

Міністерство освіти і науки України
Харківський національний університет імені В.Н. Каразіна
Навчально-науковий інститут комп'ютерних наук та штучного інтелекту

Рекомендовано до захисту:
протокол засідання кафедри
№ ____ від ____ ____ 2025 р.
Завідувач кафедри ІПСіТ

(підпис) (прізвище та ініціали)

Пояснювальна записка

до кваліфікаційної роботи бакалавра
на тему «Аналіз та порівняння стратегій впровадження мікро-фронтендів із
використанням бібліотеки ReactJS, засобів Webpack Module Federation,
Single-SPA та додаткових плагінів»

Захищено на засіданні ЕК № _____
протокол № ____ від ____ ____ 2025 р.
Оцінка ____ / _____
Голова ЕК

(підпис) (прізвище та ініціали)

Виконав: студент 4 курсу, групи КС-41
Спеціальності:
122 Комп'ютерні науки
Кузьмін Богдан Олексійович

Керівник: ст. викладач КІПСіТ
Паршенцев Б.В.
Рецензент к.ф.-м.н., доцент
Сопоров О.Є.

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ

Харківський національний університет імені В.Н.Каразіна

Навчально-науковий інститут комп'ютерних наук та штучного інтелекту

Кафедра інтелектуальних програмних систем і технологій

Рівень вищої освіти (освітньо-кваліфікаційний рівень) бакалавр

Спеціальність 122 Комп'ютерні науки

ЗАТВЕРДЖУЮ

В.о. завідувача кафедри ІПСіТ

_____ Олег ОЛЕШКО
підпис ім'я, прізвище

“ _____ ” _____ 2025 року

**З А В Д А Н Н Я
НА КВАЛІФІКАЦІЙНУ РОБОТУ**

Кузьміна Богдана Олексійовича

(прізвище, ім'я, по батькові студента)

1. Тема роботи: «Аналіз та порівняння стратегій впровадження мікро-фронтендів із використанням бібліотеки ReactJS, засобів Webpack Module Federation, Single-SPA та додаткових плагінів».

керівник роботи старший викладач КІПСіТ Паршенцев Богдан Володимирович
(прізвище, ім'я, по батькові, науковий ступінь, вчене звання)

затверджені наказом по університету від “16” квітня 2025 року № 4101-5/962

2. Строк подання студентом роботи 02.06.2025 р.

3. Перелік питань, які потрібно розробити:

Дослідити теоретичні засади та принципи архітектури мікрофронтендів, ознайомитись з особливостями використання бібліотеки ReactJS для розробки мікрофронтендів, ознайомитись та провести аналіз Webpack Module Federation як засобу для реалізації мікрофронтендів, розглянути можливостей фреймворку Single-SPA для організації архітектури мікрофронтенд, розробити прототип вебдодатку з використання Module Federation та Single-SPA, провести тестування та порівняння впровадження мікрофронтендів за допомогою Module Federation та Single-SPA.

4. План роботи

№ з/п	Назви етапів роботи
1	Затвердження теми роботи.
2	Аналіз та опрацювання літературних джерел з основ архітектури мікрофронтендів, бібліотеки ReactJS, плагіну Webpack Module Federation та фреймворку Single-SPA.
3	Огляд сучасних підходів, інструментів та додаткових плагінів для реалізації мікрофронтендів.
4	Вибір конкретних інструментів, плагінів для проектування прототипу.
5	Реалізація додатку з використанням Webpack Module Federation.
6	Реалізація застосунку з використанням Single-SPA.
7	Проведення тестування та порівняння впровадження мікрофронтендів за допомогою Module Federation та Single-SPA.
8	Оформлення пояснювальної записки відповідно до вимог.
9	Представлення кваліфікаційної роботи керівнику та рецензенту.

5. Дата видачі завдання 10.12.2024 р.

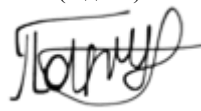
Студент


(підпис)

Кузьмін Б.О.

(ініціали, прізвище)

Керівник роботи


(підпис)

Паршенцев Б.В.

(ініціали, прізвище)

АНОТАЦІЯ

Звіт про виконання кваліфікаційної роботи складається з 53 сторінок, які включають 4 розділи, 36 рисунки, 1 додаток та 17 джерел.

Об'єкт дослідження: процес розробки та впровадження мікрофронтенд архітектури в веб-додатках, що використовують бібліотеку ReactJS.

Предмет дослідження: стратегія реалізації мікрофронтендів з використанням бібліотеки ReactJS, засобів Webpack Module Federation та фреймворку Single-SPA.

Мета роботи: проведення аналізу, порівняння стратегій впровадження мікрофронтендів та розробка прототипу на базі ReactJS з використанням Webpack Module Federation та Single-SPA.

Структура роботи включає дослідження теоретичних засад архітектури мікрофронтендів, аналіз особливостей імплементації мікрофронтендів за допомогою Webpack Module Federation та можливостей фреймворку Single-SPA, розгляд використання ReactJS у контексті мікрофронтендів. Розроблено прототипи вебзастосунку з використанням обох досліджуваних технологій, проведено їх тестування та порівняльний аналіз.

Ключові слова: МІКРОФРОНТЕНД, REACTJS, WEBPACK, MODULE FEDERATION, SINGLE-SPA, АРХІТЕКТУРА, ПРОТОТИП, АНАЛІЗ, ПОРІВНЯННЯ, СТРАТЕГІЯ, ФРОНТЕНД, МОДУЛЬ, ЗАЛЕЖНІСТЬ, ОРКЕСТРАЦІЯ

ANNOTATION

The qualification thesis consists of 52 pages, including 4 sections, 36 figures, 1 appendix, and 17 references.

Object of research: the process of developing and implementing microfrontend architecture in web applications using the ReactJS library.

Subject of research: strategies for implementing microfrontends using the ReactJS library, Webpack Module Federation tools, and the Single-SPA framework.

Aim of the work: to analyze and compare microfrontend implementation strategies and to develop a prototype based on ReactJS using Webpack Module Federation and Single-SPA.

The structure of the work includes the study of theoretical foundations of microfrontend architecture, analysis of the specifics of microfrontend implementation using Webpack Module Federation and the capabilities of the Single-SPA framework, and consideration of ReactJS usage in the context of microfrontends. Prototypes of a web application were developed using both researched technologies, followed by their testing and comparative analysis.

Keywords: MICROFRONTEND, REACTJS, WEBPACK, MODULE FEDERATION, SINGLE-SPA, ARCHITECTURE, PROTOTYPE, ANALYSIS, COMPARISON, STRATEGY, FRONTEND, MODULE, DEPENDENCY, ORCHESTRATION

ЗМІСТ

ПЕРЕЛІК СКОРОЧЕНЬ, УМОВНИХ ПОЗНАЧЕНЬ, ТЕРМІНІВ	5
ВСТУП	6
1 ОСНОВИ ТЕОРІЇ МІКРОФРОНТЕНДІВ	8
1.1 Моноліт та мікрофронтенд архітектура	8
1.2 Існуючі підходи до реалізації мікрофронтендів	11
2 АНАЛІЗ ІНСТРУМЕНТІВ ДЛЯ СТВОРЕННЯ МІКРОФРОНТЕНДІВ..	15
2.1 Аналіз Webpack Module Federation	15
2.2 Аналіз Single-SPA	18
2.3 Використання ReactJS у контексті мікрофронтендів	21
3 ПРОЄКТУВАННЯ ТА ПРОГРАМНА РЕАЛІЗАЦІЯ ПРОТОТИПУ	23
3.1 Структура та функціональні компоненти застосунку	23
3.2 Створення API	24
3.3 Структура монорепозіторію та організація коду	27
3.4 Реалізація маршрутизації та захисту роутів	29
3.5 Інтеграція з Module Federation	32
3.6 Інтеграція з Single-SPA	36
4 ТЕСТУВАННЯ РОЗРОБЛЕНОГО ПРОТОТИПУ ТА ПОРІВНЯННЯ СТРАТЕГІЙ ВПРОВАДЖЕННЯ	41
4.1 Тестування прототипу на базі Module Federation	41
4.2 Тестування прототипу на базі Single-SPA	43
4.3 Тестування працездатності прототипу	44
4.4 Порівняння стратегій впровадження мікрофронтендів	46
4.4.1 Концепція та архітектурний підхід	46

4.4.2	Конфігурація засобів	47
4.4.3	Управління залежностями та їх спільне використання	47
4.4.4	Завантаження під час виконання	48
ВИСНОВКИ.....		50
ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАННЯ		51
ДОДАТОК А.....		53

ПЕРЕЛІК СКОРОЧЕНЬ, УМОВНИХ ПОЗНАЧЕНЬ, ТЕРМІНІВ

SSC – Server-side composition

CSC – Client-side composition

HTML - HyperText markup language

UI - User Interface

URL – Uniform resource locator

DOM – Document object model

SPA – Single page application

API – Application programming interface

HTTP – HyperText transfer protocol

ORM – Object related mapping

JWT – JSON web token

CORS – Cross-origin resource sharing

JSON – JavaScript object notation

JSX - JavaScript XML

ВСТУП

Сучасна веб-розробка характеризується стрімким зростанням складності та масштабованості інтерфейсів користувача. Монолітні фронтенд архітектури часто стають вузьким місцем, ускладнюючи роботу, тестування та розгортання великих проєктів, а також обмежуючи можливість незалежної роботи команд та впровадження нових технологій. Через це найперспективнішою концепцією останніх років стала архітектура мікрофронтендів, яка розширює принципи мікросервісної парадигми на клієнтську частину веб-застосунків.

Мікрофронтенди дозволяють розбивати великий монолітний додаток на менші, незалежно розроблювані та розгортанні частини [1]. Кожна така частина може відповідати за окрему бізнес-функціональність або візуальну область інтерфейсу, що сприяє підвищенню гнучкості, масштабованості та полегшує підтримку складних систем. Використання таких підходів дозволяє командам розробників працювати паралельно, обирати оптимальні технології для конкретних завдань та швидше впроваджувати оновлення та новий функціонал для користувачів.

На сьогодні існує значна кількість інструментів та підходів для реалізації мікрофронтендних архітектур [2]. Серед яких особливої уваги заслуговують плагін Module Federation інструменту Webpack для динамічного завантаження модулів та спільного використання залежностей і фреймворк Single-SPA для оркестрації кілька односторінкових додатків в єдину систему.

Об'єктом дослідження є процес розробки та впровадження мікрофронтенд архітектури в веб-додатках, що використовують бібліотеку ReactJS.

Предметом дослідження є стратегія реалізації мікрофронтендів з використанням бібліотеки ReactJS, засобів Webpack Module Federation та фреймворку Single-SPA.

Метою дослідження є проведення аналізу, порівняння стратегій впровадження мікрофронтендів та розробка прототипу на базі ReactJS з використанням Webpack Module Federation та Single-SPA.

Для досягнення цієї мети були встановлені наступні завдання:

1. Дослідити теоретичні засади та принципи архітектури мікрофронтендів.
2. Проаналізувати особливості імплементації мікрофронтендів за допомогою Webpack Module Federation.
3. Розглянути можливості фреймворку Single-SPA для організації мікрофронтенд архітектури.
4. Розробити прототипи на основі досліджуваних технологій.
5. Провести порівняльний аналіз цих двох підходів.

У рамках цієї роботи буде проведено комплексне дослідження теоретичних засад та практичних аспектів реалізації мікрофронтендної архітектури. Результати аналізу дозволять визначити оптимальні сценарії застосування кожного підходу, а також сформулювати рекомендації щодо вибору технологічного стеку та організації процесу розробки веб-застосунків.

1 ОСНОВИ ТЕОРІЇ МІКРОФРОНТЕНДІВ

1.1 Моноліт та мікрофронтенд архітектура

Монолітна архітектура програмного забезпечення являє собою традиційний підхід до розробки, при якому вся система формується як єдина структура з тісно пов'язаними компонентами. У контексті фронтенд-розробки це означає, що весь користувацький інтерфейс веб-додатку розробляється, компілюється та розгортається як єдина кодова база. Усі функціональні модулі, компоненти та ресурси об'єднані в одному проєкті, що призводить до створення єдиного артефакту збірки [3].

На рисунку 1.1 зображена схема монолітної архітектури фронтенду, де всі компоненти 1-3 збираються в один бандл. Він проходить стадію розгортання та запускається на клієнтській або на серверній стороні.

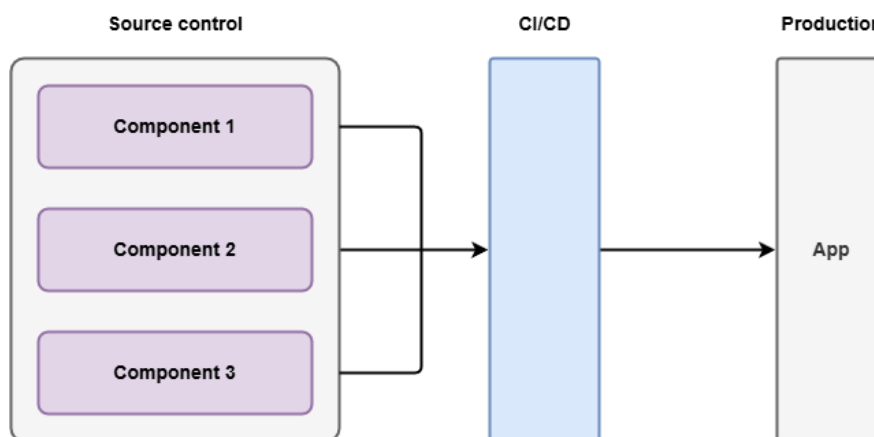


Рисунок 1.1 – Схема монолітної архітектури

Така модель виправдовує себе на початкових етапах життєвого циклу продукту або для невеликих застосунків з малою командою. Завдяки відсутності потреби в координації між різними частинами системи та управлінні складними залежностями, розробка, тестування та розгортання додатку відбувається швидше. Крім того, простіше реалізовувати глобальний стан і спільну логіку, так як всі компоненти мають прямий доступ один до одного в межах єдиної кодової бази.

Водночас, зі збільшенням розміру та складності системи недоліки цієї архітектури стають дедалі помітнішими. По-перше, часто доводиться масштабувати весь додаток, навіть якщо навантаження зростає лише на окремі його функціональні частини. По-друге, великі об'єми коду стає важко підтримувати. Внесення змін в одну частину застосунку може мати непередбачувані наслідки для інших, що ускладнює розробку нового функціоналу та виправлення помилок. По-третє, зазвичай такий додаток прив'язаний до єдиного технологічного стеку, оновлення технологій або впровадження нових стає дорогим і часозатратним процесом. З розвитком додатку зростає час, необхідний для компіляції, тестування та розгортання, що уповільнює цикли розробки в великих проєктах. Навіть незначні оновлення вимагають повного циклу збірки продукту. Також слід зазначити, що між командами існує залежності впровадження нових можливостей, які призводять до конфліктів злиття, що в свою чергу збільшує загальний час розробки.

Як наслідок, технічні обмеження моноліту безпосередньо трансформуються в організаційні проблеми. Саме тому перехід до мікрофронтендів часто є розв'язком не лише для технічних викликів масштабування коду але й для організації потреб у підвищенні гнучкості та паралелізації робочих процесів [1].

Архітектура мікрофронтендів являє собою сучасний підхід до розробки клієнтської частини веб-додатків. Суть цього підходу полягає в декомпозиції об'ємного застосунку на менші, відокремлені, слабо зв'язані частини. Кожен з таких фрагментів може бути відповідальним за певну бізнес-функціональність, окрему сторінку, її секцію чи навіть окремий елемент інтерфейсу користувача. Ці автономні компоненти згодом інтегруються: на сервері чи безпосередньо на стороні клієнта, для формування єдиного цілісного користувацького досвіду [4].

На рисунку 1.2 зображена схема мікрофронтендної архітектури, де всі модулі MF1, MF2, MF3 розробляються та розгортаються незалежно і потім об'єднуються в єдиний застосунок.

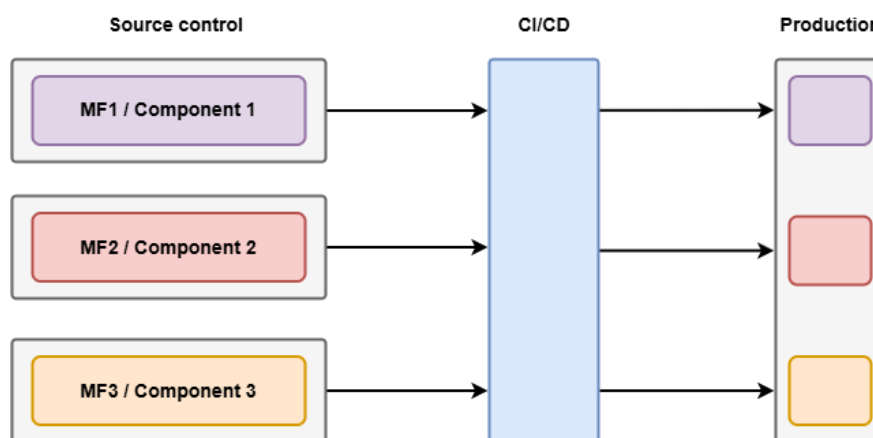


Рисунок 1.2 – Схема мікрофронтендної архітектури

Ефективність такої архітектури залежить від дотримання низки основоположних принципів, які забезпечують її функціональність та переваги [5]. Насамперед, кожен мікрофронтенд має бути спроектований таким чином, щоб його можна було розгортати незалежно від інших частин системи, уникаючи необхідності перезбирати усю програму при оновленні одного компонента, також це дозволяє унеможливити виникнення ризиків при злитті коду. Важливим аспектом є автономність команд розробників, які беруть на себе повну відповідальність за розробку своїх частин додатку – від етапу проектування до підтримки в продакшені. Гнучкість у виборі технологій також відіграє ключову роль: команди можуть використовувати різні технологічні стеки залежно від потреб проєкту. При цьому кожен мікрофронтенд має чітко окреслені межі, відповідаючи за конкретну частину логіки чи інтерфейсу, що сприяє ізоляції коду, що спрощує його організацію та підтримку.

Незважаючи на численні переваги, впровадження мікрофронтендної архітектури супроводжується певними труднощами. Управління множиною мікрофронтендів, їхньою взаємодією та загальною інфраструктурою вимагає ретельної координації та може бути значно складнішим, ніж у монолітному

підході [6]. Ефективна комунікація між командами стає критично важливою для уникнення неузгодженого користувацького досвіду та проблем інтеграції. Управління спільними залежностями між мікрофронтендами може ускладнювати розробку, призводячи до ризику конфліктів версій або надлишкового дублювання коду. Також нетривіальним завданням є забезпечення єдиного та послідовного візуального стилю, поведінки та загального користувацького досвіду, що в свою чергу вимагає впровадження спільних стилів, окремих компонентів тощо. Ще однією перешкодою до впровадження мікрофронтендів є тестування, а саме процес тестування між окремими компонентами.

Технологічна гнучкість, яку надають мікрофронтенди, є дуже привабливою, але водночас і потенційним джерелом проблем. Дозвіл командам обирати різні технології може стимулювати інновації та дозволити використовувати найкращі інструменти для конкретних завдань. Однак, без належного управління та чітких стандартів така свобода може призвести до хаосу в архітектурі, складнощів у підтримці та зниженні загальної якості програмного продукту [1].

1.2 Існуючі підходи до реалізації мікрофронтендів

Реалізація мікрофронтендної архітектури передбачає вибір стратегії композиції – способу, завдяки якому незалежно розроблені компоненти системи об'єднуються для формування єдиного додатку. Загалом існує два підходи: композиція на стороні сервера (SSC) та на стороні клієнта (CSC) [7].

Композиція на стороні сервера – це метод, при якому збирання сторінки з окремих фрагментів, що надаються різними мікрофронтендами відбувається на сервері перед тим як готова HTML сторінка буде відправлена клієнтському браузеру [8]. Коли користувач робить запит на певну сторінку, спеціальний сервіс звертається до відповідних мікрофронтендних додатків, отримує від них фрагменти, компонує їх і відправляє результат браузеру (рис. 1.3).

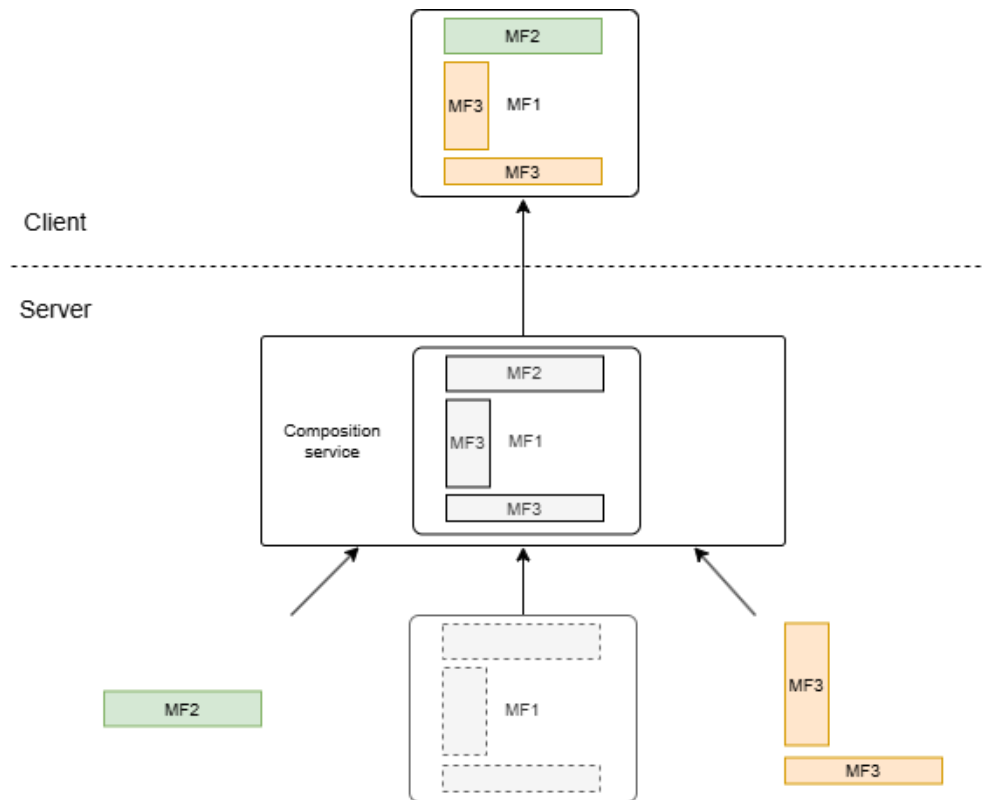


Рисунок 1.3 – Схема SSC підходу

Завдяки SSC користувач швидше бачить значущий контент на сторінці, оскільки браузеру не потрібно чекати завантаження та виконання великих JavaScript-бандлів, також це може добре вплинути на продуктивність. З недоліків можна виділити збільшене навантаження на сервер.

Інший підхід це композиція на стороні клієнта, при якому збирання окремих мікрофронтендів в єдиний користувацький інтерфейс відбувається безпосередньо в браузері. Зазвичай, для цього використовується контейнерний додаток, котрий відповідає за завантаження, ініціалізацію та оркестрацію різних мікрофронтендів, а також за управління їхніми життєвими циклами та взаємодією на сторінці [9]. Кожен мікрофронтенд завантажується як окремий JavaScript-бандл і рендерить свою частину UI у визначене для нього місце (рис. 1.4).

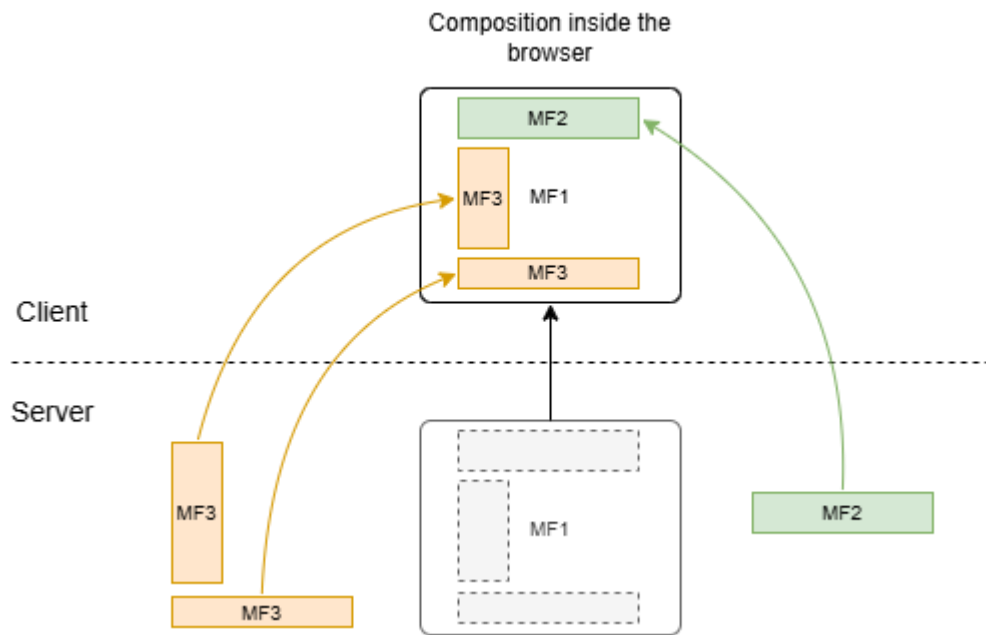


Рисунок 1.4 – Схема CSC підходу

Однією з ключових переваг CSC є можливість динамічного завантаження потрібних фрагментів, що в свою чергу підвищує продуктивність та може значно покращити час початкового завантаження всього додатку. До недоліків можна віднести складність управління залежностями, що призводить до збільшення загального розміру завантажуваного коду, складність в координації, інтеграції, та в комунікації між різними додатками.

Існує декілька способів побудови композиції на стороні клієнта:

Інтеграція під час збірки

При такому підході мікрофронтенди комбінуються в єдиний бандл або набір бандлів на етапі збірки всього додатку [10]. Окремі мікрофронтенди часто розробляються як незалежні проекти і потім публікуються як npm-пакети. Головний додаток встановлює ці пакети як свої залежності та інтегрує їх у свій процес збірки. У кінцевому результаті, додаток, що розгортається вже містить в собі віддалені фрагменти.

Інтеграція під час виконання

При інтеграції під час виконання мікрофронтенди завантажуються та інтегруються в головний додаток динамічно або безпосередньо під час роботи

застосунку. Контейнерний додаток приймає рішення про те, який віддалений фрагмент завантажити, коли його ініціалізувати та де відобразити на сторінці [10]. Цю інтеграцію можна реалізувати за допомогою iFrames, JavaScript, Web Components, Module Federation або Single-SPA.

Отже вибір між цими підходами залежить від специфіки проєкту, зокрема від вимог до швидкості, складності інтеграції та ресурсів команди розробників.

2 АНАЛІЗ ІНСТРУМЕНТІВ ДЛЯ СТВОРЕННЯ МІКРОФРОНТЕНДІВ

2.1 Аналіз Webpack Module Federation

Webpack - це інструмент, який збирає JavaScript модулі до єдиного бандлу [11]. Тобто у конфігурації цього інструменту вказуємо вхідний файл, котрий буде аналізуватися на наявність імпортованих модулів. Завдяки цьому до HTML сторінки буде під'єднаний один файл замість багатьох окремих, що дозволяє зменшити кількість запитів до серверу. Загалом Webpack велика екосистема, яка містить дуже багато плагінів, котрі розширюють його стандартні можливості.

Module Federation є плагіном Webpack, який докорінно змінює підхід до побудови складних клієнтських застосунків, зокрема тих, що використовують мікрофронтенду архітектуру. Він дозволяє динамічно завантажувати код з інших незалежно розгорнутих додатків у двосторонньому порядку безпосередньо під час виконання [12]. Основна мета Module Federation полягає у забезпеченні ефективного спільного використання модулів та залежностей між різними частинами програми, що є фундаментальним для реалізації мікрофронтендів.

Основні терміни Module Federation:

Головний додаток (Host)

Виступає споживачем модулів, наданих іншими додатками. Зазвичай ініціалізується першим при завантаженні сторінки й відповідає за загальну структуру та завантаження віддалених модулів.

Додаток постачальник (Remote)

Експортує свої частини коду для споживання їх головними додатками.

Експортовані модулі (Exposed Modules)

Конкретний модуль (компонент, функція, об'єкт) всередині віддаленого додатку, які він робить доступними для зовнішнього споживання. Конфігуруються за допомогою опції `exposes` (приймає об'єкт експортованих модулів) у файлі конфігурації Webpack.

Імпортовані модулі (Remotes Modules)

Модуль який імпортує додаток приймач для своїх потреб. Конфігуруються за допомогою опції `remotes` (приймає об'єкт імпортованих модулів) у файлі конфігурації Webpack.

Спільні залежності (Shared Dependencies)

Бібліотеки, які можуть бути спільно використані між головним додатком та віддаленими. Цей механізм дозволяє налаштувати які бібліотеки повинні завантажуватись одразу, наприклад `ReactJS`, `ReactDOM`. Також надає можливість вказувати конкретні версії чи діапазон версій та чи будуть вони завантажені в одному екземплярі. Завдяки таким гнучким налаштуванням можна покращити продуктивність завантаження застосунку та мінімізувати ризик конфліктів між різними версіями однієї бібліотеки [12].

Розглянемо приклад конфігурації Webpack для Host (рис. 2.2) та Remote (рис. 2.1) додатків з плагіном Module Federation.

```
const { ModuleFederationPlugin } = require('webpack').container;

module.exports = {
  plugins: [
    new ModuleFederationPlugin({
      // Унікальне ім'я віддаленого додатка
      name: 'remoteModule',
      // Назва файлу, що буде згенерований для доступу до модулів
      filename: 'remoteEntry.js',
      // Об'єкт exposes визначає модулі, що віддаються
      exposes: {
        // Маска та шлях до модуля, який віддаємо
        './Header': './src/components/header/Header',
      },
      // Об'єкт shared визначає спільні бібліотеки
      shared: {
        react: { eager: true, singleton: true, requiredVersion: '^18.2.0' },
        'react-dom': { eager: true, singleton: true, requiredVersion: '^18.2.0' },
        'react-router-dom': { eager: true, singleton: true, requiredVersion: '^7.4.0' }
      }
    })
  ],
  devServer: { port: 3001 }
};
```

Рисунок 2.1 – Конфігурація Webpack Module Federation для Remote додатку

У цьому прикладі додаток постачальник віддає модуль Header.js, який він експортує під маскою ./Header у своєму remoteEntry.js файлі. Під час запуску цей файл реєструє в глобальному скоупі Module Federation доступний віддалено модуль, який потім зможе використовувати головний додаток. Також в об'єкті shared ключами є назви бібліотек, які ми хочемо спільно використовувати а в об'єкті значень цих ключів зазначені додаткові параметри:

- singleton - забезпечує єдину версію бібліотеки;
- eager - завантажує бібліотеку одразу під час ініціалізації додатка;
- requiredVersion - вказує діапазон сумісних версій бібліотеки.

Перейдемо до конфігурації головного додатку

```
const { ModuleFederationPlugin } = require('webpack').container;

module.exports = {
  plugins: [
    new ModuleFederationPlugin(
      {
        // Унікальне ім'я головного додатка
        name: 'container',
        // Назва файлу, що буде згенерований для доступу до модулів
        filename: 'remoteEntry.js',
        // Об'єкт remotes визначає модулі, що імпортуються
        remotes: {
          // URL віддаленого модуля
          remoteModule: 'remoteModule@http://localhost:3001/remoteEntry.js',
        },
        // Об'єкт shared визначає спільні бібліотеки
        shared: {
          react: { eager: true, singleton: true, requiredVersion: '^18.2.0' },
          'react-dom': { eager: true, singleton: true, requiredVersion: '^18.2.0' },
          'react-router-dom': { eager: true, singleton: true, requiredVersion: '^7.4.0' }
        }
      }
    ),
  ],
  devServer: { port: 3000 }
};
```

Рисунок 2.2 – Конфігурація Webpack Module Federation для Host додатку

У прикладі для Host додатку в об'єкті remotes визначаємо модулі, що імпортуються, вказуючи ключем локальне ім'я для імпортованого модуля, а значенням цього ключа є URL де він знаходиться. Після чого можемо

використовувати його для написання функціоналу імпортувавши у потрібний компонент (рис. 2.3).

```
import { lazy, Suspense } from 'react';
const Header = lazy(() => import('remoteModule/Header'));

function App() {
  return (
    <Suspense fallback={<div>Loading...</div>}>
      <Header />
    </Suspense>
  );
}

export default App;
```

Рисунок 2.3 – Компонент App, який імпортує віддалений модуль

Отже, плагін Module Federation є потужним інструментом для створення масштабованих вебдодатків. Завдяки механізмам `remotes` та `exposes` дозволяє динамічно об'єднувати незалежно розроблений функціонал, а налаштування через `shared` забезпечує гнучке керування залежностями.

2.2 Аналіз Single-SPA

Single-SPA - це фреймворк основним призначенням якого є забезпечення інтеграції та оркестрації декількох незалежних мікрофронтенд застосунків, в єдиний додаток [13]. Це досягається шляхом управління життєвими циклами окремих незалежних програм та їх монтуванням в зазначене місце на основі URL-маршрутизації.

Single-SPA базується на ідеї застосування концепції життєвого циклу компонента, яка використовується в багатьох JavaScript фреймворках. Це означає, що у кожного мікрофронтенду є три життєві стадії: `bootstrap` (ініціалізація додатка), `mount` (монтування в DOM, відображення) та `unmount` (демонтування з DOM) [13]. Коли користувач переходить за певним URL, Single-SPA перевіряє його, щоб він відповідав умовам у файлі конфігурації і

якщо умова дійсна, то відповідний віддалений додаток завантажується, ініціалізується, відображається, і при зміні маршруту демонтується.

Основні терміни Single-SPA:

Додатки (Applications)

Повноцінні односторінкові застосунки (SPA) або їх значні частини, які адаптовані для роботи в середовищі Single-SPA. Вони є автономними частинами вебсайту, які реєструються в Single-SPA і активуються за певних умов. Кожен такий додаток повинен експортувати функції життєвого циклу, які будуть контрольовано викликатись фреймворком оркестрації.

Посилки (Parcels)

Менші, незалежні частини коду (UI компоненти, функції тощо), які не є повноцінними застосунками, але можуть бути вбудовані в них. На відміну від applications, parcels не реєструються в Single-SPA (не керуються безпосередньо URL-адресою), а викликаються імперативно з коду. Вони також повинні експортувати функції життєвого циклу.

Коренева конфігурація (Root Config)

Елемент в якому відбувається збірка усіх віддалених додатків в один повноцінний. Коренева конфігурація складається з HTML файлу (слугує точкою входу для всього додатку) та JavaScript файлу (відповідає за реєстрацію всіх мікрофронтендів у системі Single-SPA та за запуск фреймворку) [13].

Для спрощення інтеграції мікрофронтендів, розроблених на ReactJS існує допоміжна бібліотека single-spa-react. Вона надає зручний API для реалізації необхідних функцій життєвого циклу Single-SPA для React додатків [14].

Розглянемо приклад кореневої конфігурації (рис. 2.6-2.7) та вхідного файлу мікрофронтенду (рис. 2.5).

```

import React from 'react';
import ReactDOM from 'react-dom/client';
import singleSpaReact from 'single-spa-react';
import App from './components/app/App';

// Створення об'єкта життєвого циклу для Single-SPA за допомогою single-spa-react
const lifecycles = singleSpaReact({
  React,
  ReactDOM,
  // Кореневий компонент, який буде відображатися
  rootComponent: App,
  renderType: 'createRoot',
  // Визначення обробника помилок
  errorBoundary(err) {
    return <div>Error in App: {err.message}</div>;
  },
  domElementGetter: () => document.getElementById('app'),
});

// Експорт функцій життєвого циклу
export const { bootstrap, mount, unmount } = lifecycles;

```

Рисунок 2.5 – Конфігурація вхідного файлу мікрофронтенду

Функція `singleSpaReact` повертає об'єкт із трьома методами (`bootstrap`, `mount`, `unmount`), які Single-SPA використовує для керування життєвим циклом експортованого додатку.

```

import { registerApplication, start } from 'single-spa';

// Функція для реєстрації мікрофронтенду в кореновому застосунку
registerApplication({
  name: 'app',
  app: () => System.import('app'),
  // Маршрут активації мікрофронтенду
  activeWhen: '/',
  customProps: { domElement: document.getElementById('app') },
});

// Функція для запуску системи маршрутизації та життєвих циклів
start();

```

Рисунок 2.6 – Конфігураційний файл реєстрації віддалених додатків

```

<!DOCTYPE html>
<html lang="en">

<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <title>Example</title>
  <script type="systemjs-importmap">
    {
      "imports": {
        "app": "http://localhost:3001/js/app.js",
      }
    }
  </script>
  <script src="https://cdn.jsdelivr.net/npm/systemjs@6.11.0/dist/system.min.js"></script>
</head>

<body>
  <div id="app"></div>
</body>

</html>

```

Рисунок 2.7 - Import Map у головному HTML файлі

Скрипт Import Map дає браузеру інформацію про те за яким URL потрібно шукати імпорт "app". Таким чином при переході на шлях "/" застосунку в елементі div з id="app" відобразиться віддалений контент.

Використання кореневої конфігурації дозволяє змінювати склад і версії віддалених модулів без перекомпіляції основного застосунку, а допоміжні бібліотеки, як single-spa-react роблять оркестрацію мікрофронтендів більш тривіальною.

2.3 Використання ReactJS у контексті мікрофронтендів

ReactJS одна з найпопулярніших бібліотек для побудови користувацьких інтерфейсів [15]. ReactJS має дуже велику екосистему, яка надає розробникам широкий вибір інструментів, бібліотек та готових рішень для вирішення типових завдань, що виникають при розробці мікрофронтендів.

Фундаментальною основою ReactJS є компонентна архітектура, яка надає можливість розбивати складний UI на невеликі, незалежні та перевикористовувані компоненти, що ідеально відповідає ідеології

мікрофронтендів, де кожен додаток може розглядатися як окремий, самодостатній функціональний блок або набір таких блоків, що інкапсують свою логіку [15].

Наявність вбудованого React Context API дозволяє реалізовувати різноманітні та гнучкі стратегії управління станом як всередині окремих мікрофронтендів, так і для організації спільного стану між ними [16].

Додаткова бібліотека React Router дозволяє забезпечити як глобальну навігацію між мікрофронтендами (коли головний додаток несе відповідальність за визначення того, який саме мікрофронтенд має бути активним на основі поточного URL шляху в адресному рядку браузера), так і внутрішню навігацію всередині кожного з них [17].

Однією з істотних проблем при інтеграції декількох незалежно розроблених мікрофронтендів на одній сторінці є потенційні конфлікти CSS-стилів. Стили визначені в одному мікрофронтенді можуть випадково впливати на вигляд інших мікрофронтендів, якщо не застосовані належні механізми ізоляції. Для того щоб це забезпечити потрібно використовувати CSS модулі, бібліотеки для написання стилів безпосередньо в JavaScript коді (генерують унікальні імена класів), розробити та використовувати спільну бібліотеку UI компонентів.

Функціональність у ReactJS як ліниве завантаження допомагає з оптимізацією. Тобто ми можемо не одразу завантажувати віддалені модулі до основного додатку, а за потреби, завдяки цьому зменшується початковий розмір бандлу.

Також ReactJS надає можливість досить легко реалізовувати мемоізацію (збереження даних функцією поки не зміняться вхідні дані) лише за допомогою хуків `useMemo` та `useCallback`.

ReactJS ідеально підходить для мікрофронтендів завдяки своїй модульності, гнучкості та дуже великій екосистемі. Потрібно пам'ятати, що успішна реалізація залежить від правильного управління станом, маршрутизацією, комунікацією та стилізацією.

3 ПРОЄКТУВАННЯ ТА ПРОГРАМНА РЕАЛІЗАЦІЯ ПРОТОТИПУ

3.1 Структура та функціональні компоненти застосунку

У рамках дослідження було спроектовано та реалізовано прототип вебзастосунку, основною метою якого є спрощення процесу створення та управління заявами. Платформа орієнтована на зручність користувачів і надає інтуїтивно зрозумілий інтерфейс для взаємодії з системою. Користувачі мають можливість проходити процедуру реєстрації, автентифікації, створювати нові заяви, а також переглядати, фільтрувати та відстежувати їхній статус.

З метою забезпечення масштабованості, підтримуваності та модульності було обрано мікрофронтендний підхід, він дозволяє розбивати великий застосунок на незалежні функціональні частини, кожна з яких відповідає за окрему бізнес-логіку. У результаті додаток було розбито на такі основні компоненти:

Головний SPA

- Відображення загальної навігації.
- Забезпечення механізмів автентифікації та авторизації користувачів.
- Динамічне завантаження та інтеграція окремих мікрофронтендів.
- Управління маршрутизацією.

Мікрофронтенд з перегляду заяв

- Завантаження та візуалізація списку заяв користувача.
- Забезпечення фільтрації, сортування та пошуку заяв.
- Оптимізація завантаження та відображення великої кількості даних через пагінацію.

Мікрофронтенд зі створення заяв

- Підтримка декількох типів заяв з різною структурою форм.
- Використання динамічних форм, які змінюються залежно від вибраного типу заяви.
- Використання спільних компонентів.

Застосування такої архітектури дозволило забезпечити високий рівень ізоляції функціональностей, що є ключовою вимогою у мікрофронтендному підході. Кожен модуль може розроблятися, тестуватися і оновлюватися незалежно, що значно спрощує підтримку та розвиток системи у майбутньому.

3.2 Створення API

Для забезпечення взаємодії клієнтської частини прототипу з серверною логікою та базою даних було реалізовано простий серверний API. Воно відповідає за обробку HTTP запитів від клієнтських мікрофронтендів, управління даними користувачів (реєстрація, оновлення профілю), процеси автентифікації та авторизації, а також за виконання всіх операцій, пов'язаних із заявами.

Основним інструментом для створення серверної частини було обрано FastAPI - сучасний фреймворк для Python. Завдяки використанню Pydantic для валідації та серіалізації даних, API забезпечує суворе дотримання форматів вхідних і вихідних структур, що, у свою чергу, зменшує ймовірність помилок та підвищує стабільність системи.

Основою взаємодії з базою даних слугує ORM бібліотека SQLAlchemy. Вона дозволяє розробникам працювати з таблицями реляційної бази даних у вигляді класів та об'єктів, абстрагуючись від написання запитів вручну, що прискорює розробку та зменшує кількість потенційних помилок. У якості системи управління базою даних використовується PostgreSQL, що гарантує надійність, масштабованість та сумісність із сучасними стандартами реляційних систем. Для представлення таблиць users та requests у коді були розроблені відповідні моделі SQLAlchemy, які описують структуру даних та логіку їх збереження в базі.

Ключовим елементом безпеки системи є автентифікація на основі JWT (JSON Web Token), яка поєднує надійність і гнучкість. Після успішного входу користувача сервер генерує токен, що містить у собі унікальний ідентифікатор користувача, час створення та час закінчення дії. Цей токен зберігається у

HTTP-only cookie, що унеможливило б доступ до нього на клієнті через JavaScript.

При кожному наступному запиті до захищених ресурсів API клієнтський браузер автоматично надсилає цей cookie разом із запитом. Сервер, у свою чергу, отримує токен, розшифровує його, перевіряє підпис та валідність. У разі успішної валідації токена запит обробляється, в іншому випадку (невалідний, підроблений або прострочений токен) користувач отримує відповідь 401 Unauthorized, що сигналізує про необхідність повторної автентифікації.

API умовно розділене на кілька сервісів, кожен з яких відповідає за певну функціональність.

Сервіс автентифікації (рис. 3.1) відповідає за управління реєстрацією, входом та виходом користувачів.

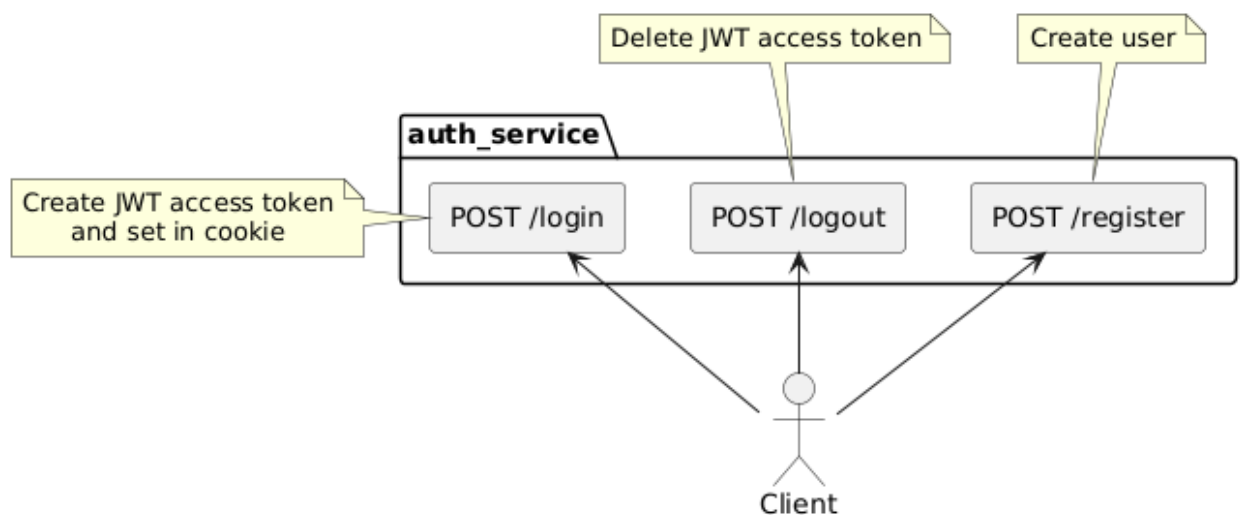


Рисунок 3.1 – Схема ендпоінтів сервісу автентифікації

Процес створення нових користувачів включає перевірку унікальності вказаної електронної пошти в базі даних. Вхід у систему відбувається шляхом перевірки наданих облікових даних (email та пароль) з хешованим паролем у базі та, у разі успіху, генерацією та відправкою JWT. Процедура виходу зазвичай передбачає видалення HTTP-only cookie з токеном на стороні клієнта.

Сервіс користувача (рис. 3.2) надає можливість отримувати та редагувати персональні дані.

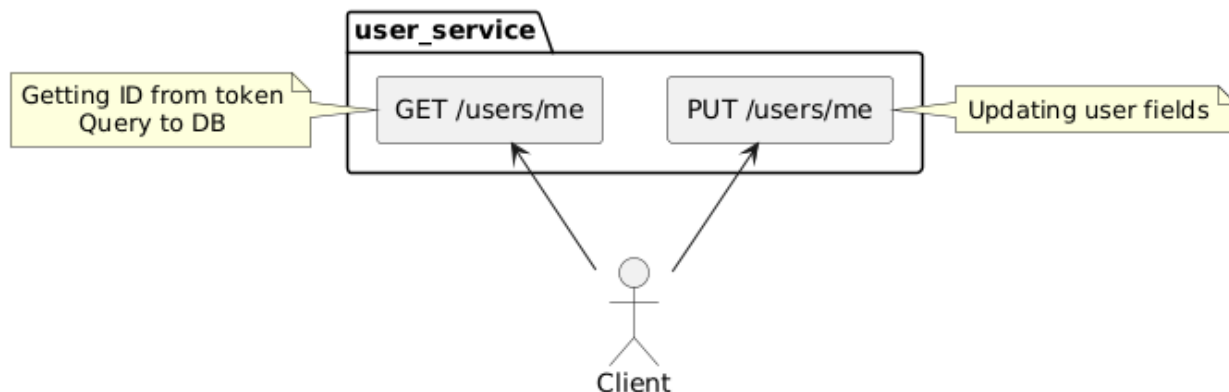


Рисунок 3.2 – Схема ендпоінтів сервісу користувача

Уся інформація, зокрема email, ім'я та прізвище, витягується за допомогою JWT і оновлюється лише у межах сесії відповідного користувача.

Сервіс заяв (рис. 3.3) відповідає за створення й витяг усіх заяв користувача з бази даних.

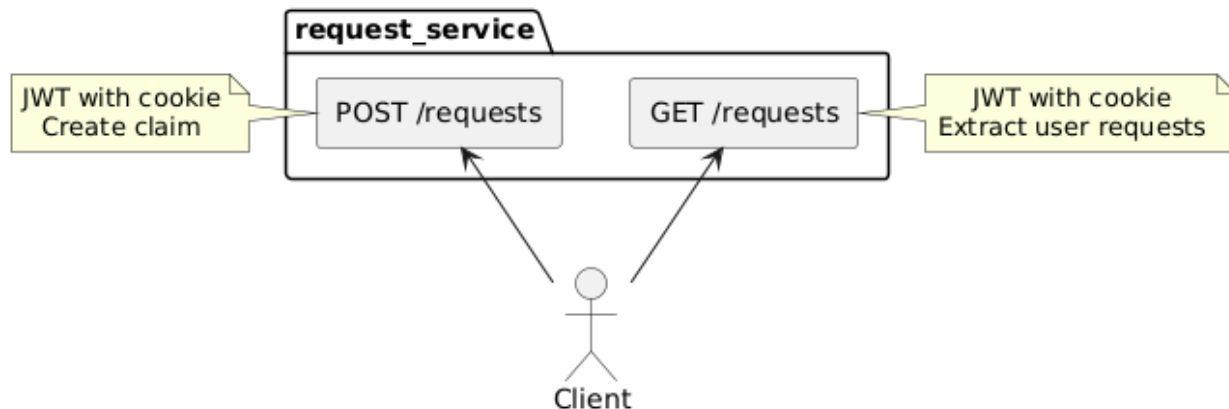


Рисунок 3.3 - Схема ендпоінтів сервісу заяв

Під час створення нової заяви до неї автоматично додається ідентифікатор користувача, витягнутий з токена, встановлюється тип, структура даних у вигляді JSON (перелік полів форми з відповідями) та статус, який призначається випадково.

Оскільки фронтенд-частини системи розгорнуті на різних портах (localhost:3000, 3001, 3002), була впроваджена підтримка Cross-Origin

Resource Sharing (CORS). Це дозволяє клієнтським застосункам безперешкодно взаємодіяти з API, надсилати авторизаційні cookie та виконувати запити будь-якими HTTP-методами. Налаштування CORS реалізоване за допомогою CORSMiddleware із вказанням списку дозволених джерел та повним набором дозволених методів.

3.3 Структура монорепозиторію та організація коду

Для реалізації мікрофронтендної архітектури було обрано підхід монорепозиторію, що складається з декількох взаємопов'язаних проєктів. Замість того, щоб мати окремі репозиторії для кожного мікрофронтенду, всі вони розміщуються в одному, що спрощує управління, керування залежностями і повторне використання коду між проєктами. Основна структура монорепозиторію наведена на рисунку 3.4.

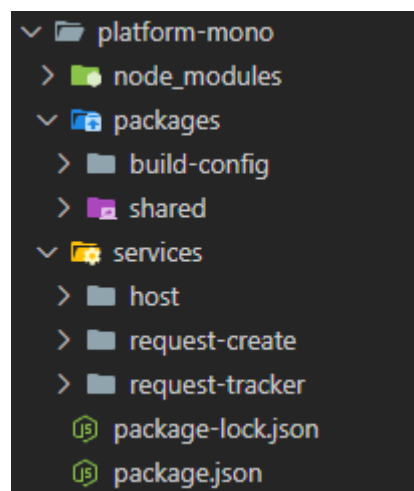


Рисунок 3.4 – Структура монорепозиторію

Основна структура монорепозиторію складається з двох ключових директорій: `packages` та `services`. Директорія `packages` містить спільні ресурси, які використовуються різними мікрофронтендами. У папці `packages` розташований підкаталог `shared`. У цьому каталозі зберігаються спільні UI компоненти, глобальні стилі, а також кастомні React хуки. Такий підхід до організації спільного коду має фундаментальне значення для підтримки

єдиного візуального стилю всього застосунку та значно спрощує процес розробки.

Директорія `services` містить підкаталоги для кожного мікрофронтенду. У кожному каталозі мікрофронтенду є свій `package.json` для встановлення локальних залежностей та `webpack.config.js` для налаштування процесу збірки кожного додатку.

Для ефективного управління залежностями між цими підпроектами та оптимізації встановлення пакетів в проєкті використовується механізм `workspaces`, що надається менеджером пакетів `npm`. Цей підхід реалізований через відповідну конфігурацію в кореновому файлі `package.json` монорепозиторію (рис. 3.5), де у властивості `workspaces` визначено шляхи до директорій, що містять окремі пакети (`packages/*` та `services/*`).

```
{
  "name": "platform-mono",
  "version": "1.0.0",
  "workspaces": [
    "packages/*",
    "services/*"
  ]
}
```

Рисунок 3.5 – Конфігурація кореневого `package.json`

При такій конфігурації, коли виконуємо команду `npm install` в кореневій директорії монорепозиторію, `npm` автоматично розпізнає всі підпроекти в директоріях `packages` та `services` як частини єдиного `workspace`. Це призводить до створення оптимізованої структури `node_modules`, де спільні залежності встановлюються лише один раз на рівні кореневої директорії, а не дублюються для кожного підпроекту. Також це забезпечує, що мікрофронтенди можуть напряду використовувати спільні компоненти з директорії `packages`, вказуючи їх як залежності у своїх `package.json` файлах. `Workspace` автоматично створює правильні посилання між цими пакетами, що робить процес розробки більш легшим.

Така організація коду сприяє підвищенню продуктивності розробки, а також забезпечує масштабованість системи в довгостроковій перспективі.

3.4 Реалізація маршрутизації та захисту роутів

У прототипі реалізована комплексна система маршрутизації з використанням бібліотеки React Router версії 7, яка поєднує декларативний підхід до опису маршрутів із гнучкою логікою контролю доступу. Основною метою впровадженої системи є забезпечення безпечного доступу до ресурсів та функціональності додатку лише для автентифікованих та авторизованих користувачів, а також підтримка чіткої та зрозумілої архітектури маршрутів для полегшення подальшого масштабування та обслуговування.

Центральним елементом системи автентифікації виступає компонент `AuthProvider`, який реалізує контекст для управління станом автентифікації користувача. Цей компонент створює єдину точку управління автентифікацією для всього додатку, надаючи доступ до відповідних даних та методів через React Context API. При першому рендері `AuthProvider` виконує перевірку поточного стану автентифікації користувача шляхом відправки запиту до серверного API. У період очікування відповіді від сервера компонент встановлює стан завантаження, що дозволяє відображати відповідний індикатор для користувача.

Для підвищення зручності доступу до контексту автентифікації розроблено кастомний хук `useAuth` (рис. 3.6). Цей хук інкапсулює логіку отримання контексту і надає простий та декларативний інтерфейс для доступу до стану автентифікації та методів управління нею з будь-якого функціонального компонента застосунку. Це робить код компонентів чистішим та зменшує повторення коду.

```
import { useContext } from "react"
import AuthContext from "../src/components/auth/AuthProvider"

function useAuth() {
  return (
    |   useContext(AuthContext)
  )
}

export default useAuth;
```

Рисунок 3.6 – Реалізація кастомного хука useAuth

Для безпосереднього захисту маршрутів реалізовано спеціальний компонент ProtectedRoute (рис. 3.7).

```
import { Navigate, Outlet } from 'react-router-dom';
import useAuth from '../../hooks/useAuth';

function ProtectedRoute() {
  const { isAuthenticated, isLoading } = useAuth();

  if (isLoading) {
    |   return <div>Loading...</div>;
  }

  return (
    |   isAuthenticated === true ? <Outlet /> : <Navigate to="/login" replace />
  )
};

export default ProtectedRoute;
```

Рисунок 3.7 – Реалізація компоненту ProtectedRoute

Цей компонент виступає обгорткою для захищених маршрутів та забезпечує додатковий рівень контролю доступу. При спробі доступу до захищеного маршруту, що обгорнутий ProtectedRoute, цей компонент перевіряє поточний стан автентифікації, використовуючи дані з AuthContext за допомогою хука useAuth. Якщо користувач не автентифікований, компонент автоматично перенаправляє його на сторінку входу. У протилежному випадку рендерить дочірні компоненти за допомогою Outlet.

Система маршрутизації в головному застосунку побудована з використанням декларативного підходу React Router. Всі маршрути визначені в компоненті `AppWrapper`, який служить центральним конфігураційним елементом для навігації. Публічні маршрути, такі як сторінки входу та реєстрації, доступні безпосередньо, тоді як всі захищені маршрути обгорнуті компонентом `ProtectedRoute`. Така структура забезпечує чітке розмежування доступу та спрощує подальше розширення системи маршрутизації.

Кожен активний мікрофронтенд не має власної внутрішньої маршрутизації. Проте, оскільки вони входять до складу захищених маршрутів, при їх завантаженні їм передається спеціальна функція `handleError` (рис. 3.8).

```
const handleError = ({ status }) => {  
  |   if (status === 401 || status === 404) navigate('/login');  
  | };
```

Рисунок 3.8 – Реалізація функції `handleError`

Функція `handleError` дозволяє централізовано обробляти помилки, які можуть виникнути у віддаленому додатку. У випадках, коли сервер повертає статуси 401 (Unauthorized) або 404 (Not Found), функція ініціює перенаправлення на сторінку входу, тим самим гарантує відповідну реакцію користувацького інтерфейсу на зміну стану сесії або втрату доступу до ресурсу.

Використання контексту автентифікації у поєднанні з компонентами для захисту маршрутів та централізованою обробкою помилок забезпечує гнучкий, надійний та добре масштабований механізм контролю доступу. Така архітектура є придатною до подальшого масштабування як у межах одного застосунку, так і при роботі з мікрофронтендами.

3.5 Інтеграція з Module Federation

У основі інтеграції мікрофронтендів у прототипі за допомогою Module Federation лежить концепція "host-remote", де головний додаток відіграє роль контейнера, який об'єднує віддалені мікрофронтенди в єдиний застосунок.

Головний додаток є центральною точкою входу ходу для користувача та основою всієї системи мікрофронтендів. У прототипі він налаштований для роботи на порту 3000 і бере на себе відповідальність за формування базової структури вебзастосунку. Це включає в себе ініціалізацію головної сторінки, управління глобальною навігацією та маршрутизацією між різними функціональними розділами, а також функцію управління станом автентифікації користувача. Від того, наскільки коректно та надійно налаштований головний додаток, залежить стабільність та узгодженість роботи всієї мікрофронтендної системи.

Ключовим елементом конфігурації головного додатку є налаштування плагіну ModuleFederation у конфігураційному файлі webpack.config.js. Для цього необхідно створити новий екземпляр класу ModuleFederationPlugin і в якості параметру передати об'єкт з окремими налаштуваннями. На рисунку 3.9 представлена частина конфігурації, а саме підключення віддалених модулів.

```
name: 'container',
remotes: {
  create: 'create@http://localhost:3001/remoteEntry.js',
  tracker: 'tracker@http://localhost:3002/remoteEntry.js'
},
```

Рисунок 3.9 – Конфігурація підключення віддалених модулів

Параметр name присвоює унікальне ім'я поточному додатку в екосистемі ModuleFederation. Об'єкт remotes встановлює зв'язок між головним додатком та віддаленими модулями. Кожен запис у цьому об'єкті визначає один віддалений модуль, який хост може імпортувати. Наприклад, параметр create це локальний псевдонім, котрий буде використовуватися в коді головного додатку для імпорту компонентів з відповідного мікрофронтенду. Значенням для цього ключа є рядок, що складається з двох частин, розділених символом

@. Перша частина create вказує на глобальне ім'я віддаленого модуля, як воно визначено у його власній конфігурації ModuleFederation. Після цієї частини вказана URL-адреса за якою головний додаток може знайти завантажити файл маніфесту віддаленого модуля. Цей файл містить метадані про експортовані модулі та їх залежності.

Далі потрібно налаштувати об'єкт shared (рис. 3.10), котрий дозволяє визначити бібліотеки, які будуть спільними між головним додатком та усіма підключеними мікрофронтедами.

```
shared: {
  ...packageDependencies.dependencies,
  react: {
    singleton: true,
    requiredVersion: packageDependencies.dependencies['react']
  },
  'react-dom': {
    singleton: true,
    requiredVersion: packageDependencies.dependencies['react-dom']
  },
  'react-router-dom': {
    singleton: true,
    requiredVersion: packageDependencies.dependencies['react-router-dom']
  },
  formik: {
    singleton: true,
    requiredVersion: packageDependencies.dependencies['formik']
  },
  yup: {
    singleton: true,
    requiredVersion: packageDependencies.dependencies['yup']
  },
}
```

Рисунок 3.10 – Конфігурація використання спільних бібліотек

Наприклад, такі бібліотеки як react, react-dom, react-router-dom, formik та yup, оголошені як спільні. Параметр singleton у значенні true гарантує, що в середовищі браузера буде завантажена та ініціалізована лише одна версія кожної із зазначених бібліотек, навіть якщо різні мікрофронтеди були розроблені з використанням дещо відмінних, але сумісних версій. Це запобігає

не тільки непотрібному додатковому завантаженню бандлів бібліотек, що збільшує дублювання коду, але й значно серйознішим проблемам, таким як конфлікти версій, особливо для бібліотек, що зберігають внутрішній стан (наприклад, ReactJS). Наступним є параметр `requiredVersion`, який чітко вказує версію спільної залежності яку очікує додаток. Webpack використовує цю інформацію під час вирішення залежностей, щоб обрати найбільш підходящу спільну версію. Якщо версії несумісні, Webpack може видати попередження або помилку під час збірки. Аналогічним чином налаштовано об'єкт `shared` і в конфігураціях Webpack для обох мікрофронтендів.

Без ретельного налаштування спільних бібліотек кожен мікрофронтенд міг би завантажувати власну копію, що призвело б до значного збільшення загального розміру завантажуваного коду для всього застосунку, довшого часу завантаження сторінки, а також до потенційних помилок, пов'язаних із несумісністю стану або контексту між різними екземплярами цих залежностей.

Мікрофронтенд створення заяв розгорнутий на окремому порту 3001. Його основне призначення - надання користувачам інтерфейсу для створення різних типів заяв. Як і головний додаток, він використовує плагін `ModuleFederation`, тільки для експорту (рис. 3.11) своєї функціональності, роблячи її доступною для споживання іншими додатками.

```
name: 'create',  
filename: 'remoteEntry.js',  
exposes: {  
  | './Forms': './src/components/forms/block/FormsModule.js'  
  },  
},
```

Рисунок 3.11 – Конфігурація експорту модуля

Параметр `name` визначає унікальне глобальне ім'я для цього мікрофронтенду. Саме це ім'я використовується головним додатком у його конфігурації `remotes` для ідентифікації та зв'язку з цим мікрофронтендом. Файл `remoteEntry.js` генерується Webpack під час збірки мікрофронтенду і містить всю необхідну метадані про експортовані модулі, їх

залежності та способи їх завантаження, дозволяючи головному додатку динамічно їх знаходити, завантажувати та інтегрувати.

Об'єкт `exposes` визначає, які саме модулі або компоненти цей мікрофронтенд робить доступними для зовнішнього використання. У даному випадку мікрофронтенд експортує компонент `FormsModule.js`. Під маскою `./Forms` буде використовуватися хостом для імпорту цього конкретного модуля. Шлях вказує на фактичне розташування компоненту всередині структури проєкту відносно конфігураційного файлу `webpack.config.js`. Аналогічно налаштований мікрофронтенд для перегляду заяв розгорнутий порту 3002.

Для завантаження віддалених модулів у головному додатку використовується стандартний для ReactJS механізм динамічного імпорту в поєднанні з `React.lazy` (рис. 3.12). Замість того, щоб завантажувати код усіх мікрофронтендів одразу при старті головного застосунку `React.lazy` дозволяє завантажувати код конкретного мікрофронтенду лише тоді, коли він дійсно потрібен. Наприклад, коли користувач переходить на відповідний маршрут, що веде до функціональності наданої цим мікрофронтендом.

```
const RequestCreate = lazy(() => import('create/Forms'));  
const RequestTracker = lazy(() => import('tracker/RequestTracker'));
```

Рисунок 3.12 – Динамічний імпорт віддалених компонентів

Webpack завдяки `ModuleFederation`, розуміє синтаксис динамічних імпортів і під час виконання запитує відповідний `remoteEntry.js` файл, знаходить там інформацію про експортований модуль мікрофронтенду і асинхронно завантажує його код та залежності.

Оскільки завантаження віддаленого модуля є асинхронною операцією, яка може зайняти деякий час, ReactJS надає компонент `Suspense`, який приймає `props`, значенням якого є JSX-елемент, що буде відображатися доти, доки

віддалений компонент не буде повністю завантажений та готовий до рендерингу.

Реалізація мікрофронтендної архітектури за допомогою Webpack Module Federation у даному прототипі наочно продемонструвала, як можна ефективно та відносно легко розділити великий і складний застосунок на менші, керовані та незалежно розроблювані частини. Ці частини працюють ізольовано, але водночас безперешкодно взаємодіють між собою, формуючи для кінцевого користувача єдине ціле. Завдяки потужним можливостям Module Federation, таким як спільне використання залежностей, експорт та імпорт модулів, а також динамічне завантаження, досягається високий рівень гнучкості, оптимізації ресурсів та покращення загальної архітектури застосунку.

3.6 Інтеграція з Single-SPA

Реалізація мікрофронтендного застосунку за допомогою фреймворку Single-SPA дещо відрізняється за підходом та інструментарієм від Module Federation, проте основна логіка та цілі залишаються схожими: необхідно забезпечити можливість розшарювання модулів або цілих застосунків, створення посилань на них з головного застосунку та їх динамічне завантаження й монтування за потреби. У випадку розробки через Module Federation, завдання розшарювання та оркестрації значною мірою покладалися на Webpack та його плагін ModuleFederation. При розробці мікрофронтендів за допомогою Single-SPA частіше використовуються стандартні браузерні можливості ES модулів у поєднанні з `import maps`.

`Import map` дозволяє контролювати, як браузер розпізнає та завантажує JavaScript-модулі. Взагалі це JSON об'єкт, що розміщується всередині скрипту у HTML документі, в даному випадку головного додатку. Ключами в цьому об'єкті є імена модулів (до яких можна звертатися для подальшого імпорту), а значеннями цих ключів є URL-адреси, за якими браузер може знайти та завантажити відповідний файл модуля. Наприклад, коли в коді з'являється

`import('request-create')`, браузер дивиться в карту імпорту, знаходить посилання і підвантажує файл як модуль. Це дозволяє абстрагувати імена модулів від їхніх фактичних шляхів розташування, що дуже зручно для мікрофронтендних архітектур.

Мікрофронтенди вбудовувались у контейнерний додаток за принципом Single-SPA parcels. Parcels в екосистемі Single-SPA пропонують більш гнучкий підхід порівняно з повноцінними Single-SPA applications. Ключова відмінність полягає у їхній незалежності від URL-маршрутизації на рівні фреймворку. Якщо традиційні Single-SPA applications зазвичай активуються при зміні URL і відповідають за певну частину сторінки, асоційовану з цим маршрутом, то parcels можуть бути змонтовані програмно в будь-якому місці та в будь-який час в межах основного застосунку.

Щоб мікрофронтенди могли бути використані як Single-SPA parcels, їх необхідно відповідним чином підготувати. Так як мікрофронтенд розроблений за допомогою ReactJS, то бібліотека `single-spa-react` значно спрощує цей процес, надаючи утиліти для обгортання ReactJS компонентів у формат, сумісний із життєвими циклами Single-SPA. На рисунку 3.13 реалізована конфігурація вхідної точки мікрофронтенду зі створення заяв для інтеграції з Single-SPA.

```
import React from 'react';
import ReactDOM from 'react-dom/client';
import singleSpaReact from 'single-spa-react';
import RequestCreate from './components/forms/block/FormsModule';

const lifecycles = singleSpaReact({
  React,
  ReactDOM,
  rootComponent: (props) => <RequestCreate {...props} />,
  renderType: 'createRoot',
  errorBoundary(err) {
    return <div>Error in RequestCreate: {err.message}</div>;
  }
});

export const { bootstrap, mount, unmount } = lifecycles;
```

Рисунок 3.13 – Конфігурація вхідної точки

Функція `singleSpaReact` приймає об'єкт конфігурації, до якого входить метод відображення компонента, сам компонент та функція `errorBoundary`, котра поверне запасний UI, якщо під час рендерингу або в життєвих циклах виникне помилка для того щоб аварійно не завершився весь додаток. По закінченню виконання функції в змінну `lifecycles` запишеться об'єкт з необхідними функціями життєвого циклу (`bootstrap`, `mount`, `unmount`). Саме ці функції знадобляться Single-SPA, щоб коректно ініціалізувати (одноразово), відрендерити (при активації) й демонтувати мікрофронтенд. Аналогічно реалізовано конфігурацію для іншого мікрофронтенду.

Для динамічного підключення та управління відображенням цих `parcels` у головному додатку реалізований компонент `MicrofrontendParcel` (Додаток А). Цей компонент інкапсулює логіку завантаження та вставлення модулів мікрофронтендів за допомогою функцій, що надаються бібліотекою Single-SPA.

Суть його роботи полягає в наступному. Під час рендерингу компонента, на основі переданої назви модуля запускається асинхронне завантаження відповідного коду з віддаленого джерела, зазначеного в `import map`. Після успішного завершення завантаження модуля, компонент `MicrofrontendParcel` використовує функцію `mountRootParcel` з Single-SPA для монтажу завантаженого `parcel` у визначений DOM-елемент

При зміні сторінки або видаленні цього компонента зі сторінки (наприклад, при переході користувача на інший розділ), відбувається автоматичне відключення та очищення ресурсу, що дозволяє уникнути витоків пам'яті та конфліктів між модулями.

Щоб мікрофронтенди могли бути коректно завантажені браузером як ES модулі та інтегровані за допомогою Single-SPA, їх конфігурацію збірку Webpack потрібно трішки модифікувати, а саме додати декілька опцій, котрі вказують, що потрібно генерувати вихідний файл як ES модуль. Ці налаштування гарантують, що Webpack створить бандл, який експортує об'єкт

з життєвими циклами (bootstrap, mount, unmount) у форматі, зрозумілому для завантажувача модулів та Single-SPA.

Далі потрібно налаштувати об'єкт externals (рис. 3.14) в кожному конфігураційному файлі Webpack окремого мікрофронтенду. Цей об'єкт вказує, що перелічені в ньому бібліотеки не потрібно включати у фінальний бандл мікрофронтенду, оскільки вони будуть надані ззовні. Це забезпечує спільне використання ключових залежностей, суттєво зменшує розмір кожного окремого бандлу мікрофронтенду та запобігає дублюванню коду.

```
externals: {
  react: 'react',
  'react-dom': 'react-dom',
  'react-router-dom': 'react-router-dom',
  formik: 'formik',
  yup: 'yup'
},
```

Рисунок 3.14 – Налаштування властивості externals

Для того щоб при ініціалізації головний додаток мав доступ до бібліотек і не завершився помилкою потрібно додати на них посилання в JSON об'єкті import map (рис. 3.15) розміщеному в HTML файлі головного додатку. Також в import map необхідно вказати посилання на точки входу кожного мікрофронтенду.

```
<script type="importmap">
{
  "imports": {
    "react": "https://esm.sh/react@18.3.1",
    "react-dom": "https://esm.sh/react-dom@18.3.1",
    "react-router-dom": "https://esm.sh/react-router-dom@7.4.0?external=react,react-dom",
    "formik": "https://esm.sh/formik@2.4.6?external=react,react-dom",
    "yup": "https://esm.sh/yup@1.6.1",

    "request-tracker": "http://localhost:3002/js/request-tracker.js",
    "request-create": "http://localhost:3001/js/request-create.js"
  }
}
</script>
```

Рисунок 3.15 – Конфігурація import map

Після того, як всі конфігурації завершені: мікрофронтеди налаштовані для експорту життєвих циклів Single-SPA, Webpack оптимізовано для створення ES модулів та коректної роботи з externals, а import map у головному HTML файлі містить всі необхідні посилання, то система готова до взаємодії.

Таким чином, описаний підхід з використанням Single-SPA, ES модулів та import map для реалізації мікрофронтедів у вигляді parcels створює гнучку, легко керовану та ефективну архітектуру.

4 ТЕСТУВАННЯ РОЗРОБЛЕНОГО ПРОТОТИПУ ТА ПОРІВНЯННЯ СТРАТЕГІЙ ВПРОВАДЖЕННЯ

4.1 Тестування прототипу на базі Module Federation

Запустимо три додатки на окремих портах: головний застосунок на порту 3000, мікрофронтенд зі створення заяв на порту 3001 та мікрофронтенд для перегляду створених заяв на 3002. Відкривши браузер та перейшовши за посиланням, котре веде на головний застосунок. Відкриємо консоль розробника перейдемо у вкладку мережа, виберемо фільтр відображати тільки скрипти. Побачимо багато JavaScript бандлів (рис. 4.1).

Name	Status	Type	Initiator	Size
bundle.16a0fe31e7f71284979d.js	200	script	load script:41	59.8 ...
bundle.176315e8297ebddf54a...	200	script	load script:41	610 kB
bundle.19d3820bd5d2c5b3acb...	200	script	load script:41	240 kB
bundle.4e516fcb66bb8769c04f.js	200	script	(index):6	98.5 ...
bundle.4eac8ca7b3e119a9ef5c.js	200	script	load script:41	112 kB
bundle.9a1181b7d9acf58363d...	200	script	load script:41	65.2 ...
bundle.9d5768be24257b42df4...	200	script	load script:41	36.1 ...
bundle.e8f06d09a62a1ad58c4...	200	script	load script:41	52.8 ...

Рисунок 4.1 – Список завантажених JavaScript бандлів

У цьому переліку є бандл для головного застосунку підключених до нього стилів, а усі інші це бібліотеки, котрі були вказані в об'єкті shared конфігурації Module Federation для спільного використання. Кожен мікрофронтенд поділився окремою бібліотекою.

Також у цьому списку є два remoteEntry.js файли (рис. 4.2) отриманих головним додатком від кожного мікрофронтенду. Саме вони будуть надавати функціонал кожного віддаленого застосунку.

Name	Status	Type	Initiator	Size
remoteEntry.js	200	script	load script:41	93.2 ...
remoteEntry.js	200	script	load script:41	93.8 ...

Рисунок 4.2 – Список завантажених remoteEntry.js файлів

Пройшовши етап автентифікації побачимо головну сторінку, де можна обрати вкладки створити заяву чи переглянути перелік вже існуючих. Наприклад, якщо перейти на вкладку створення нових заяв, то підвантажиться додатковий JavaScript файл (рис. 4.3), котрий буде містити модуль з необхідним функціоналом.

Name ▼	Status	Type	Initiator	Size
remoteEntry.js	200	script	load_script:41	93.2 kB
remoteEntry.js	200	script	load_script:41	93.8 kB
bundle.dd5a533a9fa05973ec39.js	200	script	profile:6	98.5 kB
bundle.b76c96dfb92cd4e8cfe8.js	200	script	load_script:41	49.7 kB
bundle.b043eec3ff552f6bb850.js	200	script	load_script:41	42.9 kB
bundle.9a1181b7d9acf58363d2.js	200	script	load_script:41	65.2 kB
bundle.787aaa1dcb8097f14b55.js	200	script	load_script:41	35.6 kB
bundle.4eac8ca7b3e119a9ef5c.js	200	script	load_script:41	112 kB
bundle.19d3820bd5d2c5b3acb1.js	200	script	load_script:41	240 kB
bundle.176315e8297ebddf54ae.js	200	script	load_script:41	610 kB
bundle.16a0fe31e7f71284979d.js	200	script	load_script:41	59.8 kB

Рисунок 4.3 – Імпортований модуль з мікрофронтенду

Для прикладу, якщо перестати ділитися деякими бібліотеками та залежностями, то кількість завантажуваних бандлів зменшиться, а сам розмір бандлу головного додатку та під'єднуваного збільшиться так як ці бібліотеки будуть входити до нього. На рисунку 4.4 продемонстровано розмір бандлів головного додатку та мікрофронтенду зі створення заяв.

Name	Status	Type	Initiator	Size ▼
bundle.69f2eda8d45edfdcef97.js	200	script	load_script:41	1,016 kB
bundle.33e79f407f4331657baf.js	200	script	load_script:41	198 kB

Рисунок 4.4 – Бандл головного додатку та мікрофронтенду

Отже, результати запуску застосунку демонструють, як головний контейнер взаємодіє з мікрофронтендами, динамічно підвантажуючи необхідні модулі через remoteEntry.js файли. Це підтверджує коректну роботу механізму Module Federation та ефективну організацію розділення коду.

4.2 Тестування прототипу на базі Single-SPA

Запустимо три додатки на окремих портах. Відкривши браузер та перейшовши за посиланням, котре веде на головний застосунок. Відкриємо консоль розробника перейдемо у вкладку мережа, виберемо фільтр відображати тільки скрипти. Побачимо дуже багато JavaScript бандлів (рис. 4.5).

Name	Status	Type	Initiator	Size
 bundle.a7dad5574542c8fc24d5.js	200	script	(index):22	200 kB
 formik@2.4.6?external=react,rea...	200	script	bundle.a7dad55...js:1	0.8 kB
 react@18.3.1	200	script	bundle.a7dad55...js:2	0.6 kB
 react-dom@18.3.1	200	script	bundle.a7dad55...js:3	0.7 kB
 react-router-dom@7.4.0?externa...	200	script	bundle.a7dad55...js:4	0.8 kB
 yup@1.6.1	200	script	bundle.a7dad55...js:5	0.6 kB
 dom.mjs	200	script	react-router-dom@7.4.0	2.2 kB
 react-router.mjs	200	script	react-router-dom@7.4.0	2.6 kB
 react-router-dom.mjs	200	script	react-router-dom@7.4.0	0.8 kB
 react.mjs	200	script	react-dom@18.3.1:2	4.3 kB
 scheduler@%5E0.23.2?target=es...	200	script	react-dom@18.3.1:3	0.6 kB
 react-dom.mjs	200	script	react-dom@18.3.1:4	53.0 kB
 deepmerge@%5E2.1.1?target=e...	200	script	formik@2.4.6?external=r	0.6 kB

Рисунок 4.5 – Список завантажених JavaScript бандлів

У порівнянні з підходом спільного доступу до бібліотек через об'єкт `shared` в `Module Federation` там надавався доступ до локальних бібліотек, а в цьому варіанті в `import map` вказуються адреси на віддалені сервери, де знаходяться бібліотеки. Тому одна бібліотека тягне за собою іншу залежність імпортуючи дуже багато файлів, але з іншого боку при такому підході кожен бандл віддаленого модуля не буде містити в собі спільні залежності.

Якщо прибрати завантажування спільних бібліотек через `import map`, то розміри бандлу для кожного додатку збільшаться (рис. 4.6).

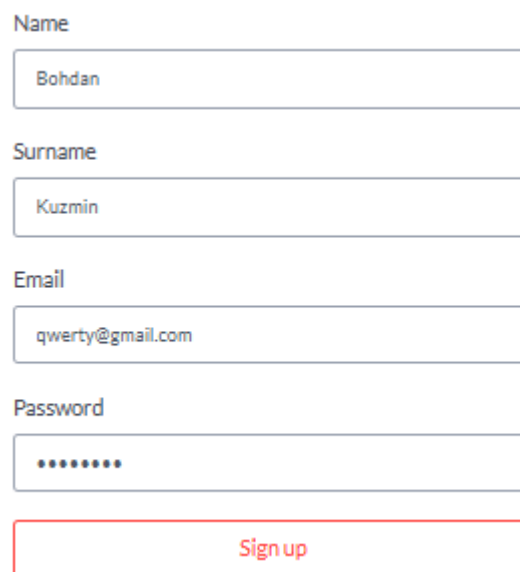
Name	Status	Type	Initiator	Size
 bundle.f23f8567c4c1ef2a64e5.js	200	script	(index):18	1,221 kB
 request-create.js	200	script	external import "request-	956 kB
 request-tracker.js	200	script	external import "request-	795 kB

Рисунок 4.6 – Бандл головного додатку та двох мікрофронтендів

При цьому застосунок не зламається через використання дублікатів одних і тих самих залежностей, особливо ReactJS, тому що кожен мікрофронтенд працює в ізольованому середовищі виконання, забезпеченому архітектурою Single-SPA. Завдяки цьому різні версії бібліотек не конфліктують між собою, оскільки мікрофронтенди не діляться спільним контекстом, як це відбувається у Module Federation. Це дозволяє кожному модулю бути самодостатнім та незалежно керувати власними залежностями, навіть якщо вони дублюються.

4.3 Тестування працездатності прототипу

Для початку потрібно зареєструвати нового користувача, для цього потрібно перейти на сторінку реєстрації (рис. 4.7).



The image shows a registration form with the following fields and values:

- Name:** Bohdan
- Surname:** Kuzmin
- Email:** qwerty@gmail.com
- Password:** (masked with dots)
- Sign up button:** A red-outlined button with the text "Sign up".

Рисунок 4.7 – Форма реєстрації

Після успішної реєстрації дані про користувача записуються до таблиці users (рис. 4.8).

id [PK] integer	email character varying	password character var	first_name character var	last_name character var
1	example@gmail.com	\$2b\$12\$...	John	Doyle
2	test@gmail.com	\$2b\$12\$...	test	test
3	qwerty@gmail.com	\$2b\$12\$e...	Bohdan	Kuzmin

Рисунок 4.8 - Дані користувачів в таблиці users

Для того щоб потрапити до головної сторінки потрібно пройти етап автентифікації ввівши електронну адресу та пароль. Повинен згенеруватись JWT та бути розміщеним в cookie (рис. 4.9).

Name	Value	Domain	Path	Expires / Max-Age	Size	HttpOnly
access_token_cookie	eyJhbGciOiJIUzI1NiI...	localhost	/	Session	160	✓

Рисунок 4.9 – Згенерований JWT

Створимо заяву (рис. 4.10) на проведення івенту за допомогою функціоналу, який реалізовано в мікрофронтенді зі створення заяв. Після переходу на відповідну вкладку в головному застосунку динамічно підвантажується модуль цього мікрофронтенду, що забезпечує форму для введення даних заяви. Заповнюємо необхідні поля, після чого дані надсилаються до бекенду, де обробляються відповідним API-ендпоінтом.

Event

Vacation

Course

Event name

Cybersecurity Conference

Date

23.05.2025

Location

San Francisco

Purpose

Gain insights on new cyber threats and defense tactics.

Submit

Рисунок 4.10 – Інтерфейс мікрофронтенду зі створення заяв

Тепер можна переглянути новостворену заяву (рис. 4.11) за допомогою функціоналу, котрий надав мікрофронтенд з перегляду заяв.

Request type

event

Reset

Event name

Filter by event name

Event date

ДД.ММ.ГГГГ

Location

Filter by location

Status

resolved

Event name	Event date	Location	Purpose	Status
Cybersecurity Conference	23.05.2025	San Francisco	Gain insights o...	resolved

← 1 →

Рисунок 4.11 – Інтерфейс мікрофронтенду з перегляду заяв

Таким чином, було протестовано ключовий функціонал мікрофронтендного застосунку: реєстрацію та автентифікацію користувача, створення заяви через відповідний мікрофронтед і подальший перегляд цієї заяви в іншому мікрофронтенді з застосуванням фільтрів. Отримані результати підтверджують коректну взаємодію компонентів системи.

4.4 Порівняння стратегій впровадження мікрофронтедів

Аналіз та практична реалізація прототипів з використанням Webpack Module Federation та Single-SPA дозволили виявити ключові особливості, переваги та недоліки кожного з підходів.

4.4.1 Концепція та архітектурний підхід

Webpack Module Federation

В основі Module Federation лежить ідея спільного використання модулів JavaScript між різними, незалежно розгорнутими застосунками безпосередньо під час виконання. Тобто Module Federation дозволяє одному застосунку динамічно завантажувати код з іншого застосунку, причому цей зв'язок може бути двостороннім. Архітектура будується навколо конфігурації модулів, що експортуються та споживаються, а також механізму shared для управління спільними залежностями.

Single-SPA

Single-SPA є JavaScript фреймворком для оркестрації та інтеграції декількох незалежних мікрофронтендних застосунків в єдиний додаток.

Ключовою концепцією є управління життєвими циклами окремих мікрофронтендів. Кожен мікрофронтенд розглядається як окремий застосунок, що реєструється в кореневій конфігурації та монтується та демонтується залежно від певних умов, часто на основі URL-маршрутизації. Single-SPA більше зосереджений на композиції застосунків як окремих одиниць, забезпечуючи їх співіснування на одній сторінці.

4.4.2 Конфігурація засобів

Webpack Module Federation

Налаштування Module Federation вимагає розуміння його основних концепцій та знань Webpack. Необхідно чітко визначити хто з додатків буде приймати, а хто відавати модулі, правильно налаштувати об'єкти `remotes`, `exposes` та `shared`.

Single-SPA

Налаштування Single-SPA включає створення кореневої конфігурації, де реєструються всі мікрофронтенди. Кожен мікрофронтенд повинен експортувати необхідні функції життєвого циклу. Також потрібно налаштувати `import map` в головному додатку, що вказує на розташування модулів, які екпортують інші додатки. Потрібно не забувати, що при використанні модулів.

4.4.3 Управління залежностями та їх спільне використання

Webpack Module Federation

Одна з переваг плагіну Module Federation це те, що він пропонує потужний механізм управління залежностями через конфігурацію `shared`. У прототипі це дозволило уникнути дублювання башатьох бібліотек, що позитивно вплинуло на розмір завантажуваних бандлів, що в свою чергу впливає на продуктивність. При використанні Module Federation не створюється ізольоване середовище і тому при не вірному налаштуванні можуть виникнути помилки зв'язані з контекстом бібліотек.

Single-SPA

Не має свого механізму управління спільними залежностями. Часто використання спільних залежностей реалізовується через налаштування `externals` у Webpack та додаванні посилань на віддалені ресурси до `import map`. Ще одним способ є використання Module Federation для управління залежностями, а Single-SPA для оркестрації мікрофронтендів. У Single-SPA кожен мікрофронтенд працює в ізольованому середовищі виконання, завдяки цьому різні версії бібліотек не конфліктують між собою.

Таким чином, Module Federation оптимальний для проєктів із централізованим управлінням залежностями, тоді як Single-SPA краще підходить для децентралізованих команд, де кожен модуль може мати власний набір бібліотек.

4.4.4 Завантаження під час виконання

Webpack Module Federation

Інтеграція відбувається шляхом динамічного імпорту модулів, експортованих віддаленими застосунками. Головний застосунок завантажує `remoteEntry.js` файл віддаленого мікрофронтенду, який містить метадані про експортовані модулі. Для завантаження компонентів використовується `React.lazy` разом з `React.Suspense` для відображення запасного UI під час завантаження. Це забезпечує завантаження мікрофронтендів лише тоді, коли вони потрібні. Також можна реалізувати через динамічні імпорти, за рахунок цього весь функціонал мікрофронтендів буде завантажений разом з головним додатком.

Single-SPA

Мікрофронтенди реєструються в кореневій конфігурації і повинні експортувати функції життєвого циклу, далі Single-SPA викликає ці функції для ініціалізації, відображення та демонтування мікрофронтендів. Завантаження коду мікрофронтенду зазвичай відбувається через динамічний імпорт, шляхи до яких можуть бути визначені в `import map`.

Загалом, обидві стратегії є ефективними засобами для впровадження мікрофронтендної архітектури. Результати аналізу та створення прототипів засвідчили, що Module Federation дозволяє напряду ділитися модулями (компонентами, бібліотеками тощо) між різними частинами мікрофронтенду під час компіляції та виконання, використовуючи можливості Webpack, тоді як Single-SPA спеціалізується на оркестрації та контролі життєвих циклів окремих застосунків. Вибір між цими підходами або їх поєднанням слід здійснювати, виходячи з глибокого розуміння архітектурних вимог, структури команд і стратегічних цілей проєкту.

ВИСНОВКИ

У рамках цієї кваліфікаційної роботи бакалавра було проведено дослідження, спрямоване на аналіз та порівняння стратегій впровадження мікрофронтендної архітектури у веб-додатках на базі бібліотеки ReactJS із використанням двох підходів: Webpack Module Federation та Single-SPA. Основною метою роботи стало вивчення теоретичних засад мікрофронтендів, практична реалізація прототипів із застосуванням зазначених технологій, а також оцінка їхніх особливостей для визначення оптимальних сценаріїв використання.

Було проведено аналіз теоретичних основ мікрофронтендної архітектури, яка є сучасним рішенням для подолання обмежень моноліту.

Розглянуто інструменти Webpack Module Federation та Single-SPA, за допомогою яких можна впроваджувати мікрофронтенди.

На основі теоретичного аналізу було спроектовано та реалізовано прототип веб-застосунку для створення та управління заявами, який складався з головного додатку та двох мікрофронтендів: для створення заяв і для їх перегляду. Прототип було реалізовано із застосуванням Webpack Module Federation та Single-SPA, що дозволило провести практичне порівняння цих стратегій. Реалізація охоплювала розробку API на базі FastAPI, інтеграцію з базою даних PostgreSQL через SQLAlchemy, а також впровадження системи автентифікації на основі JWT.

Подальші дослідження можуть бути спрямовані на впровадження глобального стану між мікрофронтендами, через спеціалізовані бібліотеки, наприклад Redux. Також можливим варіантом є розробка та впровадження CI/CD пайплайнів для автоматизованої збірки та розгортання мікрофронтендних застосунків. Додатково можна реалізувати гібридну систему з застосуванням інших фреймворків: Angular та Vue.

ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАННЯ

1. Mezzalana, L. "Building Micro-Frontends". O'Reilly Media, 2021. 317 с.
2. Micro Frontends. Стаття [Електронний ресурс] URL: <https://martinfowler.com/articles/micro-frontends.html> (Дата останнього звернення: 25.03.2025)
3. Глибоке занурення в архітектуру мікрофронтендів з React.js. Стаття [Електронний ресурс] URL: <https://javascript.org.ua/gliboke-zanurenniya-v-arhitekturu-mikrofrontendiv-z-react-js/> (Дата останнього звернення: 28.03.2025)
4. Побудова незалежних застосунків з використанням мікрофронтендів. Переваги та недоліки різних рішень. Сайт [Електронний ресурс] URL: <https://dou.ua/forums/topic/43354/> (Дата останнього звернення: 28.03.2025)
5. What are Micro Frontends? Definition, Uses, Architecture. Сайт [Електронний ресурс] URL: <https://www.geeksforgeeks.org/what-are-micro-frontends-definition-uses-architecture/> (Дата останнього звернення: 31.03.2025)
6. Micro frontend architecture: A strategic solution for enterprises. Сайт [Електронний ресурс] URL: <https://www.n-ix.com/micro-frontend-architecture/> (Дата останнього звернення: 31.03.2025)
7. Understanding Server-Side and Client-Side Composition in Microfrontends. Стаття [Електронний ресурс] URL: <https://article.arunangshudas.com/understanding-server-side-and-client-side-composition-in-microfrontends-55814b5efe20> (Дата останнього звернення: 05.04.2025)
8. Micro Frontends Patterns: Server Side Composition. Стаття [Електронний ресурс] URL: <https://dev.to/okmttdhr/micro-frontends-patters-13-server-side-composition-1of5> (Дата останнього звернення: 10.04.2025)
9. Micro Frontends Patterns: Client Side Composition. Стаття [Електронний ресурс] URL: <https://dev.to/okmttdhr/micro-frontends-patterns-5gc2> (Дата останнього звернення: 12.04.2025)

10. Micro frontend Architecture using Runtime Configuration. Стаття [Електронний ресурс] URL: <https://technicoded.com/micro-frontend-architecture> (Дата останнього звернення: 15.04.2025)
11. Webpack. Сайт [Електронний ресурс] URL: <https://webpack.js.org/> (Дата останнього звернення: 22.04.2025)
12. Webpack Contributors. Module Federation. Сайт [Електронний ресурс] URL: <https://webpack.js.org/concepts/module-federation/> (Дата останнього звернення: 28.04.2025)
13. Single-SPA Team. Getting Started with single-spa. Сайт [Електронний ресурс] URL: <https://single-spa.js.org/docs/getting-started-overview/> (Дата останнього звернення: 03.05.2025)
14. Breaking Monoliths: Exploring Micro-Frontends with Single-Spa. Стаття [Електронний ресурс] URL: <https://medium.com/deutsche-telekom-gurgaon/breaking-monoliths-exploring-micro-frontends-with-single-spa-ec6456a6b4d2> (Дата останнього звернення: 10.05.2025)
15. Minnick, C. "Beginning ReactJS Foundations Building User Interfaces with ReactJS: An Approachable Guide". Wiley, 2022. 512 с.
16. ReactJS. Сайт [Електронний ресурс] URL: <https://react.dev/learn> (Дата останнього звернення: 12.05.2025)
17. React Router. Сайт [Електронний ресурс] URL: <https://reactrouter.com/home> (Дата останнього звернення: 16.05.2025)

ДОДАТОК А

Лістинг 1 – Реалізація MicroFrontendParcel.js

```
import { useEffect, useRef } from 'react';
import { mountRootParcel } from 'single-spa';

const moduleLoaders = {
  'request-create': () => import('request-create'),
  'request-tracker': () => import('request-tracker'),
};

function MicrofrontendParcel({ moduleName, style = {},
                             ...props }) {

  const containerRef = useRef(null);
  useEffect(() => {
    let parcel;
    let isMounted = true;
    const load = moduleLoaders[moduleName];
    load().then(app => {
      if (isMounted && containerRef.current) {
        parcel = mountRootParcel(app, {
          domElement: containerRef.current,
          ...props,
        });
      }
    });
  });
  return () => {
    isMounted = false;
    if (parcel && parcel.unmount) {
      parcel.unmount();
    }
  };
}, [moduleName]);

return <div ref={containerRef} style={style} />;
}

export default MicrofrontendParcel;
```