості, та активній підтримці спільноти [7].

Весь стек використаних технологій узагальнено в таблиці 2.3.

Таблиця 2.3 – Стек технологій

Назва технології	**Опис**	**Застосування**
Node.js	Програмна платформа, заснована на двигуні V8, що перетворює JavaScript з вузькоспеціалізованої мови на мову загального призначення.	Створення серверної частини сервісу

Продовження табл. 2.3

Назва технології	**Опис**	**Застосування**
NestJS	Прогресивний фреймворк для створення масштабованих серверних додатків	Створення серверної частини сервісу
NGINX	HTTP-сервер і зворотний проксі-сервер	Сервер статичного контенту, балансування навантаження
ReactJS	JavaScript-фреймворк з відкритим вихідним кодом для створення інтерфейсів користувача	Інтерактивні вебкомпоненти
PostgreSQL	Реляційна база даних	Зберігання інформації
S3 Minio	Об’єктне сховище	Сховище для документів та файлів
Jest	Фреймворк тестування	Автотестування
OpenAPI	Фреймворк для опису вебсервісів REST	Документація інтерфейсів обміну

2.2 Розробка архітектури та структури

Для розробки PWA-застосунку був обраний сучасний стек, що включає Node.js, NestJS, React, Redux Toolkit, PostgreSQL та Prisma.

Застосунок буде побудований за принципом клієнт-сервер з використанням односторінкової архітектури (SPA). Клієнтська частина (фронтенд) взаємодіє з серверною частиною через RESTful API для отримання та відправки даних [8].

Організація взаємозв’язків між компонентами системи зображено на рисунку 2.1. Взаємодія компонентів організована таким чином, щоб

забезпечити швидкий обмін даними між фронтендом та бекендом, ефективне кешування, а також стабільну роботу застосунку в умовах нестабільного або відсутнього інтернет-з'єднання.

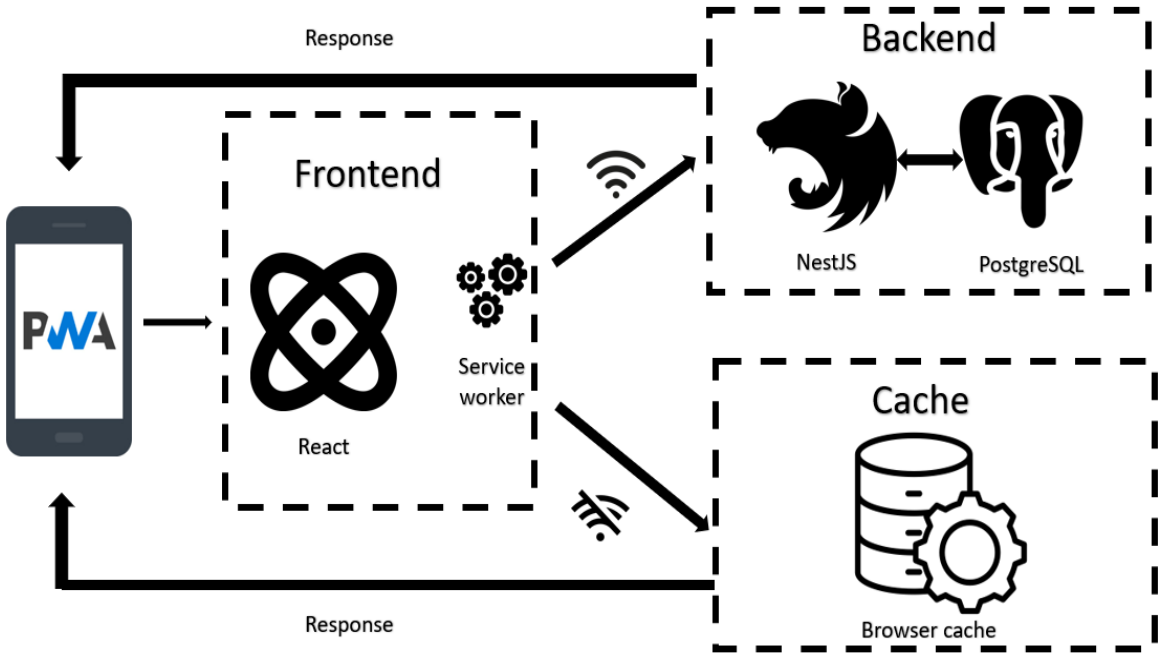


Рисунок 2.1 – Архітектурна взаємодія компонентів системи

Загальна схема роботи системи полягає у наступному: коли користувач здійснює запит із мобільного пристрою, PWA-застосунок на фронтенді надсилає запит до бекенду через API. Якщо пристрій користувача має нестабільне або відсутнє з'єднання з мережею, Service Worker зберігає запит і виконує його, коли з'єднання відновлюється. У відповідь сервер надсилає дані, які кешуються у браузері для покращення продуктивності та зменшення навантаження на мережу.

При розробці архітектури застосунку важливо дотримуватися низки фундаментальних принципів SOLID, які допоможуть створити стійкий, масштабований і підтримуваний продукт [9].

Принцип єдиного обов'язку (Single Responsibility Principle, SRP) дозволить зменшити складність і підвищити підтримуваність.

Моделі даних повинні відповідати лише за структурування і збереження даних (наприклад, модель для транзакцій або звітів).

Компоненти UI повинні бути відповідальними лише за відображення даних, без втручання в логіку.

Сервіси повинні виконувати конкретні бізнес-операції або обробку запитів, не переплітаючи їх з іншими аспектами програми.

Принцип відкритості/закритості (Open/Closed Principle, OCP) – класи повинні бути відкриті для розширення, але закриті для модифікацій. Це досягається за допомогою наслідування та інтерфейсів. Наприклад, при додаванні нового типу податку, треба створити новий клас чи компонент, який розширює існуючі класи, без потреби зміні коду або система звітів може бути розширена новими типами звітів, не змінюючи базової логіки генерації.

Принцип підстановки Ліскова (Liskov Substitution Principle, LSP) – підкласи повинні бути замінні їхніми базовими класами, не порушуючи коректності програми. Наприклад, при розширенні класу для податкових декларацій, при створенні спеціалізованих класів для окремих видів податків, ці нові класи повинні зберігати очікувану поведінку (методи для валідації, розрахунку податку) без порушення логіки.

Принцип розділення інтерфейсу (Interface Segregation Principle, ISP) **–** великі інтерфейси не повинні використовуватись клієнтами, що не потребують їхніх методів. Це дозволяє уникнути надмірних залежностей у проєкті. Наприклад, створення інтерфейсів для різних частин бізнес-логіки, наприклад, окремо для взаємодії з платіжними системами, для обробки звітів тощо.

Принцип інверсії залежностей (Dependency Inversion Principle, DIP) – високорівневі модулі не повинні залежати від низькорівневих модулів, обидва повинні залежати від абстракцій (інтерфейсів). Всі сервіси, репозиторії та інші компоненти повинні бути інжектовані в класи, а не створюватися всередині них [10].

Правильно організована структура папок також є критично важливим аспектом розробки. Структура проєкту забезпечує чітке розмежування відповідальності між різними компонентами – від інтерфейсу користувача до серверної частини та бази даних.

Структура клієнтської частини є досить поширеною для React-застосунків і дозволяє організувати код максимально логічно та ефективно. На рисунку 2.2 зображено структуру, що забезпечить зручний поділ на модулі для зберігання компонентів, логіки, статичних ресурсів та конфігурацій.

```
/react                          # Клієнтська частина
├── /src
│   ├── /assets                 # Папка для статичних ресурсів
│   ├── /components             # Фронтенд-компонентикартки.
│   ├── /features               # Модулі для реалізації окремих функцій
│   ├── /http                   # Конфігурація HTTP-запитів
│   ├── /layout                 # Компоненти макетів
│   ├── /pages                  # Сторінки застосунку
│   ├── /services               # Логіка та сервіси для обробки даних
│   ├── /store                  # Стейт-менеджер, який організує глобальний стан застосунку
│   ├── serviceWorker.ts        # Сервіс-воркер
│   └── App.ts                  # Основний компонент
├── /public                     # Статичні файли,, включаючи favicon, HTML-шаблон, PWA-маніфест
└── package.json                # Файл конфігурації проекту, що містить перелік залежностей
```

Рисунок 2.2 – Структура папок клієнтської частини

Структура серверної частини з NestJS та Prisma чітко розділяє функціональні модулі, що полегшує підтримку й розширення. На рисунку 2.3 зображено структуру директорій, кожна директорія містить логіку для окремих аспектів системи – авторизації, управління рахунками, податкових звітів і шаблонів документів.

```
/nest                           # Серверна частина (NestJS)
├── /prizma                     # ORM для взаємодії з БД
│   ├── /migrations             # Міграції БД
├── /src                        # Серверна частина (Node.js)
│   ├── /accounts               # Рахунки користувача
│   ├── /auth                   # Авторизація
│   ├── /document-templates     # Шаблони документів
│   ├── /tax-events             # Події
│   ├── /tax-info               # Інформація про податки
│   ├── /tax-office-requisites  # Реквізити податкових служб
│   ├── /tax-reports            # Звіти
│   ├── /transactions           # Операції
│   └── /user-tax-info          # Податки користувача
├── package.json                # Файл конфігурації проекту
└── tsconfig.json               # Файл конфігурації Typescript
```

Рисунок 2.3 – Структура папок серверної частини

ORM Prisma організовує взаємодію з базою даних, включаючи міграції, що спрощує керування моделями. Завдяки такому розподілу забезпечується масштабованість проєкту, оскільки кожна папка відповідає певній бізнес-логіці, а конфігураційні файли визначають основні налаштування TypeScript та залежностей.

2.3 Проєктування інтерфейсу

Інтерфейс PWA-застосунку має складатися з декількох основних елементів (рис. 2.4):

– головна сторінка;
– меню навігації;
– футер – блок в нижній частині сторінки;
– хедер – блок у верхній частині сторінки.

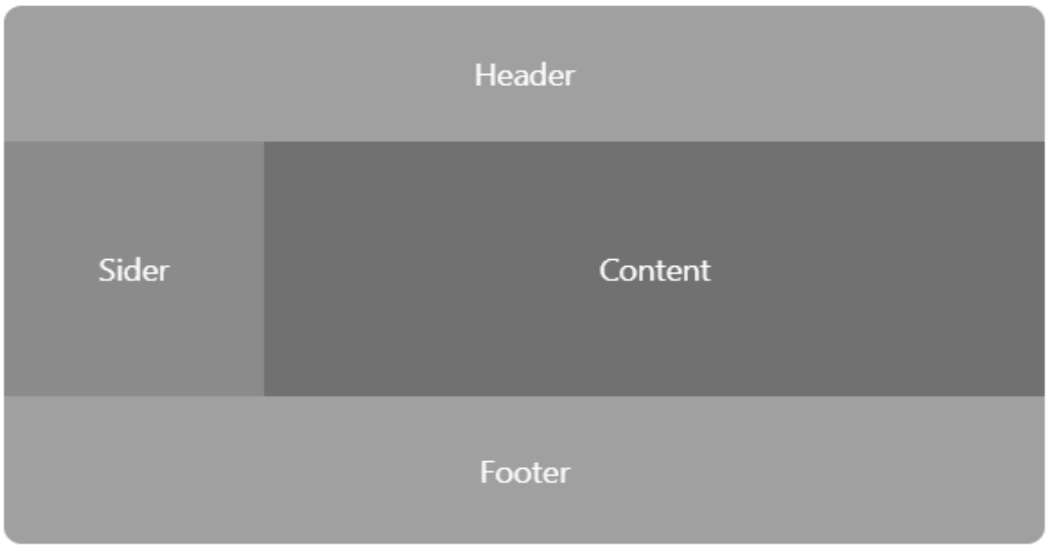


Рисунок 2.4 – Макет інтерфейсу

На головній сторінці користувач має можливість швидко переглянути основну інформацію про податкові зобов’язання, дату наступного платежу, а також швидкий доступ до основних функцій застосунку (додавання доходів, витрат, генерація звітів тощо).

Меню має бути розміщене у верхній частині або зліва (графічний інтерфейс зліва або у вигляді гамбургер-меню на мобільних пристроях). Це дозволить користувачеві швидко переміщатися між основними розділами застосунку.

Основні пункти меню:

– головна сторінка;
– реквізити;
– календар;
– звітність;
– калькулятор;
– платежі;
– рахунки;
– налаштування.

На головній сторінки мають бути податкові події та актуальні новини.

В інтерфейсі «Реквізити» мають бути контактна інформація по податкові служби та реквізити для оплати податків.

Інтерфейс «Звітність» повинен включати форму для автоматичного розрахунку податків на основі введених даних. Розрахунки повинні відображати реальну інформацію відповідно до податкового законодавства, з можливістю автоматичного оновлення та коригування даних. Для зручності користувачів передбачено створення різноманітних звітів.

Інтерфейс «Календар» повинен показати всі поточні дати подання звітності та суми до сплати, що стосуються податкової групи користувача.

В інтерфейсі «Платежі» користувач повинен мати можливість додавати доходи та витрати за допомогою форм. Важливою частиною є валідація введених даних, щоб уникнути помилок при розрахунках.

Інтерфейс «Налаштування користувача» передбачає налаштування параметрів профілю (наприклад, вибір податкової категорії, місця проживання для визначення ставок податку, налаштування способів оплати).

2.4 Проєктування бази даних та серверної частини

Для реалізації серверної частини застосунку обрано Node.js з фреймворком NestJS. Це дозволяє ефективно обробляти HTTP запити та побудувати REST API для взаємодії з клієнтською частиною.

NestJS підтримує модульну структуру, це дозволить розділити функціонал окремо на модулі що значно спрощує підтримку і розширення додатка.

Основні маршрути, які реалізують взаємодію з сервером:

– /api/accounts – керування рахунками користувачів;
– /api/auth – реєстрація та аутентифікація користувачів;
– /api/transactions – додавання та перегляд транзакцій;
– /api/tax-events – обробка податкових подій;
– /api/tax-reports – створення та перегляд податкових звітів;
– /api/tax-calculator – розрахунок податків на основі доходів і витрат;
– /api/document-templates – отримання бланків документів;
– /api/tax-info – отримання податкової інформації;
– /api/tax-office-requisites – отримання реквізитів податкової служби.

Кожен маршрут пов’язаний з контролером, який відповідає за логіку обробки запитів та взаємодії з базою даних.

Для автоматизації розгортання серверної частини застосунку обрано GitLab CI/CD. Необхідно налаштувати конвеєр в декілька етапів:

а) встановлення залежностей та компіляція TypeScript-коду в JavaScript;
б) запуск юніт-тестів і перевірка покриття коду;
в) автоматичне копіювання коду на сервер.

Після успішного виконання конвеєра, застосунок розгортається на сервері за допомогою Nginx, який виступає як зворотній проксі.

Система використовуватиме реляційну базу даних PostgreSQL, оскільки вона надійна та забезпечує ефективну роботу з великим обсягом даних.

Для кожної таблиці в базі даних створюється модель за допомогою бібліотеки Prisma (для PostgreSQL).

Загалом база даних вміщує в собі 10 таблиць та 1 сервісну таблицю Prisma:

– 1 таблиця для даних користувача;
– 5 таблиць за транзації, рахунки, звітність та податкова інформація;
– по одній таблиці для виводу даних в інтерфейсах, таких як календар подій, калькулятор, реквізити та шаблони документів.

Загальна діаграма БД зображена на рисунку 2.5.

Розглянемо основні моделі даних.

Модель користувача (User):

– id (integer, PK) – унікальний ідентифікатор користувача;
– username (string) – ім’я користувача (унікальне);
– password (string) – хешований пароль;
– hashedRT (string?) – хеш Refresh Token;
– created_at (timestamp) – дата реєстрації;
– updated_at (timestamp) – дата останнього оновлення.

Модель податкової інформації (TaxInfo):

– id (integer, PK) – унікальний ідентифікатор;
– tax_name (string) – назва податку;
– tax_rate (float) – податкова ставка;
– created_at (timestamp) – дата створення;
– updated_at (timestamp) – дата останнього оновлення.

Модель реквізитів податкового органу (TaxOfficeRequisite):

– id (integer, PK) – унікальний ідентифікатор реквізиту;
– office_name (string) – назва податкового органу;
– requisites (string) – реквізити (рахунок, платіжні дані);
– created_at (timestamp) – дата створення;
– updated_at (timestamp) – дата останнього оновлення.

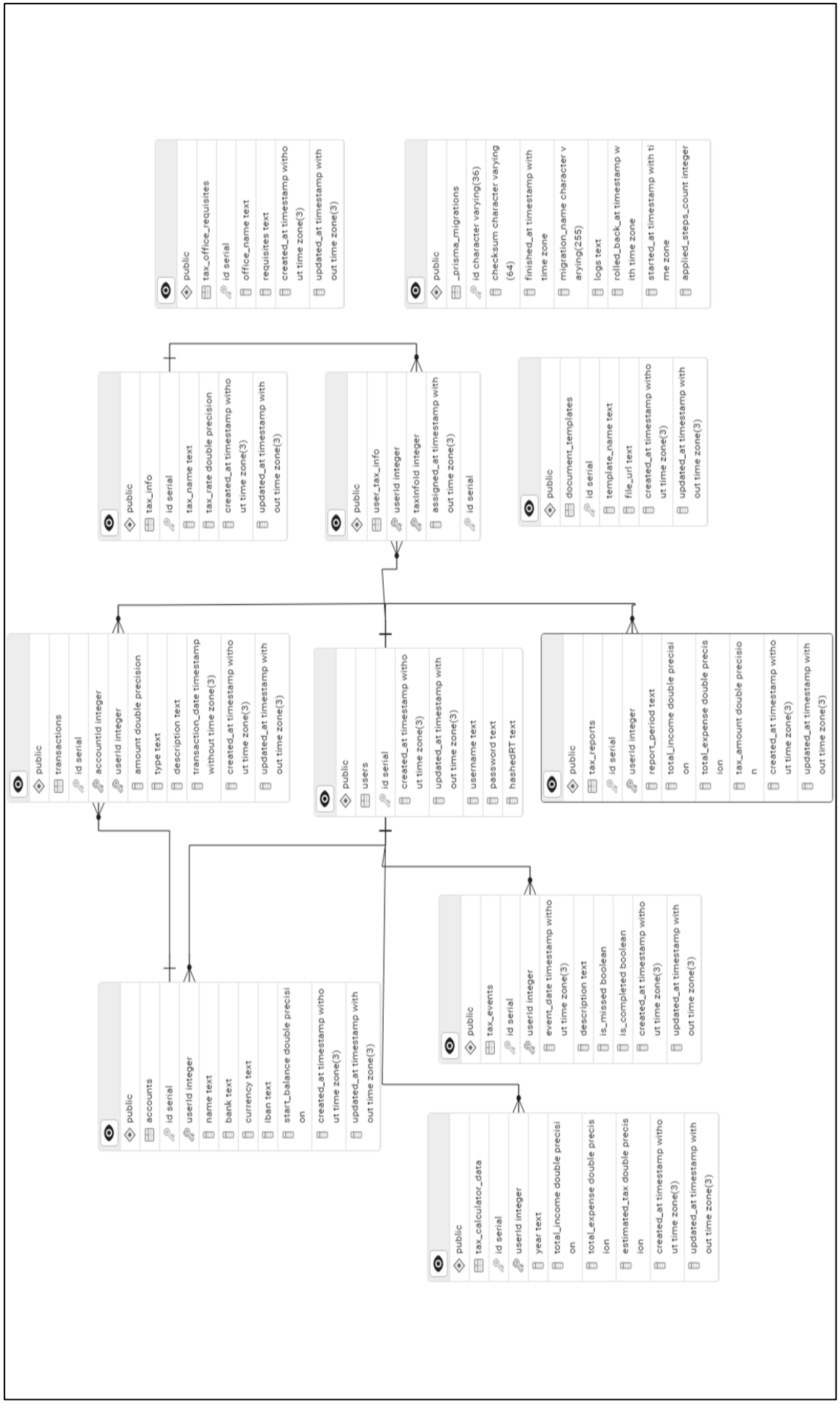


Рисунок 2.5 – Схема бази даних PostgreSQL

Модель транзакцій (Transaction):

– id (integer, PK) – унікальний ідентифікатор транзакції;
– accountId (integer, FK) – зв’язок з рахунком;
– userId (integer, FK) – зв’язок з користувачем;
– amount (float) – сума транзакції;
– type (string) – тип транзакції (доходи або витрати);
– description (string?) – опис транзакції;
– transaction_date (timestamp) – дата транзакції;
– created_at (timestamp) – дата створення;
– updated_at (timestamp) – дата останнього оновлення.

Модель податкових подій (TaxEvent):

– id (integer, PK) – унікальний ідентифікатор податкової події;
– userId (integer, FK) – зв’язок з користувачем;
– event_date (timestamp) – дата події;
– description (string) – опис події;
– is_missed (boolean) – чи пропущена подія;
– is_completed (boolean) – чи завершена подія;
– created_at (timestamp) – дата створення;
– updated_at (timestamp) – дата останнього оновлення.

Модель податкових звітів (TaxReport):

– id (integer, PK) – унікальний ідентифікатор податкового звіту;
– userId (integer, FK) – зв’язок з користувачем;
– report_period (string) – період звіту;
– total_income (float) – загальний дохід;
– total_expense (float) – загальні витрати;
– tax_amount (float) – сума податку;
– created_at (timestamp) – дата створення;
– updated_at (timestamp) – дата останнього оновлення.

Модель податкових калькуляцій (TaxCalculatorData):

– id (integer, PK) – унікальний ідентифікатор;

– userId (integer, FK) – зв’язок з користувачем;
– year (string) – рік для розрахунку податків;
– total_income (float) – загальний дохід для розрахунку податків;
– total_expense (float) – загальні витрати;
– estimated_tax (float) – оцінковий розмір податку;
– created_at (timestamp) – дата створення;
– updated_at (timestamp) – дата останнього оновлення.

Модель шаблонів документів (DocumentTemplate):

– id (integer, PK) – унікальний ідентифікатор документа;
– template_name (string) – назва шаблону;
– file_url (string) – URL файлу шаблону;
– created_at (timestamp) – дата створення;
– updated_at (timestamp) – дата останнього оновлення.

Користувач може мати багато рахунків та транзакцій, тому між таблицями User і Account, User і Transaction встановлені зв’язки типу один до багатьох.

Один користувач може мати багато податкових подій та податкових звітів, тому між таблицями User і TaxEvent, User і TaxReport також зв’язки типу один до багатьох.

Кожен податковий звіт може мати багато податкових подій, зв’язок між таблицями TaxReport та TaxEvent типу один до багатьох.

2.5 Офлайн-режим та кешування даних

Щоб забезпечити доступність застосунку навіть за відсутності інтернет-з’єднання, важливо правильно спроєктувати офлайн-режим і механізм кешування даних. Це дозволить користувачам працювати з інформацією та зберігати зміни, які синхронізуються з сервером, коли підключення відновиться.

Для ефективної реалізації кешування даних підходить IndexedDB або LocalStorage для зберігання великих обсягів інформації на стороні клієнта. Service Worker надає можливість управляти кешем для статичних ресурсів, таких як HTML, CSS, JavaScript, шрифти та зображення. В якості стратегії кешування обрано «Network-First» (рис. 2.6) – при наявності підключення застосунок спочатку звертається до сервера, а в разі невдачі використовує кешовану версію [11].

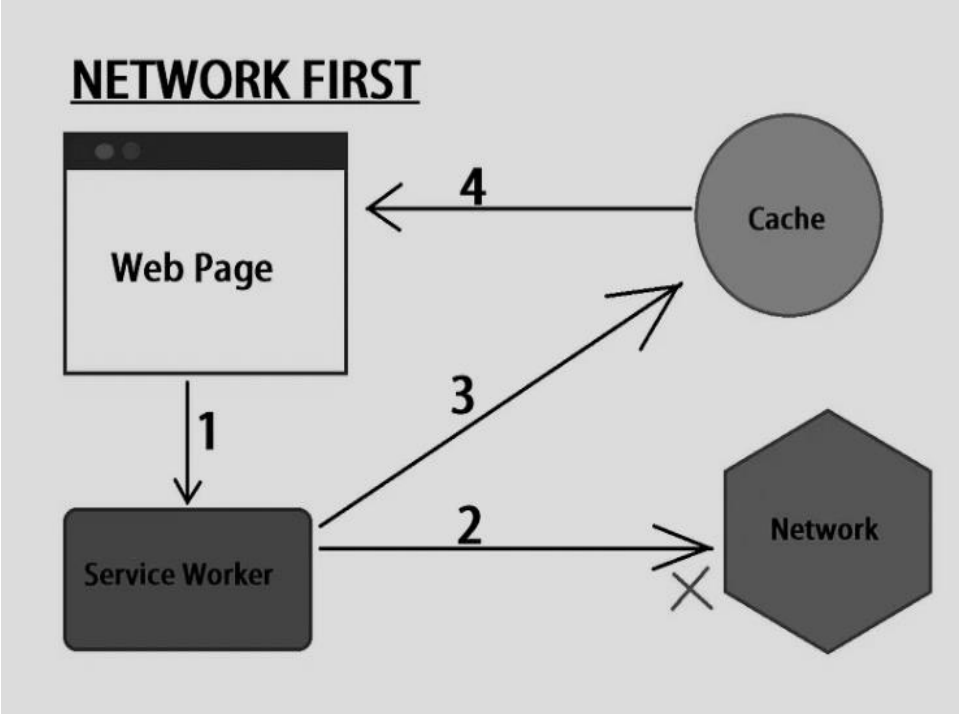


Рисунок 2.6 – Схема стратегії «Network-First»

Транзакції записуються в IndexedDB з автоматичною інкрементацією ключів, що дозволяє унікально ідентифікувати кожну операцію. Це дає можливість користувачам переглядати та оновлювати дані про витрати, доходи чи податкові зобов’язання безпосередньо у застосунку.

Для автоматичної синхронізації даних після відновлення з’єднання використовується Background Sync API. Це означає, що користувач може працювати офлайн, а застосунок зберігатиме всі зміни в IndexedDB [12]. Щойно з’єднання з мережею відновиться, накопичені дані автоматично відправляються на сервер без потреби додаткових дій з боку користувача.

Ця функціональність особливо корисна для користувачів, які вносять багато транзакцій протягом дня, коли доступ до інтернету може бути нестабільним. Всі їхні зміни будуть автоматично передані на сервер при першій можливості, а у випадку помилки дані залишаться локально для повторної спроби синхронізації.

У разі зміни стану інтернет-з'єднання застосунок відстежує та повідомляє про них користувача. Якщо інтернет відсутній, користувач отримує відповідне повідомлення про роботу в офлайн-режимі, а після відновлення з'єднання відображається інформація про оновлення даних на сервері. Це зручність для бухгалтерів та підприємців, які завжди бачать, чи є їхні дані актуальними.

2.6 Безпека

Аутентифікація у застосунку реалізована за допомогою токенів доступу (Access Token) та токенів оновлення (Refresh Token) (рис. 2.7) [13].

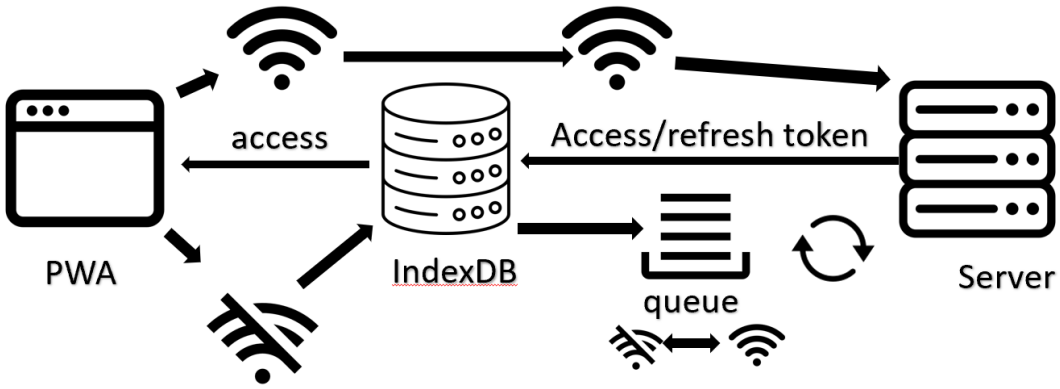


Рисунок 2.7 – Процес авторизації користувача

Процес авторизації користувача в PWA-застосунку починається з введення облікових даних, які надсилаються на сервер для перевірки. Після успішної авторизації сервер повертає два токени: access token для доступу до ресурсів і refresh token для оновлення сесії, коли строк дії access token

закінчиться. Обидва токени зберігаються на стороні клієнта, зазвичай у безпечному сховищі IndexedDB.

Коли користувач успішно авторизується в режимі онлайн, застосунок може зберегти інформацію про статус авторизації в локальному сховищі або кеші. Інформація яка зберігається:

– відмітка часу останнього успішного входу;
– відомості про доступні дані для офлайн-режиму.

При кожному запиті до захищених ресурсів застосунок додає access token до заголовків запиту. Якщо сервер визначає, що access token вже недійсний, він повертає помилку, яка свідчить про необхідність оновлення токена. У цьому випадку застосунок використовує збережений refresh token для отримання нового access token. Після успішного оновлення токени замінюються у сховищі, і запит повторюється з новим access token.

Якщо користувач переходить у офлайн-режим, застосунок автоматично переходить на кешування запитів. Запити, які потребують доступу до серверу, зберігаються в черзі до моменту, коли з’явиться підключення до Інтернету. Застосунок відслідковує стан мережі за допомогою navigator.onLine, а також подій online та offline. Коли з’єднання відновлюється, черга запитів автоматично обробляється у порядку, в якому вони були створені. При цьому застосунок перевіряє дійсність access token, щоб забезпечити успішну синхронізацію. Якщо access token недійсний, використовується refresh token для його оновлення, після чого виконуються всі відкладені запити з черги. Такий підхід забезпечує безперервний досвід користування застосунком і дозволяє синхронізувати всі дані після відновлення зв’язку.

3 РЕАЛІЗАЦІЯ ТА ТЕСТУВАННЯ PWA-ЗАСТОСУНКУ

3.1 Реалізації інтерфейсу користувача

Реалізацію застосунку розпочинаємо із запуску середовища розробки Visual Studio Code. Спочатку необхідно створити новий проєкт і ініціалізувати його як реакт-застосунок, використовуючи команду npx create-react-app. Це автоматично створить базову структуру проєкту та встановить основні залежності, необхідні для роботи React [14].

Після цього проводиться налаштування середовища шляхом додавання специфічних залежностей, таких як бібліотеки для управління станом Redux бібліотеки для обробки HTTP-запитів Axios, а також CSS-фреймворк для стилізації інтерфейсу Ant Design. Всі версії бібліотек фіксуються в файлі package.json (рис. 3.1).

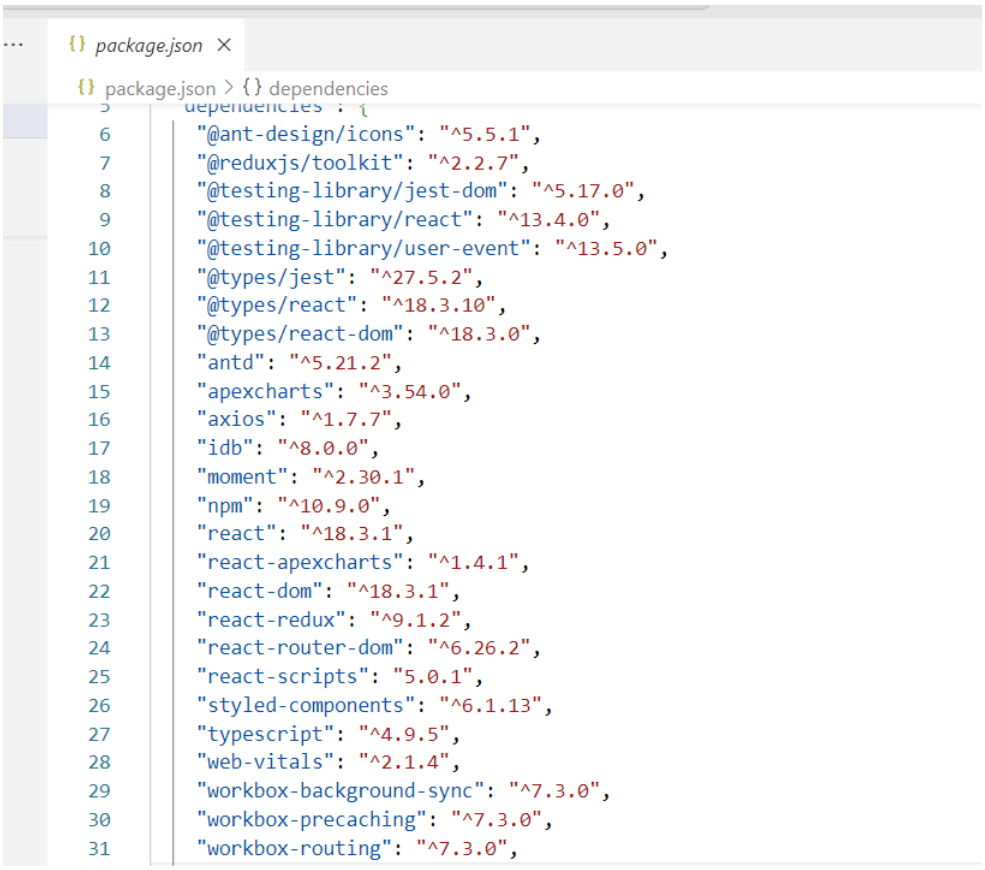


Рисунок 3.1 – Встановлені залежності проєкту

Розробку починаємо із головного файлу index.tsx який отримує id «root» html- файлу та рендерить React застосунок (рис. 3.2).

```tsx
import ReactDOM from 'react-dom/client';
import App from './App';
import reportWebVitals from './reportWebVitals';
import { Provider } from 'react-redux';
import { store } from 'store/store';
import { BrowserRouter } from 'react-router-dom'
import { registerServiceWorker } from 'serviceWorkerRegistration';

const root = ReactDOM.createRoot(
  document.getElementById('root') as HTMLElement
);
root.render(
    <Provider store={store}>
      <BrowserRouter>
        <App />
      </BrowserRouter>
    </Provider>
);
```

Рисунок 3.2 – Файл index.tsx

Далі в файлі App.tsx відбувається перенаправлення url-запитів до основних сторінок застосунку (рис. 3.3). Для того щоб перехід між сторінками відбувався без перезавантаження сторінки використовується бібліотека react-router-dom [15].

```tsx
const AppRouter: React.FC = () => {
  const isAuth = useSelector((state: RootState) => state.isAuth.isAuth)

  return (
    <ConfigProvider
      theme={{
        token: {
          colorPrimary: '#f4a725',
          borderRadius: 2,
        },
      }}>
      <Routes>
        <Route path="/" element={<Navigate to="/dashboard" replace />} />

        <Route path="/sign-in" element={isAuth ? <Navigate to="/dashboard" replace /> : <SignIn />} />
        <Route path="/sign-up" element={isAuth ? <Navigate to="/dashboard" replace /> : <SignUp />} />

        <Route element={<ProtectedRoute><Main /></ProtectedRoute>}>
          <Route path="/dashboard" element={<Dashboard />} />
          <Route path="/details" element={<Details />} />
          <Route path="/calendar" element={<Calendar />} />
          <Route path="/calculator" element={<Calculator />} />
          <Route path="/transactions" element={<Transactions />} />
          <Route path="/account" element={<Account />} />
        </Route>
        <Route path="*" element={<NotFound />} />
      </Routes>


    </ConfigProvider>
```

Рисунок 3.3 – Файл App.tsx

Взаємодія користувача із застосунком починається із сторінки реєстрації або входу (рис. 3.4).

```
<Title className="mb-15">Увійти в кабінет</Title>
<Title className="font-regular text-muted" level={5}>
  Введіть електронну пошту та пароль
</Title>
<Form
  onFinish={onFinish}
  onFinishFailed={onFinishFailed}
  layout="vertical"
  className="row-col"
>
  <Form.Item
    className="username"
    label="Електронна пошта"
    name="email"
    rules={[
      {
        required: true,
        message: "Будь ласка введіть електронну пошту!
      },
    ]}
  >
    <Input placeholder="user@gmail.com" />
  </Form.Item>

  <Form.Item
    label="Password"
```

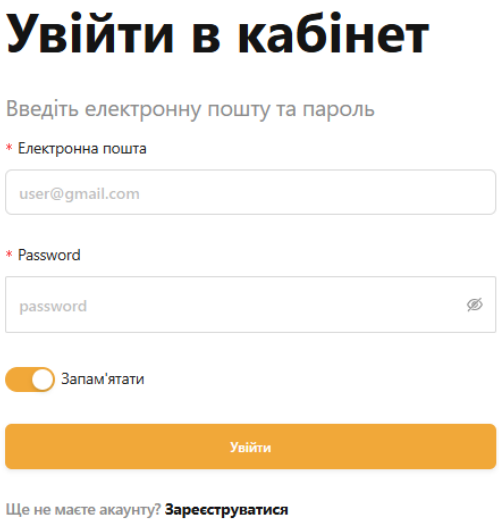


Рисунок 3.4 – Реалізація сторінки входу

Після заповнення форми потрібно надіслати запит з даними до сервера для перевірки користувача. Для цього необхідно налаштувати бібліотеку Axios для виконання HTTP-запитів (рис. 3.5).

```
const axiosInstance = axios.create({
  baseURL: process.env.REACT_APP_BASE_URL,
  withCredentials: true,
});

axiosInstance.interceptors.request.use(
  async (config) => {
    const token = await getToken();
    if (token) {
      config.headers.Authorization = `Bearer ${token}`
    }

    return config;
  }
);

axiosInstance.interceptors.response.use(
  (response) => response,
  (error) => Promise.reject(error)
);
```

Рисунок 3.5 – Модуль формування запиту

Форма із логіном та паролем звертається до сервісу авторизації бекенду по API та у разі успішної відповіді отримує пару токенів (див. рис. 3.6).

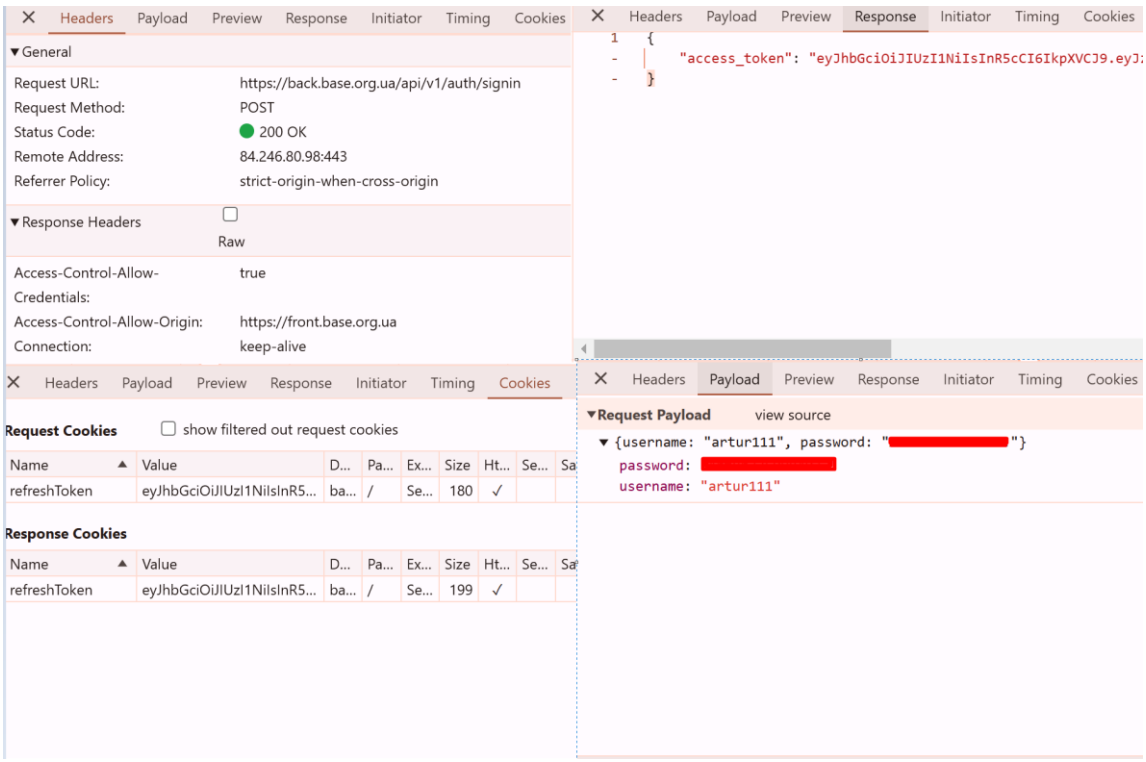


Рисунок 3.6 – Інструменти розробника в браузері

Токен зберігаємо в сховище IndexedDB для подальшої перевірки чи авторизований користувач (рис. 3.7).

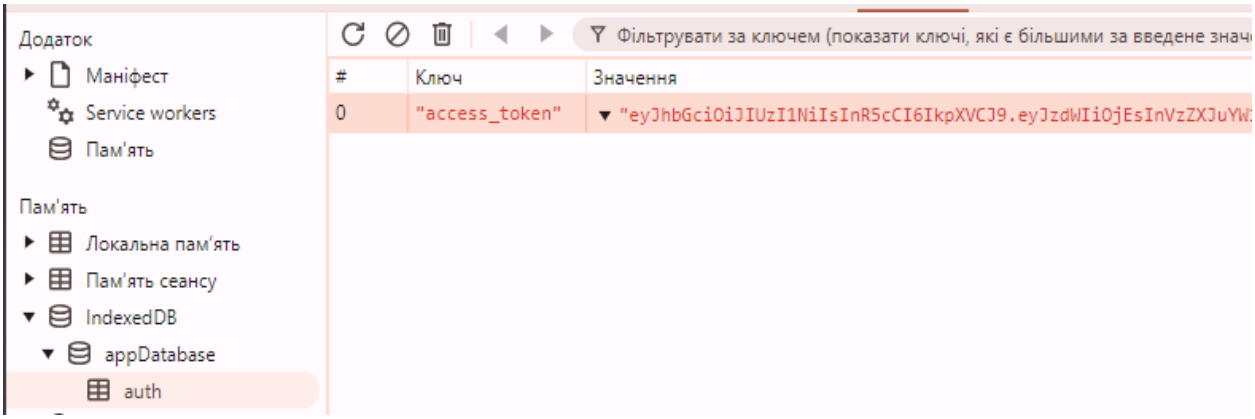


Рисунок 3.7 – Сховище IndexedDB

Після авторизації користувач потрапляє на головну сторінку. Кожна сторінка має стандартний шаблон який підключається як окрема компонента до кожної сторінки, вона включає в себе бокове та верхнє меню. Бокове меню включає в себе всі головні розділи застосунку для зручної навігації між сторінками сервісу (рис. 3.8).

```jsx
import { Menu } from 'antd';
import { NavLink, useLocation } from 'react-router-dom';
import logo from 'assets/images/logo-4.jpeg';
import { DashboardIcon, DetailsIcon, TransactionIcon, ProfileIcon, CalculatorIcon, CalendarIcon, Acco

function Sidenav({ color }:any) {
  const { pathname } = useLocation();
  const page = pathname.replace('/', '');

  const menuItems = [
    { key: '1', path: '/dashboard', label: 'Головна', icon: <DashboardIcon color={color} /> },
    { key: '2', path: '/details', label: 'Реквізити', icon: <DetailsIcon color={color} /> },
    { key: '3', path: '/calendar', label: 'Календар', icon: <CalendarIcon color={color} /> },
    { key: '4', path: '/calculator', label: 'Калькулятор', icon: <CalculatorIcon color={color} /> },
    { key: '5', path: '/transactions', label: 'Платежі', icon: <TransactionIcon color={color} /> },
    { key: '6', path: '/account', label: 'Рахунки', icon: <AccountIcon color={color} /> },
  ];

  return (
    <>
      <div className="brand">
        <img src={logo} alt="Logo" />
      </div>
      <hr />
      <Menu theme="light" mode="inline">
        {menuItems.map((item) => (
          <Menu.Item key={item.key}>
            <NavLink to={item.path}>
              <span
                className="icon"
                style={{
                  background: page === item.path.slice(1) ? color : '',
                }}
```

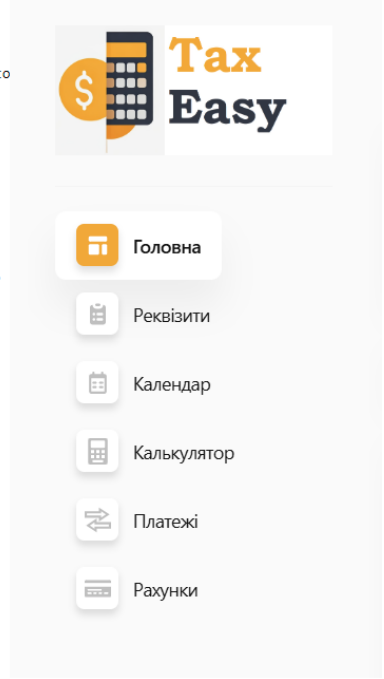


Рисунок 3.8 – Бокова навігація застосунку

Для інсталяції застосунку на робочий стіл необхідно створити файл manifest.json (рис. 3.9), який виконує роль конфігураційного файлу і служить своєрідним паспортом вебпрограми.

manifest.json

public > {} manifest.json > [ ] icons > {} 0

Artur, в прошлом месяце | 1 author (Artur)

```json
{
  "short_name": "Tax Easy",
  "name": "Tax Easy",
  "icons": [
    {
      "src": "favicon.ico",
      "sizes": "64x64 32x32 24x24 16x16",
      "type": "image/x-icon"
    },
    {
      "src": "logo192.png",
      "type": "image/png",
      "sizes": "192x192"
    },
    {
      "src": "logo512.png",
      "type": "image/png",
      "sizes": "512x512"
    }
  ],
  "start_url": ".",
  "display": "standalone",
  "theme_color": "#000000",
  "background_color": "#ffffff"
}
```

Рисунок 3.9 – Файл manifest.json

Цей файл містить метадані про додаток, що надають браузеру необхідну інформацію для коректного відображення вашого застосунку на пристрої користувача. Зокрема, manifest.json дозволяє визначити такі ключові характеристики, як назва додатку, короткий опис, набір піктограм різних розмірів для відображення на різних пристроях, а також основні кольори, що будуть використані в інтерфейсі (наприклад, кольори заголовків чи фонові відтінки) [16].

У результаті отримуємо іконку застосунку на робочому столі (рис. 3.10).

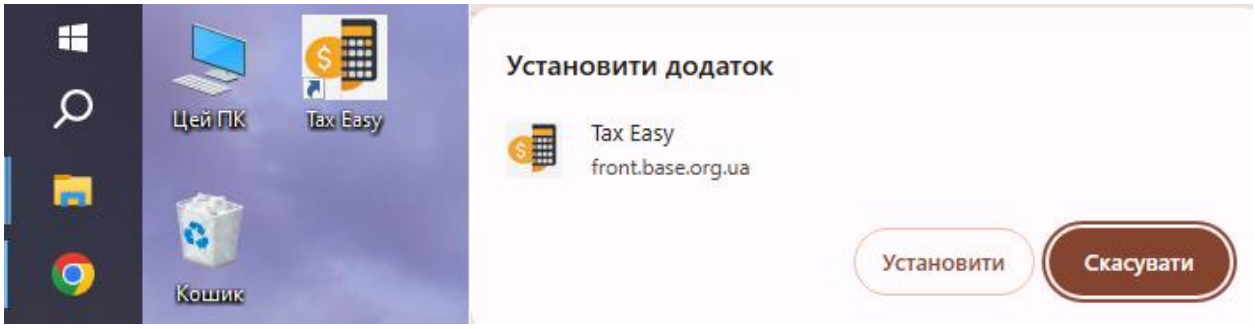


Рисунок 3.10 – Інсталяція додатку на робочий стіл

Для організації кешування застосунку необхідно створити файл service-worker.ts та зареєструвати його в головному файлі (рис. 3.11).

```
import { Workbox } from 'workbox-window';

export function registerServiceWorker() {
  if ('serviceWorker' in navigator) {
    const wb = new Workbox('/service-worker.js');

    wb.addEventListener('installed', (event) => {
      if (event.isUpdate) {
        console.log('Новий Service Worker встановлений.');
      }
    });

    wb.register();
  }
}
```

Рисунок 3.11 – Реєстрація service-worker.ts

Однією з бібліотек, яка значно полегшує роботу з Service Worker, є Workbox [17]. Workbox надає зручні інтерфейси для кешування,

маршрутизації запитів і реалізації фонового синхронного виконання операцій. Workbox має декілька модулів, що вирішує певне завдання і може бути використаний як спільно з іншими модулями, і самостійно (рис. 3.12).

```
    "workbox-background-sync": "^7.3.0",
    "workbox-precaching": "^7.3.0",
    "workbox-routing": "^7.3.0",
    "workbox-strategies": "^7.3.0",
    "workbox-window": "^7.3.0"
  },
```

Рисунок 3.12 – Модулі workbox

За допомогою модуля workbox-background-sync налаштовуємо фонову синхронізацію, що дозволяє виконувати мережні запити в режимі офлайн (рис. 3.13). Коли користувач втрачає з'єднання з мережею, Workbox автоматично зберігає запити в спеціальну чергу, яка зберігається у IndexedDB. Ця черга організована у вигляді транзакцій, кожна з яких містить деталі запиту, включаючи URL, метод, заголовки та тіло запиту. Після відновлення мережевого з'єднання Workbox автоматично витягує запити з черги та відправляє їх на сервер.

```
declare const self: ServiceWorkerGlobalScope;

precacheAndRoute(self.__WB_MANIFEST || []);

const bgSyncPlugin = new BackgroundSyncPlugin('myQueueName', {
  maxRetentionTime: 24 * 60,
})

registerRoute(
  /https:\/\/back\.base\.org\.ua\//,
  new NetworkOnly({
    plugins: [bgSyncPlugin],
  }),
  'POST'
);
```

Рисунок 3.13 – Налаштування модуля workbox-background-sync

Для обробки динамічних GET-запитів використовуємо стратегію Network First (рис. 3.14). Це підходить для запитів, де важливо мати останні дані, але необхідний fallback у випадку, якщо мережі немає.

```
registerRoute(
  ({ url }) => url.origin === 'https://back.base.org.ua',
  new NetworkFirst({
    cacheName: 'get-api-cache',
    networkTimeoutSeconds: 10,
    plugins: [
      {
        cacheWillUpdate: async ({ response }) => {
          if (response && response.status < 400) {
            return response;
          }
          return null;
        },
      },
    ],
  })
);
```

Рисунок 3.14 – Обробка динамічних GET-запитів

Для запитів, які рідко змінюються (зображення, шрифти тощо), також можна додати Network First. Використовуємо кешування статичних ресурсів (рис. 3.15). Кеші можуть накопичувати старі ресурси. Щоб цього уникнути додамо логіку очищення кешу (рис. 3.16).

```
registerRoute(
  ({ request }) => ['script', 'style', 'image', 'font'].includes(request.destination), // JS, CSS, зображення, шрифти
  new NetworkFirst({
    cacheName: 'static-resources-cache',
    networkTimeoutSeconds: 5,
    plugins: [
      {
        cacheWillUpdate: async ({ response }) => {
          if (response && response.status === 200) {
            return response;
          }
          return null;
        },
      },
    ],
  })
);
```

Рисунок 3.15 – Обробка статичних файлів

```
self.addEventListener('activate', (event) => {
  const cacheWhitelist = [ 'get-api-cache', 'static-resources-cache'];

  event.waitUntil(
    caches.keys().then((cacheNames) =>
      Promise.all(
        cacheNames.map((cacheName) => {
          if (!cacheWhitelist.includes(cacheName)) {
            console.log('застарілий кеш:', cacheName);
            return caches.delete(cacheName);
          }
        })
      )
    )
  );
});
```

Рисунок 3.16 – Очищення кешу

У разі, якщо мережа недоступна і ресурс не знайдений у кеші, можна повертати заздалегідь підготовлений offline.html.

3.2 Реалізації серверної логіки

Реалізація серверної частини починається з файлу main.ts (рис. 3.17), який створює сервер для генерації RESTful API.

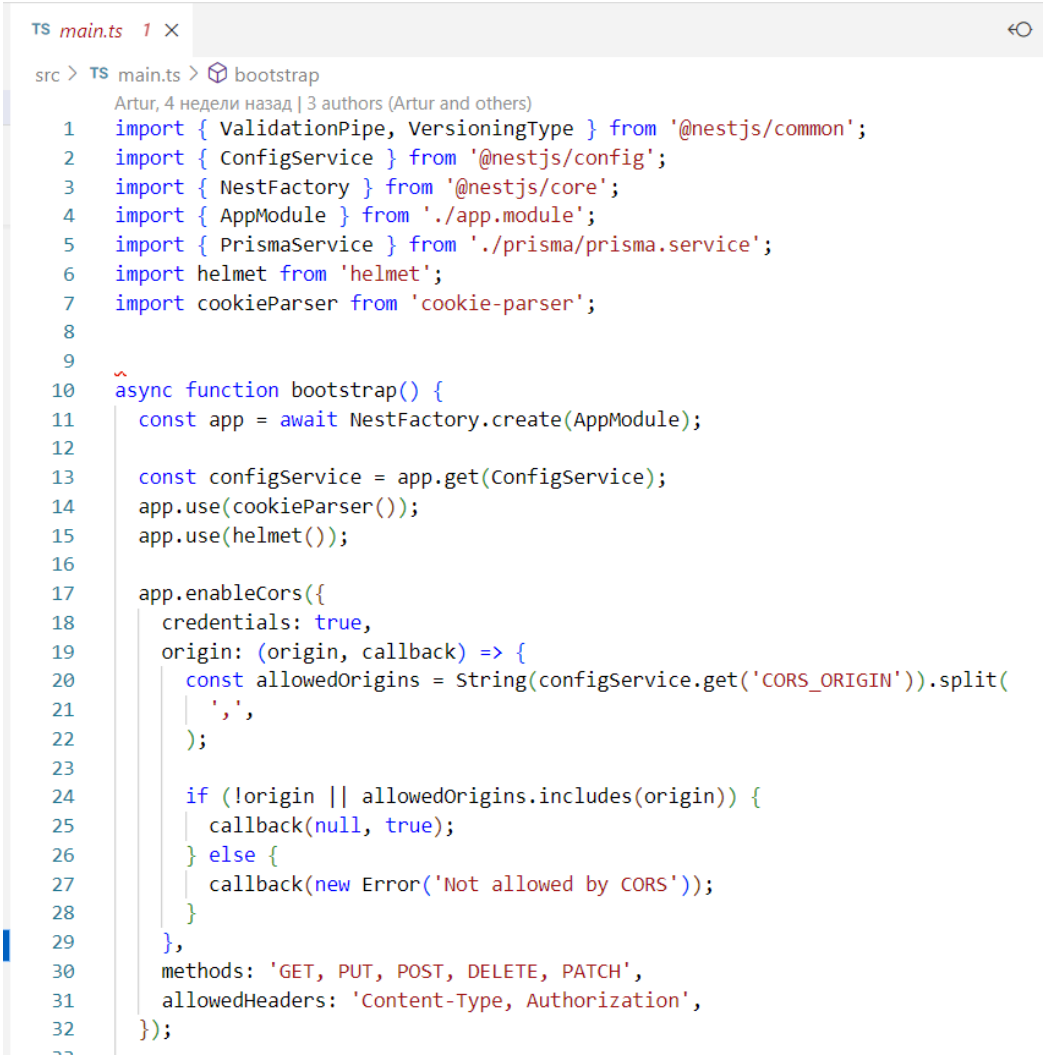


Рисунок 3.17 – Файл main.ts

У цьому файлі одразу налаштовуємо CORS-політику для забезпечення безпечного обміну HTTP-запитами з клієнтським застосунком. Далі додаємо class-validator для валідації вхідних даних і class-transformer для їхнього

перетворення. Щоб підтримувати роботу з файлами Cookie, інтегруємо cookie-parser, а для базового захисту заголовків безпеки – helmet. Це дозволяє знизити ризики XSS, захистити від клікових шахрайств і зменшити загальну вразливість API.

Далі необхідно налаштувати конфігурацію сервера у файлі app.module.ts (рис. 3.18). Завдяки цьому NestJS ініціює створення основних модулів та встановлює зв'язки між ними. У цьому ж модулі підключаються зовнішні сервіси, такі як база даних.

```
import { PrismaModule } from './prisma/prisma.module';
import { AuthModule } from './auth/auth.module';
import { APP_GUARD } from '@nestjs/core';
import { AtGuard } from './auth/guards';
import { ConfigModule } from '@nestjs/config';
import { ThrottlerGuard, ThrottlerModule } from '@nestjs/throttler';
import { AccountsModule } from './accounts/accounts.module';
import { TransactionsModule } from './transactions/transactions.module';
import { DocumentTemplatesModule } from './document-templates/document-templates.module';
import { TaxOfficeRequisitesModule } from './tax-office-requisites/tax-office-requisites.module';
import { TaxReportsModule } from './tax-reports/tax-reports.module';
import { TaxEventsModule } from './tax-events/tax-events.module';
import { TaxInfoModule } from './tax-info/tax-info.module';
import { UserTaxInfoModule } from './user-tax-info/user-tax-info.module';

You, в прошлом месяце | 2 authors (Artur and one other)
@Module({
  imports: [
    PrismaModule,
    AuthModule,
    // To enable `configModule` for all routes
    ConfigModule.forRoot({
      isGlobal: true,
    }),
    // config `RateLimiting`
    ThrottlerModule.forRoot({
      ttl: parseInt(process.env.RATE_LIMIT_TIME_TO_LIVE),
      limit: parseInt(process.env.RATE_LIMIT_MAX_NUMBER_REQUEST),
    }),
    AccountsModule,
    TransactionsModule,
    DocumentTemplatesModule,
    TaxOfficeRequisitesModule,
    TaxReportsModule,
    TaxEventsModule,
    TaxInfoModule,
    UserTaxInfoModule,
  ],

  providers: [
    // To set `ATGuard` as global guard for all routes
    { provide: APP_GUARD, useClass: AtGuard },
```

Рисунок 3.18 – Файл app.module.ts

Nest CLI дозволяє швидко створювати модулі (рис. 3.19), які допомагають структуровано організувати код і розділити функціональність застосунку. Кожен модуль у NestJS — це незалежний блок, який містить:

– контролер – обробляє вхідні запити по API, наприклад, маршрути для входу та реєстрації (у контролері перевіряється формат даних, а також відправляються запити до сервісу для виконання бізнес-логіки);

– сервіс – опрацьовує основну логіку автентифікації, як-то: перевірка пароля, генерація та оновлення JWT-токенів;

– data transfer object (DTO) – визначають структуру даних, які передаються до контролера або сервісу, наприклад, LoginDto, RegisterDto, що дозволяє стандартизувати та валідувати вхідні дані;

– guards – створюють захисні механізми, що перевіряють доступ користувача до певних маршрутів, наприклад, JwtAuthGuard перевіряє наявність дійсного токена в запиті;

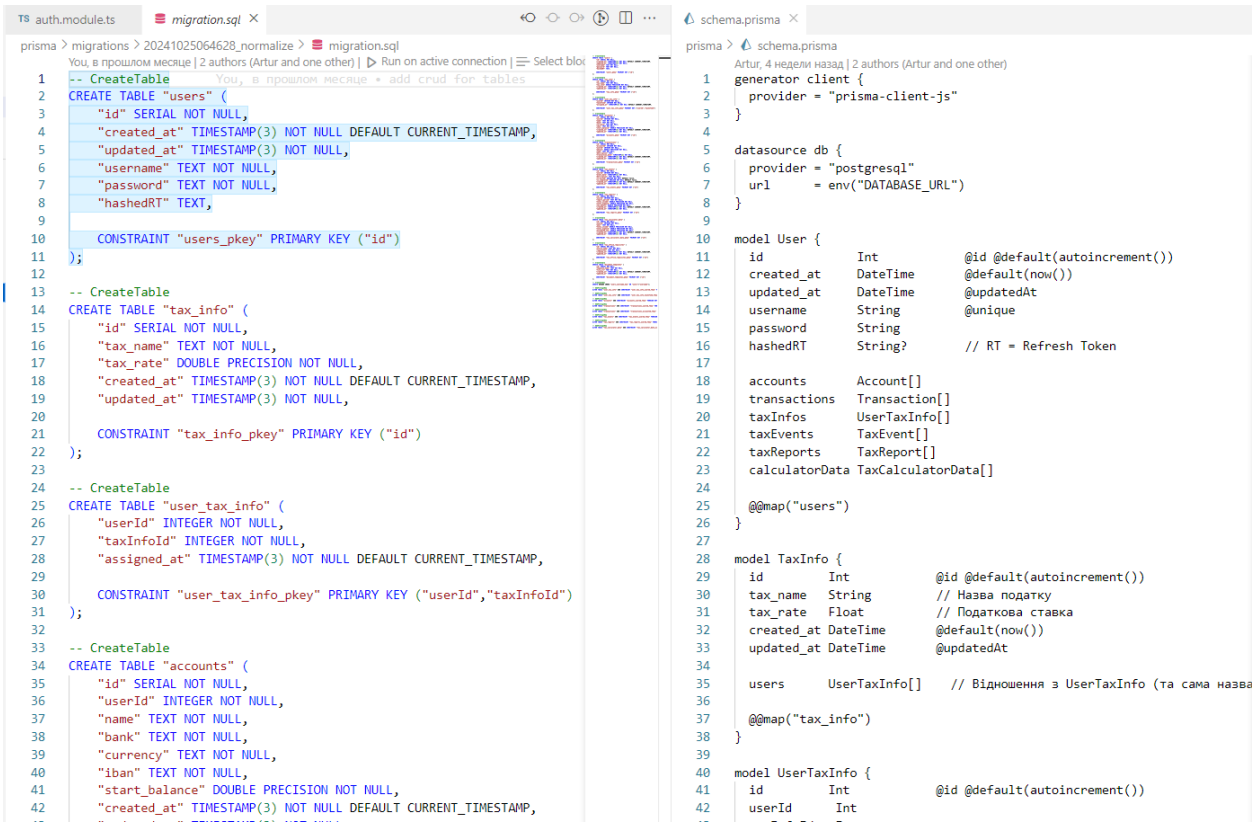
– decorators – додають додаткові метадані до методів або параметрів, як-от @User() для автоматичного отримання даних авторизованого користувача з токена.



Рисунок 3.19 – Файл auth.module.ts

У якості бази даних використано PostgreSQL, а для роботи з нею застосовуємо Prisma. У файлі schema.prisma (рис. 3.20) описано основні моделі даних, що використовуються в проєкті. При зміні моделей або схем бази даних

автоматично створюються міграції, які дозволяють синхронізувати структуру бази з кодом і підтримувати актуальність даних. Міграції генеруються за допомогою Prisma CLI та застосовуються до бази даних, що забезпечує ефективну та безпечну міграцію даних між різними середовищами.



```
-- CreateTable
CREATE TABLE "users" (
    "id" SERIAL NOT NULL,
    "created_at" TIMESTAMP(3) NOT NULL DEFAULT CURRENT_TIMESTAMP,
    "updated_at" TIMESTAMP(3) NOT NULL,
    "username" TEXT NOT NULL,
    "password" TEXT NOT NULL,
    "hashedRT" TEXT,

    CONSTRAINT "users_pkey" PRIMARY KEY ("id")
);

-- CreateTable
CREATE TABLE "tax_info" (
    "id" SERIAL NOT NULL,
    "tax_name" TEXT NOT NULL,
    "tax_rate" DOUBLE PRECISION NOT NULL,
    "created_at" TIMESTAMP(3) NOT NULL DEFAULT CURRENT_TIMESTAMP,
    "updated_at" TIMESTAMP(3) NOT NULL,

    CONSTRAINT "tax_info_pkey" PRIMARY KEY ("id")
);

-- CreateTable
CREATE TABLE "user_tax_info" (
    "userId" INTEGER NOT NULL,
    "taxInfoId" INTEGER NOT NULL,
    "assigned_at" TIMESTAMP(3) NOT NULL DEFAULT CURRENT_TIMESTAMP,

    CONSTRAINT "user_tax_info_pkey" PRIMARY KEY ("userId","taxInfoId")
);

-- CreateTable
CREATE TABLE "accounts" (
    "id" SERIAL NOT NULL,
    "userId" INTEGER NOT NULL,
    "name" TEXT NOT NULL,
    "bank" TEXT NOT NULL,
    "currency" TEXT NOT NULL,
    "iban" TEXT NOT NULL,
    "start_balance" DOUBLE PRECISION NOT NULL,
    "created_at" TIMESTAMP(3) NOT NULL DEFAULT CURRENT_TIMESTAMP,
```

```
generator client {
  provider = "prisma-client-js"
}

datasource db {
  provider = "postgresql"
  url      = env("DATABASE_URL")
}

model User {
  id             Int             @id @default(autoincrement())
  created_at     DateTime        @default(now())
  updated_at     DateTime        @updatedAt
  username       String          @unique
  password       String
  hashedRT       String?         // RT = Refresh Token

  accounts       Account[]
  transactions   Transaction[]
  taxInfos       UserTaxInfo[]
  taxEvents      TaxEvent[]
  taxReports     TaxReport[]
  calculatorData TaxCalculatorData[]

  @@map("users")
}

model TaxInfo {
  id         Int            @id @default(autoincrement())
  tax_name   String         // Назва податку
  tax_rate   Float          // Податкова ставка
  created_at DateTime       @default(now())
  updated_at DateTime       @updatedAt

  users      UserTaxInfo[]    // Відношення з UserTaxInfo (та сама назва

  @@map("tax_info")
}

model UserTaxInfo {
  id         Int            @id @default(autoincrement())
  userId      Int
```

Рисунок 3.20 – Файл schema.prizma

Авторизація реалізована за допомогою passport.js – це middleware для Node.js, яке спрощує інтеграцію авторизаційних процесів у вебдодатки. Passport.js підтримує широкий спектр зовнішніх авторизаційних сервісів, що дозволяє зручно інтегрувати його з будь-якими сучасними вебдодатками на базі NestJS [18].

Для аутентифікації користувачів використовується jsonwebtoken, що є загальноприйнятим стандартом доступу на основі JSON (описаним у RFC 7519). Це дозволяє передавати інформацію про користувача між клієнтом і сервером. Сервер генерує токен, підписує його секретним ключем, після чого передає клієнту. Надалі клієнт додає цей токен до своїх запитів для підтвердження особи [19].

Налаштування авторизації зображено на рисунку 3.21.

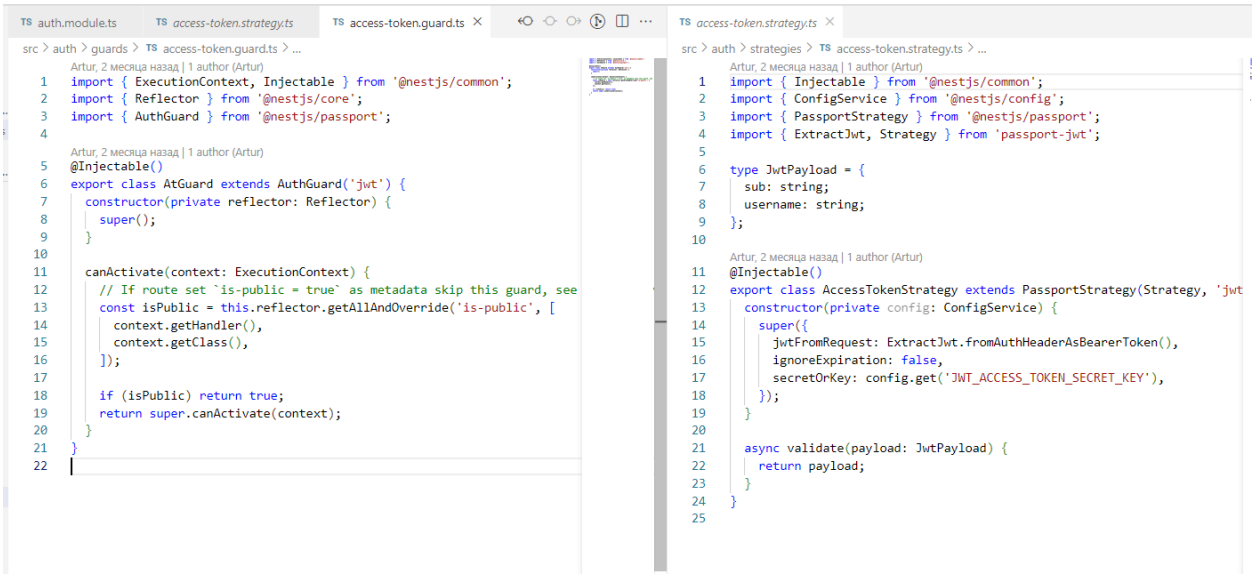


Рисунок 3.21 – Реалізація авторизації

У файлі реалізується логіка обробки вхідних запитів, спрямованих на отримання токена доступу. Ці запити спершу перевіряються на правильність структури і наявність необхідних параметрів. Після валідації запит перенаправляється до відповідного сервісу, який відповідає за подальшу обробку даних, а саме: створення токенів, їх підписання, а також збереження чи оновлення інформації в базі даних, що стосується аутентифікованих користувачів.

Для забезпечення безпеки й контролю доступу реалізовано кілька рівнів захисту, включаючи стратегії та гварди. Стратегії визначають правила, за якими обробляються запити, наприклад, обирається метод аутентифікації – jwt, local чи інші, залежно від налаштувань passport.js. Кожна стратегія обробляє запити по-своєму: наприклад, jwt стратегія перевіряє валідність токену, що надходить із запитом, і, у разі успіху, визначає роль та права користувача на подальшу взаємодію з сервісами [20].

Гварди, у свою чергу, виконують роль фільтрів для доступу до захищених маршрутів. Вони перевіряють, чи містить запит коректний токен доступу, і чи відповідає він вимогам, визначеним у стратегіях. Наприклад,

AuthGuard, налаштований на роботу з jwt стратегією, перехоплює запит і перевіряє токен на дійсність перед тим, як надати доступ до ресурсів. Якщо токен дійсний, запит пропускається далі; у протилежному випадку запит блокується з відповідним повідомленням про помилку.

Для зручності тестування документацію методів API описано в форматі JSON за допомогою технології OpenAPI [21].

3.3 Тестування функціональності

Для тестування бекенду використаємо інструмент Postman, що дозволить перевірити основні REST запити до сервера.

Тестування розпочинаємо з отримання токену (рис. 3.22).

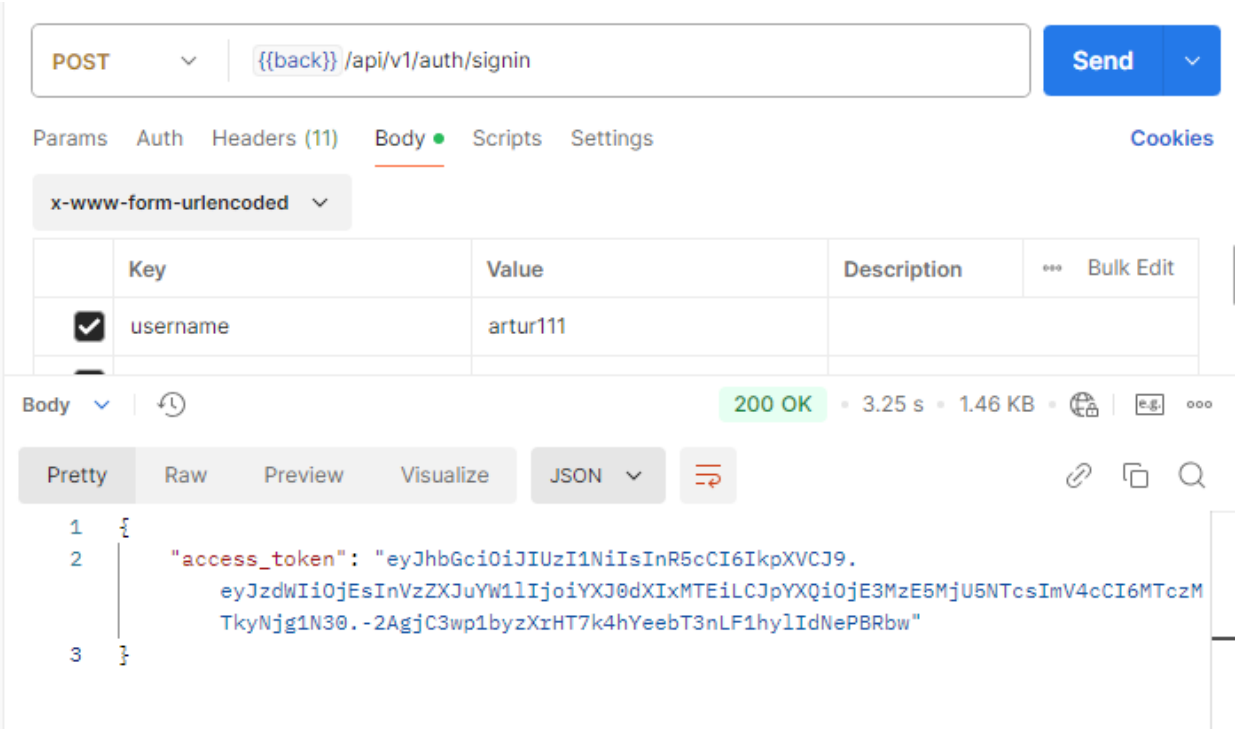


Рисунок 3.22 – Тестування входу в систему

Після отримання токену, необхідно створити рахунок користувача для прив’язки витрат чи доходів до певного рахунку (рис. 3.23).

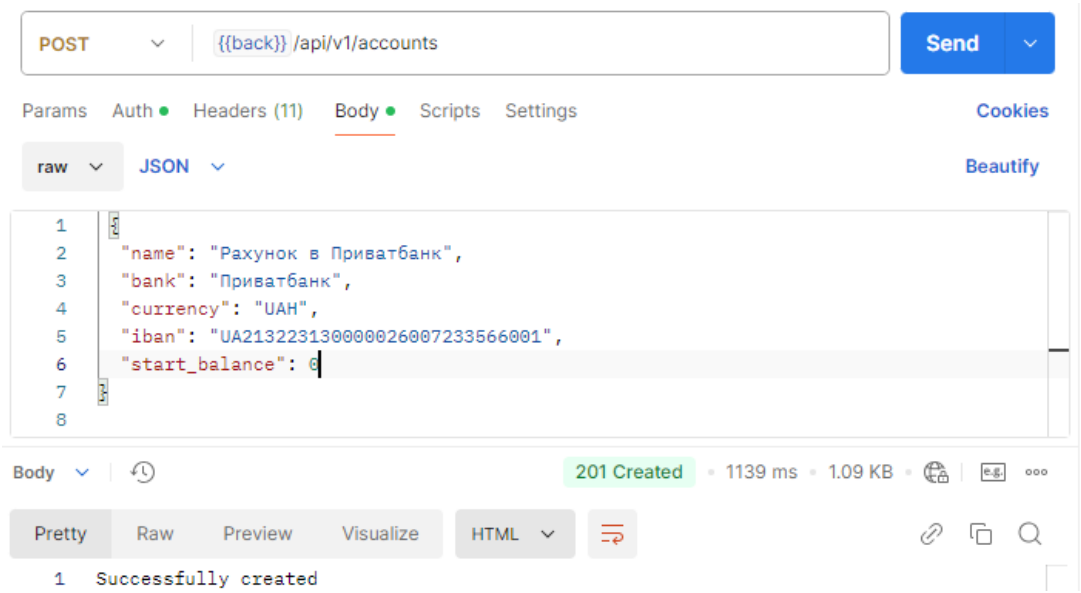


Рисунок 3.23 – Створення рахунку

Для створення транзації використовуємо метод /api/v1/transactions (рис. 3.24).

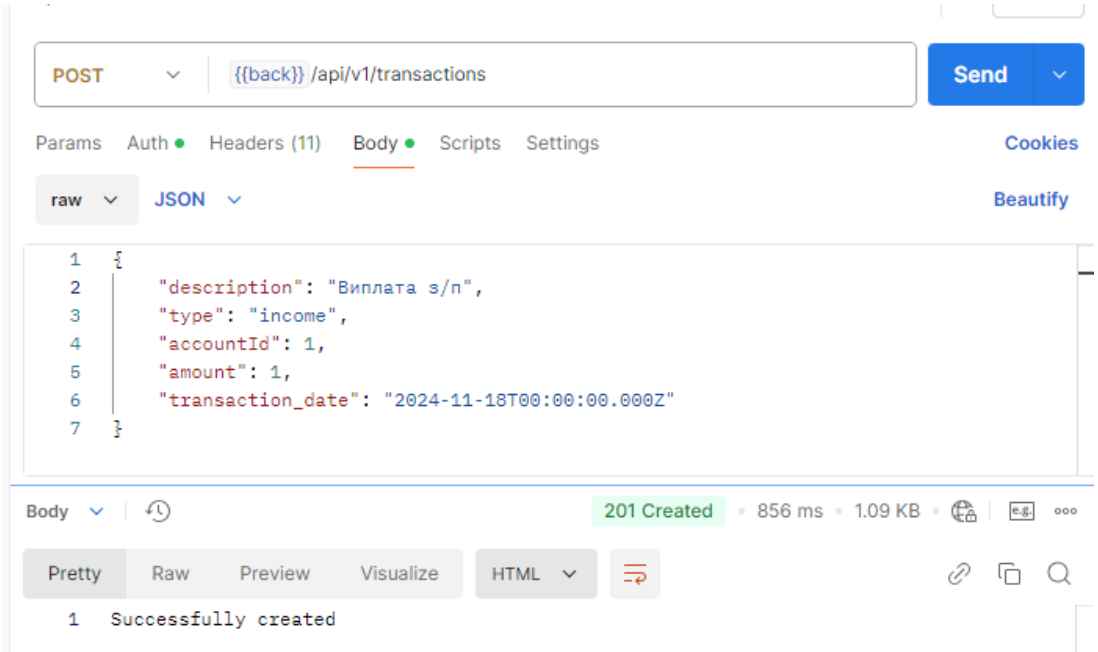


Рисунок 3.24 – Створення транзакції

Для тестування клієнтської сторони необхідно використати інструменти розробника Chrome. Ці інструменти дозволяють перевіряти роботу вебзастосунків, відстежувати запити та відповіді, а також аналізувати використання ресурсів.

При запуску PWA застосунку в інструментах розробника можна спостерігати використання пам’яті в кеші. Це є важливим моментом для оптимізації роботи застосунку, оскільки кешування дозволяє зменшити навантаження на сервер і підвищити швидкість завантаження сторінок. У розділі “Application” можна побачити обсяг пам’яті, що займається кешованими даними, а також переглядати, які саме ресурси зберігаються в кеші.

На рисунку 3.25 продемонстровано, як виглядає інформація про використання пам’яті застосунку в кеші, що дозволяє оцінити ефективність його роботи та зробити висновки щодо потреби в оптимізації кешування.

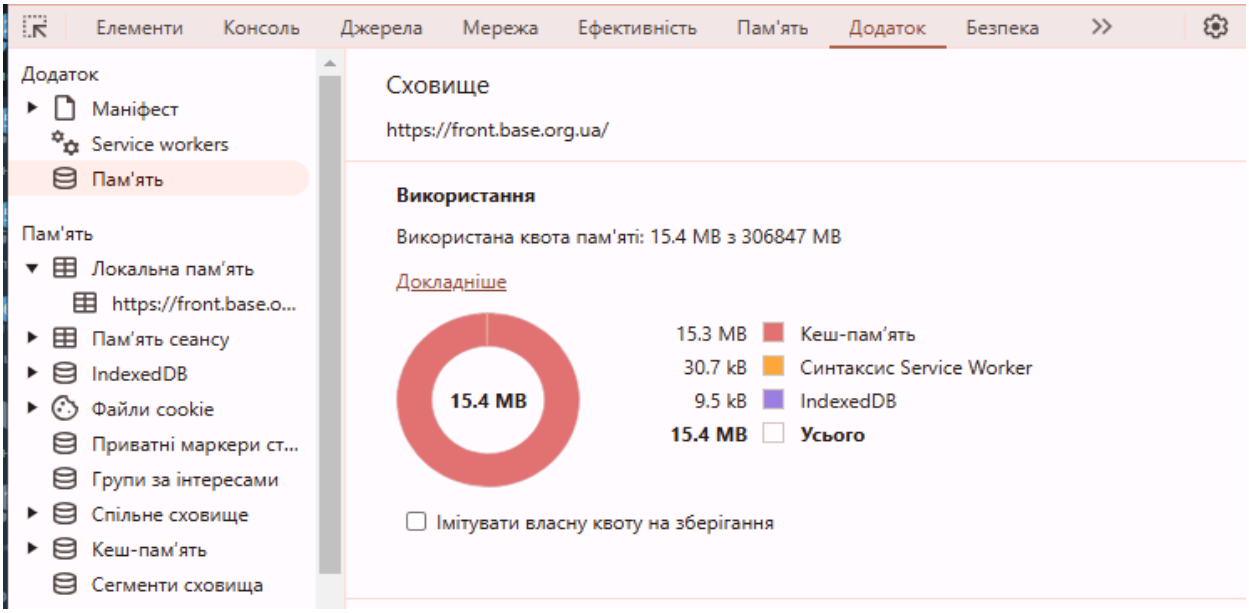


Рисунок 3.25 – Використання пам’яті застосунку

У розділі кеш-пам’ять бачимо список статичних файлів які сервісний воркер закешував при завантаженні сторінки (див. рис. 3.26).

У підрозділі get-api-cache можна побачити всі збережені запити з відповідними даними, що були отримані через метод GET (див. рис. 3.27). Це дозволяє легко відстежувати, які саме запити були кешовані, а також скільки часу займає збереження та доступ до кешованих даних.

Для емуляції роботи офлайн режиму використовуємо режим «Не в мережі» (див. рис. 3.28).

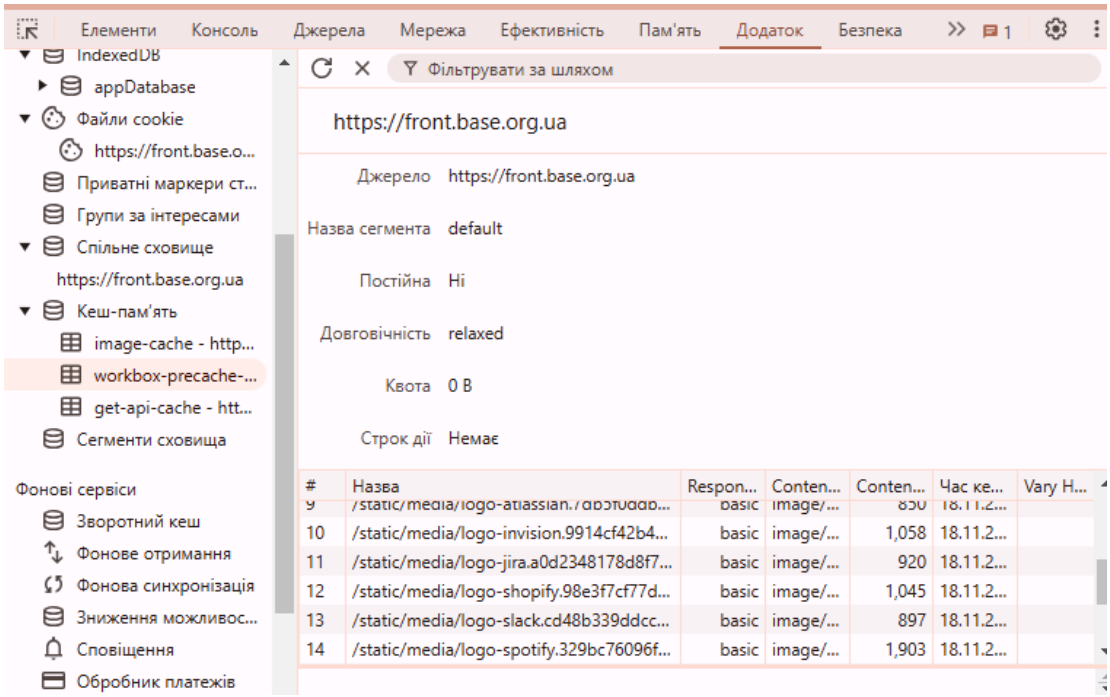


Рисунок 3.26 – Статичні файли в кеші

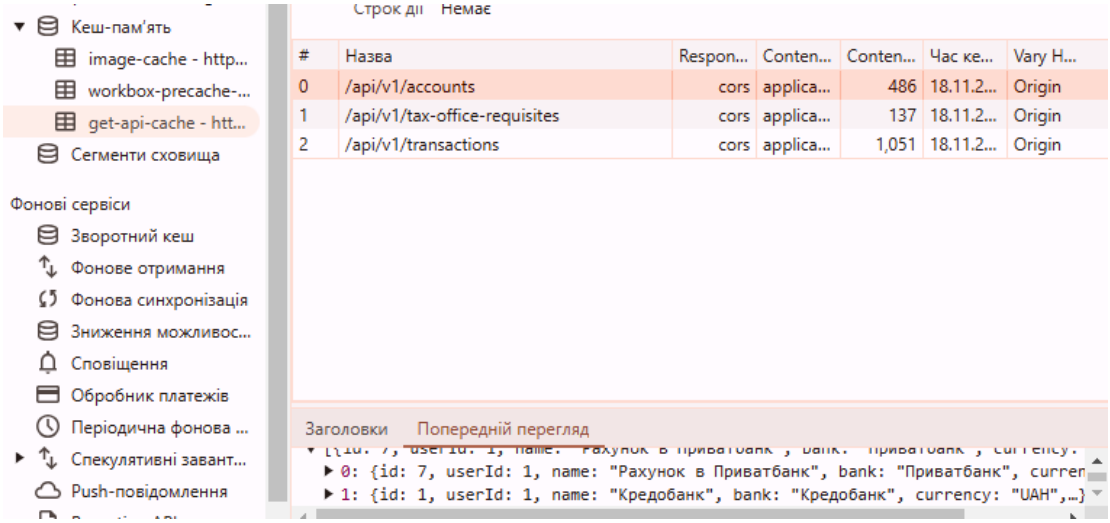


Рисунок 3.27 – Кешування GET-запитів

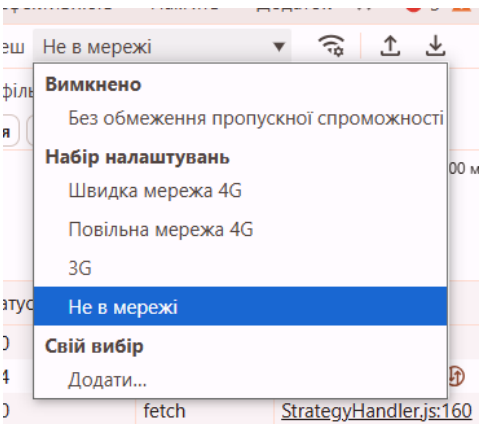


Рисунок 3.28 – Режим «Не в мережі»

В цьому режимі перевіряємо отримання статичних файлів у кеші та GET-запиті в підрозділі get-api-cache (рис. 3.29).

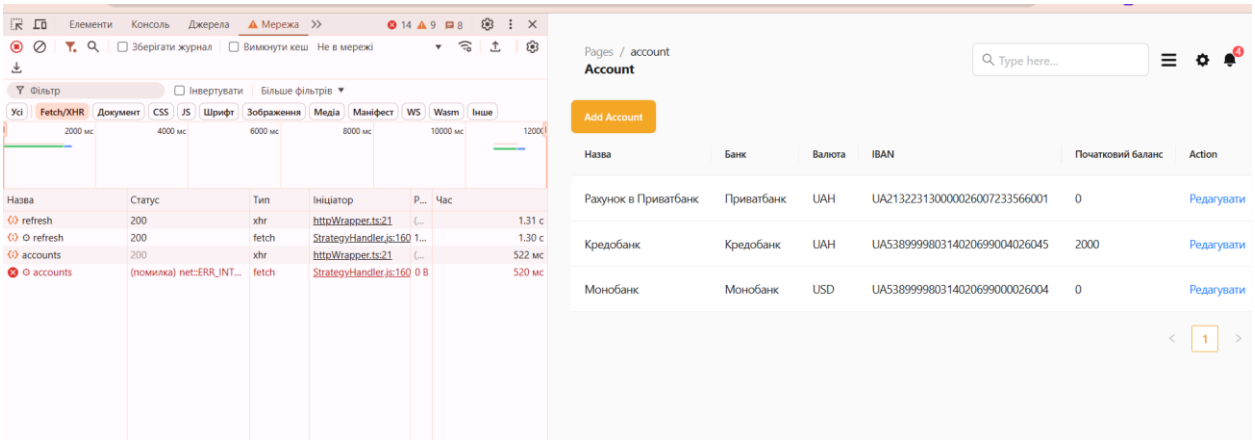


Рисунок 3.29 – Отримання даних з кешу

З результату запиту бачимо, що відсутнє з’єднання, але дані отримано. Це підтверджує, що застосунок готовий працювати у режимі офлайн, забезпечуючи стабільну роботу користувачам навіть за відсутності інтернет-з’єднання.

При створенні POST-запиту на додання рахунку бачимо, що запити на сервер не відправилися (рис. 3.30).

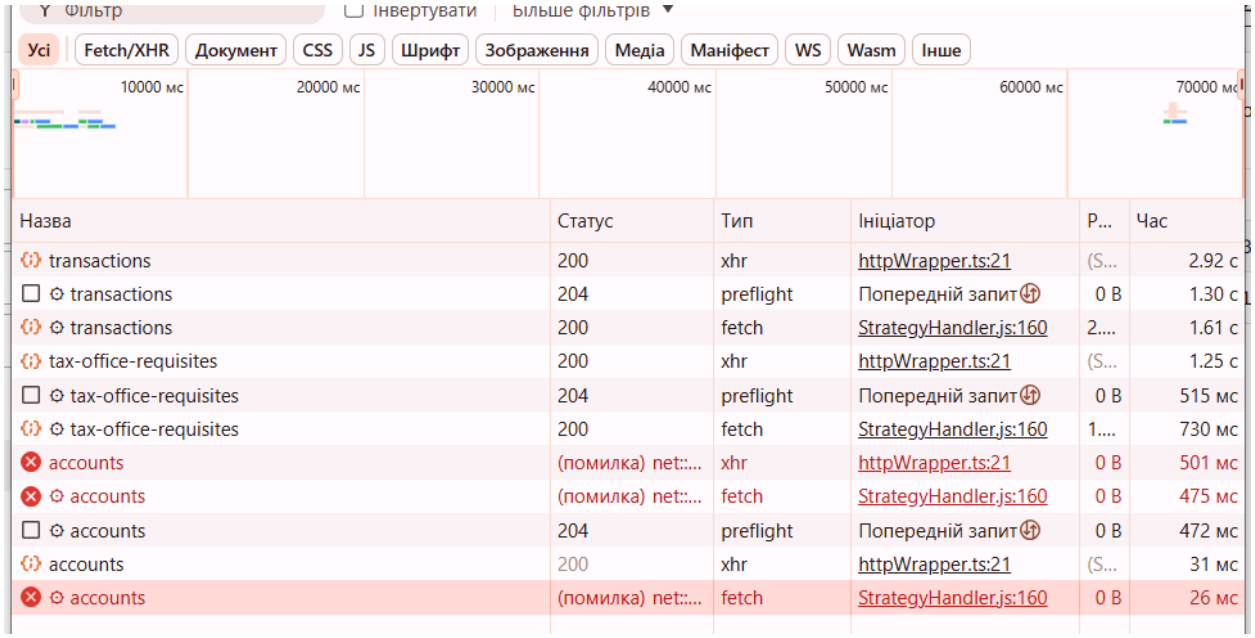


Рисунок 3.30 – Тестування відправки POST-запиту

Після помилкового запиту воркер за допомогою модулю workbox-background-sync додав всі дані цього запиту в чергу в сховище IndexedDB (рис. 3.31).

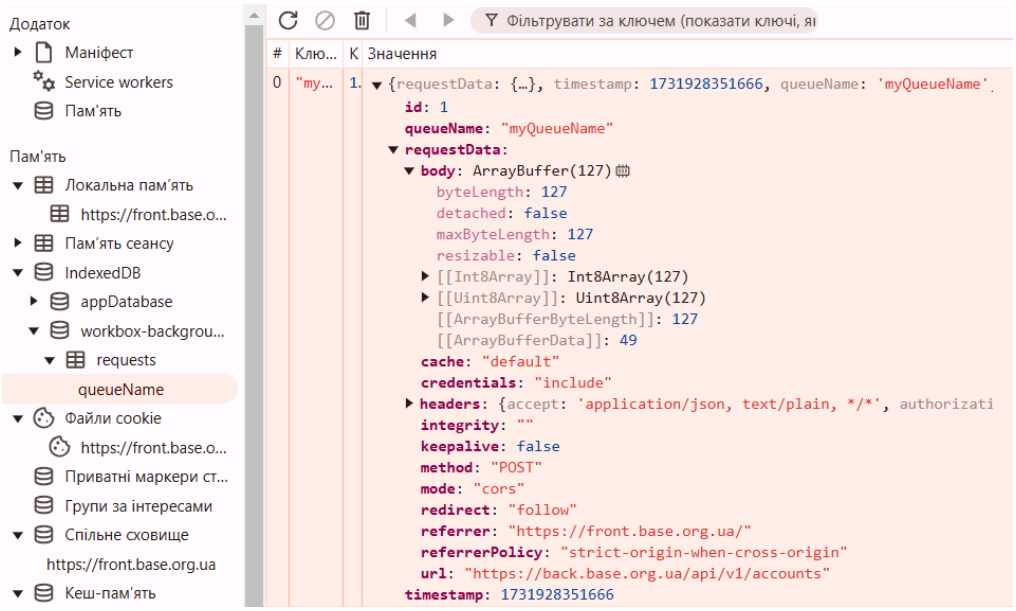


Рисунок 3.31 – Черга запитів в сховищі IndexedDB

Після повернення режиму «Без обмеження пропускної спроможності» воркер надсилає на сервер запит, який був в черзі та отримує позитивну відповідь (рис. 3.32).

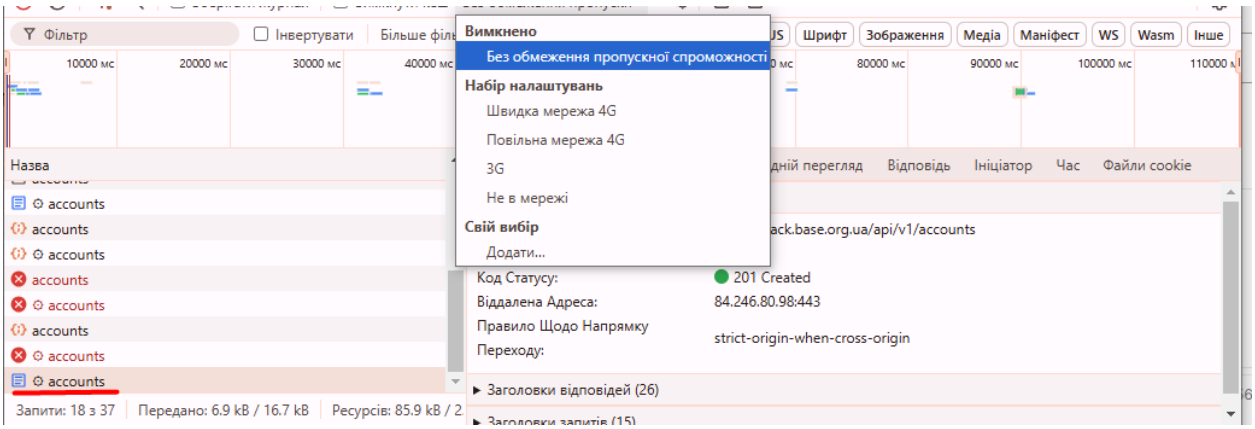


Рисунок 3.32 – Режим онлайн

Для перевірки коректності роботи розробленого PWA-застосунку було проведено тестування основних функціональних модулів. Особливу увагу

приділено перевірці ключових сценаріїв користувацької взаємодії, таких як реєстрація та авторизація користувачів, внесення та редагування податкових даних, формування звітів і управління ролями.

Тестування здійснювалося в різних браузерах і на різних пристроях для забезпечення кросплатформенності застосунку. У процесі тестування було виявлено та усунуто дрібні недоліки у відображенні інтерфейсу та обробці виняткових ситуацій, що дозволило підвищити стабільність і зручність роботи системи. Додатково перевірено коректність роботи застосунку в офлайн-режимі, що є ключовою особливістю PWA.

ВИСНОВКИ

У магістерській роботі було розроблено прогресивний вебзастосунок (PWA) для ведення обліку податків з використанням платформи Node.js. Проведене дослідження та реалізація підтвердили ефективність застосування сучасних вебтехнологій для створення високопродуктивних, зручних і гнучких рішень у сфері обліку та автоматизації фінансових процесів.

У результаті виконаної роботи досягнуто таких основних результатів.

Аналіз проблемної області: проведено дослідження існуючих систем автоматизації обліку податків та визначено основні вимоги до PWA-застосунку, включаючи функціональність, безпеку, масштабованість та доступність.

Розробка архітектури застосунку: спроєктовано архітектуру клієнт-серверної взаємодії, яка забезпечує швидкодію, надійність і адаптивність. Використання Node.js забезпечило асинхронну обробку даних і можливість інтеграції з різними сервісами.

Реалізація функціональності: створено функціонал для обліку податкових даних, автоматизації розрахунків, формування звітів, а також управління користувачами та їх ролями. Завдяки використанню PWA забезпечено можливість роботи застосунку в офлайн-режимі, інтеграції з мобільними пристроями та кросплатформенності.

Тестування: проведено комплексне тестування програмного продукту. Виявлені недоліки було усунуто, що дозволило підвищити стабільність роботи системи.

У процесі розробки було застосовано сучасні підходи до побудови вебзастосунків, що дало змогу створити якісний, функціональний і доступний програмний продукт.

Використання Node.js і NestJS для серверної частини дозволило створити високопродуктивний та масштабований бекенд, який забезпечує швидку обробку податкових даних.

Використання React на фронтенді забезпечило створення сучасного, зручного і реактивного користувацького інтерфейсу.

PostgreSQL був обраний як основна база даних завдяки її надійності, масштабованості та підтримці складних транзакцій.

Функціональність PWA забезпечує можливість роботи офлайн, що є критично важливим для користувачів, які можуть працювати у зонах з нестабільним інтернет-з’єднанням.

Використання кешування з допомогою Service Worker значно покращило продуктивність і швидкість роботи застосунку, забезпечивши збереження основних ресурсів та даних локально.

Робота демонструє практичну реалізацію концепції PWA, підтверджуючи її переваги у вирішенні завдань автоматизації.

Отримані результати можуть стати основою для подальшого розвитку та вдосконалення системи, включаючи інтеграцію з державними податковими платформами, впровадження додаткових аналітичних інструментів та масштабування для роботи з великими обсягами даних.

Розроблений застосунок не тільки дозволить знизити витрати на податковий облік, але й підвищує точність і надійність ведення фінансових даних. Це дозволяє користувачам зосередитися на основній діяльності, а не на адміністративних завданнях, пов’язаних з податковим обліком.

ПЕРЕЛІК ПОСИЛАНЬ

1. Огляд українських бухгалтерських програмних продуктів. URL: https://buhgalter911.com/uk/news/news-1080075.html (дата звернення: 27.10.2024).

2. Progressive web apps. URL: https://developer.mozilla.org/en-US/docs/Web/Progressive_web_apps (дата звернення: 27.10.2024).

3. PWA (Progressive Web App) – що це таке та в чому його особливості. URL: https://highload.today/uk/pwa (дата звернення: 27.10.2024).

4. Порівнюємо React, Angular і Vue. URL: https://dou.ua/forums/topic/39933 (дата звернення: 27.10.2024).

5. Маліцький О.С. Порівняльний аналіз сучасних засобів для розробки систем вебзастосунків. URL: https://inmad.vntu.edu.ua/portal/static/00156A66-D5B6-4671-926B-7AF5BFFA2345.pdf (дата звернення: 27.10.2024).

6. Сайт Node.JS. URL: https://nodejs.org/en (дата звернення: 28.10.2024).

7. Сайт PostgreSQL. URL: https://www.postgresql.org/ (дата звернення: 28.10.2024).

8. REST API як спосіб спілкування компонент вебдодатків. URL: https://foxminded.ua/shcho-take-rest-api (дата звернення: 28.10.2024).

9. Ланевич Т. Принципи SOLID у розробці програмного забезпечення. *Природничі та гуманітарні науки. Актуальні питання*: VII Міжнародна студентська науково-технічна конференція. Тернопільський національний технічний університет імені Івана Пулюя, 2020. С. 89–90.

10. SOLID: The First 5 Principles of Object Oriented Design. URL: https://www.digitalocean.com/community/conceptual-articles/s-o-l-i-d-the-first-five-principles-of-object-oriented-design (дата звернення: 29.10.2024).

11. Strategies for service worker caching. URL: https://developer.chrome.com/docs/workbox/caching-strategies-overview (дата звернення: 29.10.2024).

12. IndexedDB API. URL: https://developer.mozilla.org/en-US/docs/Web/API/IndexedDB_API (дата звернення: 29.10.2024).

13. Introduction to JSON Web Tokens. URL: https://jwt.io/introduction (дата звернення: 30.10.2024).

14. Сайт React. URL: https://react.dev (дата звернення: 30.10.2024).

15. Сайт ReactRouterDom. URL: https://reactrouter.com/en/main (дата звернення: 30.10.2024).

16. Add a web app manifest. URL: https://web.dev/articles/add-manifest (дата звернення: 01.11.2024).

17. Create a service worker with Workbox, Webpack and TypeScript. URL: https://dev.to/chicio/create-a-service-worker-with-workbox-webpack-and-typescript-19k3 (дата звернення: 02.11.2024).

18. Аутентифікація в Node.js за допомогою Passport.js. URL: https://medium.com/devschacht/node-hero-chapter-8-27b74c33a5ce (дата звернення: 02.11.2024).

19. JSON Web Token (JWT) and Passport.js URL: https://designtechworld.medium.com/json-web-token-jwt-and-passport-js-a5361c9a563 (дата звернення: 03.11.2024).

20. Understanding Guards in NestJS. URL: https://www.digitalocean.com/community/tutorials/understanding-guards-in-nestjs (дата звернення: 03.11.2024).

21. OpenAPI Specification. URL: https://swagger.io/resources/open-api/ (дата звернення: 03.11.2024).