Міністерство освіти і науки України

НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ «КИЄВО-МОГИЛЯНСЬКА АКАДЕМІЯ»

Кафедра інформатики факультету інформатики

Магістерська робота

освітній ступінь – магістр

На тему: **«Мікрофронтенд-фреймворк для побудови масштабованих вебзастосунків»**

Виконав: студент 2-го року навчання,

освітньо-наукової програми «Інженерія програмного забезпечення», 121

Мисько Юрій Мирославович

Керівник Нагірна А.М., _____

кандидат фіз.-мат. наук, доцент

Рецензент _____

(прізвище та ініціали)

Магістерська робота захищена/не захищена

з оцінкою _____

Секретар ЕК _____

«____» _____ 2026 р.

Київ – 2026

Міністерство освіти і науки України

НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ «КИЄВО-МОГИЛЯНСЬКА АКАДЕМІЯ»

Кафедра інформатики факультету інформатики

ЗАТВЕРДЖУЮ

Кандидат фіз.-мат. наук, доцент

_____ Нагірна А.М.

(підпис)

«____» _____ 2026 р.

ІНДИВІДУАЛЬНЕ ЗАВДАННЯ на

магістерську роботу

студенту 2 р. н. магістерської програми Інженерія програмного забезпечення

Миську Юрію Мирославовичу

Мікрофронтенд-фреймворк для побудови масштабованих вебзастосунків

Зміст текстової частини до магістерської роботи:

Перелік скорочень, умовних позначень та термінів

Анотація

Вступ

Розділ 1. Теоретичні основи мікросервісної та мікрофронтенд архітектур

Розділ 2. Аналіз існуючих рішень та формування вимог

Розділ 3. Мікрофронтенд-фреймворк для побудови масштабованих вебзастосунків

Розділ 4. Реалізація фреймворку

Розділ 5. Демонстраційний застосунок

Висновки

Перелік посилань

Дата видачі:

«____» _____ 2026 р.

Керівник:

к.ф.-м.н., доцент Нагірна А. М. _____

(підпис)

Завдання отримав:

Мисько Ю. М. _____

(підпис)

Тема:Мікрофронтенд-фреймворк для побудови масштабованих вебзастосунків

Календарний план виконання роботи:

№ з/п	ПЕРЕЛІК РОБІТ	Термін виконання	Примітки
1.	Вибір теми, затвердження її на засіданні кафедри та закріплення наукового керівника Узгодження календарного графіка підготовки кваліфікаційної роботи. Ознайомлення студента з критеріями оцінювання кваліфікаційної роботи (п. 8.5).	10 вересня	
2.	Вивчення джерел літератури, матеріалів архівів, періодичних видань, збір та узагальнення фактів, даних	жовтень – грудень	
3.	Складання плану каліф. роботи та узгодження з науковим керівником	грудень	

№ з/п	ПЕРЕЛІК РОБІТ	Термін виконання	Примітки
4.	Написання розділів роботи *або* Постановка експерименту, аналіз отриманих результатів наукового дослідження	грудень – березень	
5.	Проміжний контроль виконання роботи	лютий	
6.	Написання кваліфікаційної роботи в цілому, ознайомлення з її першим варіантом наукового керівника	січень – березень	
	Розділ 1. Теоретичні основи мікросервісної та мікрофронтенд архітектур		
	Розділ 2. Аналіз існуючих рішень та формування вимог		
	Розділ 3. Проєктування архітектури фреймворку		
	Розділ 4. Реалізація фреймворку		
	Розділ 5. Демонстраційний застосунок		
7.	Повне завершення написання кваліфікаційної роботи, оформлення її згідно з вимогами й подання на відгук науковому керівнику	квітень – початок травня	
8.	Подання кваліфікаційної роботи для перевірки письмових робіт студентів НаУКМА на відповідність вимогам академічної доброчесності	середина травня	
9.	Подання на зовнішню рецензію	до 6 квітня	
10.	Підготовка до захисту кваліфікаційної роботи на засіданні кафедри: написання доповіді та виготовлення ілюстративного матеріалу	до 27 квітня	
11.	Попередній захист кваліфікаційної роботи на засіданні кафедри	27 квітня	
12.	Подання кваліфікаційної роботи на кафедру з усіма супроводжувальними документами	до 20 травня	
13.	Публічний захист кваліфікаційної роботи перед екзаменаційною комісією	4-5 червня	

Студент _____

Керівник _____

“_____” _____

Зміст

Список умовних позначень

DDD - Предметно-орієнтоване проєктування (domain-driven design)

API - це публічний інтерфейс додатків(Application Programming Interface)

DOM - Об'єктна модель документа (Document Object Model)

REST - Передача репрезентативного стану(Representational State Transfer)

RPC - Віддалений виклик процедури(Remote Procedure Call)

CI/CD pipeline - конвеєр безперервної інтеграції та розгортання

RxJS - бібліотека для роботи з асинхронними потоками(Reactive Extensions for JavaScript)

SPA - односторінковий додаток (Single Page Application)

Анотація

Ця робота присвячена розробці мікрофронтенд-фреймворку для побудови масштабованих веб-застосунків.

У першій частині розглянуто теоретичні основи мікросервісної та мікрофронтенд архітектури. Проаналізовано шість основних принципів мікросервісів та їх застосування у фронтенд розробці. Описано стратегії поділу застосунків та взірці комунікації в розподілених системах.

У другій частині проведено порівняльний аналіз існуючих мікрофронтенд фреймворків. Виявлено їх основні недоліки та сформульовано вимоги до фреймворку.

У третій частині спроєктовано архітектуру фреймворку, яка складається з системи типобезпечних контрактів, шини подій, механізму оркестрації модулів та підтримує вертикальний і горизонтальний поділ застосунків.

У четвертій частині описано реалізацію ключових компонентів: реєстру API, шини подій, алгоритму топологічного сортування, менеджера завантаження, Webpack-плагіна, інструментів розробника та тестування.

П'ята частина містить опис демонстраційного застосунку: редактору документів, який складається з п'ятьох мікрофронтендів на чотирьох різних технологіях.

Ключові слова: фронтенд, мікрофронтенди, мікросервісна архітектура, Module Federation, TypeScript, типобезпечна комунікація, контракти, подієво-орієнтована архітектура, незалежне розгортання, топологічне сортування.

Вступ

Монолітна архітектура фронтенд застосунків часто буває проблемою для великих команд розробників. Єдина кодова база, єдиний конвеєр збірки та єдиний цикл розгортання призводять до експоненціального зростання часу розробки, збільшує кількість конфліктів при паралельній розробці та ускладнює впровадження архітектурних змін без ризику порушення наявної функціональності.

У серверній розробці зазначені проблеми уже давно вирішуються за допомогою мікросервісної архітектури: незалежне розгортання компонентів, слабку міжмодульну зв'язаність та ізоляцію збоїв.

Зростання складності фронтенд застосунків та розміру команд розробників призвело до необхідності застосування мікросервісних принципів до клієнтського частини системи. Проте стандартизованих та зрілих рішень для побудови архітектури мікрофронтендів досі бракує, а наявні інструменти мають різні обмеження.

Завдання даного дослідження охоплюють: адаптацію принципів мікросервісної архітектури до умов браузерного середовища виконання; порівняльний аналіз існуючих рішень та формулювання вимог до фреймворку; проєктування архітектури мікрофронтенд фреймворку на основі сформованих вимог, реалізація та апробація фреймворку.

Апробацію результатів здійснено на тестовому засосунку: текстовий редактор з п'ятьма модулями, реалізованими одночасно на React, Vue, Svelte і TypeScript.

Розділ 1. Теоретичні основи мікросервісної архітектури

Еволюція фронтенд-архітектури

Монолітні фронтенд застосунки

Традиційний підхід до розробки фронтенд застосунків останніх років - це монолітна архітектура, де вся логіка зосереджена в одній кодовій базі і збирається в єдину збірку, що містить весь необхідний код для відображення і роботи інтерфейсу користувача.

В основному така архітектура була перш за все зумовлена обмеженнями браузера, а саме ранніх версій HTTP протоколу. Оскільки HTTP версії 1.1 дозволяв зробити обмежену кількість паралельних з'єднань, тому завантаження великої кількості ресурсів призводило до черг та затримок. Єдина збірка вирішувала цю проблему, об'єднуючи велику кількість модулів у один файл.

Основним недоліком монолітних фронтенд систем є складність масштабування розробки. При зростанні розміру системи час збірки суттєво збільшується, що сповільнює цикл розробки. Також якщо над проєктом працює багато розробників, то виникають конфлікти при злитті коду та складна координація змін. Також крива навчання нових розробників збільшується, оскільки часто буває важко зрозуміти архітектури великих монолітних застосунків. Така архітектура також обмежує гнучкість у виборі технологій розробки, адже неможливо використовувати різні фреймворки для різних частин застосунку.

Компонентна архітектура

Компонентна архітектура широко використовується у сучасних фронтенд фреймворках таких як: React, Vue, Angular. Компоненти - це незалежні модулі, які інкапсулюють певну логіку, стан та візуальне представлення. Розбиття інтерфейсу на компоненти покращило і спростило організацію коду. Кожен компонент відповідає за конкретну частину функціональності.

Проте компонентна архітектура не вирішила проблеми масштабування застосунку та команд. Незважаючи на модульність, застосунок все ще збирається в єдину збірку, хоча зараз все частіше використовують підхід розбиття збірки, та підвантажування решти логіки, якої не потрібно для початкової інтеракції користувача пізніше. Але навіть з таким підходом система все ще розгортається, як монолітна одиниця. Це означає, що зміни в одному компоненті потребують повного перезбирання та переразгортання всього застосунку. А також тестування всього застосунку.

Перехід до мікрофронтендів

Зростання складності фронтенд-застосунків та розміру команд розробників призвело до необхідності застосування мікросервісних принципів до клієнтського частини системи. Проблема координації між великими фронтенд командами стає критичною при масштабуванні проєкту. Коли велика кількість розробників працюють над однією кодовою базою, виникають конфлікти злиття, процес валідації коду ускладнюється та уповільнюєця процес розробки. Кожна зміна потребує узгодження з іншими командами, що знижує швидкість релізу нової функціональності. Також різні частини застосунку можуть мати різні вимоги до інструментів розробки зумовлених як бізнес потребами, так і експертизою команди розробки.

Архітектура мікрофронтендів частково вирішує ці проблеми і сприяє горизонтальному маштабуванні системи.

Теоретичні основи мікросервісної архітектури

Мікрофронтенд архітектура - це спроба застосувати основні принципи та концепції класичної мікросервісної архітектури до клієнтської сторони застосунків. Тож щоб зрозуміти, як саме потрібно проєктувати мікрофронтенди, варто спершу дослідити, що ж таке мікросервісна архітектура і які саме принципи можна перенести до клієнтської сторони застосунків, враховуючи їх специфіку і обмеження.

Мікросервіси - це невеликі автономні сервіси, які працюють разом[2]. Кожен мікросервіс виконує свою функцію, має власну кодову базу, часто розробляється окремими командами і може бути розгорнутим незалежно від інших сервісів.

Принципи мікросервісної архітектури

Сем Ньюман у всоїй книзі “Building Microservices” виділяє декілька ключових принципів мікросервісної архітектури.

Незалежність розгортання - це один з ключових принципів мікросервісної архітектури, який означає, що кожен сервіс повинен бути доступним до оновлення без впливу на інші сервіси. Це важливо відразу з декількох причин: по перше це значно пришвидшує розробку великих систем, оскільки кожен артефакт розгортається незалежно, тож на відмінну від монолітної системи, нам не потрібно чекати на збірку всього застосунку, чи тестувати весь застосунок, а лише окремий сервіс. Тобто кожен сервіс має власний CI/CD pipeline. Також незалежне розгортання сервісів знижує ризик того, що ціла система не зможе бути розгорнута, через якусь помилку в

одному з сервісів, що часто трапляється в монолітних застосунках і може заблокувати випуск нового функціоналу. Основною складністю з якою можна стикнутись при спробі застосувати цей принцип в мікрофронтенд архітектурі, може стати перевикористання спільних залежностей між різними мікрофронтендами, адже без цього розмір нашої збірки стане в рази більшим, що вплине на швидкодію застосунку. Але цю проблему уже вирішили у різних популярних системах збірки таких як Webpack, rspack за допомогою додавання спеціального плагіну.

Організація навколо бізнес домену. Часто цей принцип пов’язують саме з використанням DDD підходу, тобто велику систему розбивають на менші підсистеми з прив’язкою до предметної області. І за розробку і підтримку кожної з цих підсистем відповідає конкретна команда. У контексті DDD такі підсистеми називають bounded context і кожен мікросервіс часто є власне окремим bounded context.

Децентралізоване керування даними. У контексті мікросервісної архітектури це означає, що кожен мікросервіс володіє своїм власним сховищем даних. Недоліком такого підходу є те що у певних випадках нам необхідно зберігати однакові чи схожі дані у декількох мікросервісах, що у свою чергу сприяє дублікації даних, але такий компроміс часто вважають виправданим, оскільки система стає стійкішою до помилок, оскільки помилка в одному сховищі даних не спричинить падіння всієї системи. Якщо ж спробувати перенести цей принцип на клієнтську частину, де у ролі сховища даних часто виступає збережений в оперативній пам’яті додатку, або ж рідше збережений на диск, за допомогою, наприклад браузерного API (localstorage, indexDB, cookie тощо) об’єкт чи об’єкти, які відображають поточний стан застосунку чи локальну копію даних отриманих з сервера. Тут виникає певна дилема, оскільки всі мікрофронтенди виконуються в єдиному браузерному

контексті і мають доступ до DOM, і до об'єкта window, який є частиною DOM API, спільних глобальних змінних та браузерних сховищ. Технічно будь-який мікрофронтенд може отримати доступ до стану іншого, що робить ізоляцію радше конвенцією, ніж технічним обмеженням.

Наступним важливим принципом є приховування деталей імплементації. Це особливо важливо, коли мікросервіси комунікують між собою напряму. Приховування деталей імплементації сприяє слабкій зв'язаності та мінімальним залежностям між сервісами. У контексті мікросервісної архітектури це означає, що сервіси експортують лише ту логіку, яка дійсно потрібна через публічний API (контракти), приховуючи деталі імплементації. Наприклад, це може бути REST API, схеми контрактів для RPC чи GQL, тощо. І як перенести і застосувати цей принцип до фронтенду це те що буде розглядатись в наступних розділах.

Ізоляція відмов. Цей принцип також є однією з основних переваг мікросервісної архітектури. Він говорить нам про те, що збій одного сервісу не повинен впливати на всю систему і система повинна продовжувати працювати хоч і з обмеженим функціоналом. В контексті серверної розробки для цього використовують різні патерни такі як Circuit Breaker pattern, Graceful degradation тощо. Ці патерни не дуже релевантні в контексті фронтенд розробки, тому як застосувати цей принцип до фронтенду це також те що буде розглядатись в наступних розділах детальніше.

Концепція мікрофронтендів

Мікрофронтенди - це нова архітектура, натхненна архітектурою мікросервісів.
Основна ідея полягає в тому, щоб розбити монолітну кодову базу на менші частини, що дозволяє організувати роботу між автономними командами, незалежно від того, чи розташовані вони разом, чи розподілені.[1]

Ключовою характеристикою мікрофронтендів є незалежність їх розробки та розгортання. Кожен мікрофронтенд може мати власний репозиторій, систему збірки, CI/CD pipeline. Композиція мікрофронтендів може відбуватися, як під час збірки застосунку так і під час виконання програми. Класична структура мікрофронтендів складається з основному додатку, який виконує роль оркестратора і координує завантаження та інтеграцію інших мікрофронтендів та, власне, незалежні мікрофронтенди.

Для створення маштабованих застосунків також потрібно забезпечити належну комунікацію між мікрофронтендами, а також можливість перевикористання спільних залежностей та бібліотек. Для цього часто використовуються мікрофронтенд фреймворки.

Переваги використання мікрофронтендів

Мікрофронтендна архітектура надає командам розробників значну автономію, що впливає на швидкість створення нового функціоналу. Кожна команда може працювати над своїми модулями незалежно. Також команди розробки мають повну технологічну свободу і можуть розробляти свої модулі використовуючи технології, які їм підходять найбільше. Також така архітектура сприяє ізоляції збоїв. Якщо один мікрофронтенд викидує помилку, то інші частини додатку продовжують працювати, тим самим

покращуючи користувацький досвід. З точки зору масштабування, мікрофронтенди дозволяють легко додавати нові команди та розподіляти відповідальність між ними.

Недоліки використання мікрофронтендів

Звісно архітектура мікрофронтендів не є срібною кулею і підходить далеко не всім проєктам. Перш за все через складність налаштування інфраструктури. Також виникає проблема дублювання залежностостей, якщо кілька модулів використовують, наприклад, React, то користувач може завантажувати одні й ті ж бібліотеки кілька разів, що збільшує розмір зібрки. Складність тестування інтеграцій також зростає, адже потрібно забезпечити end-to-end та інтеграційне тестування взаємодії між різними мікрофронтендами. Без правильно узгоджених контрактів та дотримання спільних практик до написання коду, різні частини системи можуть конфліктувати та бути не консистентними.

Способи імплементації мікрофронтенд архітектури

Існує декілька різних підходів для реалізації мікрофронтенд архітектури, які відрізняються в першу чергу часом інтеграції модулів і способом їх завантаження, а також рівнем ізоляції між модулями.

Інтеграція на етапі збірки

Інтеграція на етапі збірки - є найпростішим підходом, в якому мікрофронтенди об'єднуються в єдину збірку на етапі компіляції. Кожен мікрофронтенд розробляється як окремий пакет, який публікується в

репозиторій. Основний застосунок імпортує ці пакети, як звичайні залежності і збирає їх разом. При зміні мікрофронтенду публікується його нова версія, після чого основний застосунок оновлює цю залежність та збирається весь застосунок. Фінальна збірка містить код всіх мікрофронтендів разом. Такий підхід дуже простий у реалізації, адже використовуються стандартні інструменти веб розробки. Також сприяє гарній оптимізації, адже ще на етапі збірки можна оптимізувати всю кодову базу разом. Але через відсутність незалежного розгортання - зміна одного мікрофронтенду вимагає перебудови та розгортання всього застосунку, що порушує принцип незалежності і команди не можуть розгортати зміни незалежно одна від одної. Також повільний цикл зворотного зв’язку: для тестування зміни потрібно перебудувати весь застосунок.

Інтеграція на етапі збірки по суті не реалізує справжню мікрофронтенд архітектуру, оскільки порушує ключовий принцип незалежності розгортання. Цей підхід більше схожий на модульний монолітний застосунок з хорошою організацією коду.

Інтеграція під час виконання програми

Інтеграція під час виконання програми є підходом, в якому мікрофронтенди завантажуються динамічно під час виконання застосунку. Основний застосунок не містить коду окремих мікрофронтендів під час збірки, а завантажує їх з віддалених серверів в браузері користувача. Цей підхід забезпечує справжню незалежність розгортання. Кожен мікрофронтенд будується та розгортається на власний сервер або як окрема JavaScript збірка. Основний застосунок містить конфігурацію з адресами мікрофронтендів та логіку їх динамічного підвантаження. Коли потрібен певний модуль, основний застосунок завантажує JavaScript збірку з сервера і та автоматично

виконується. Основним недоліком такого підходу є складність перевикористання залежностей між мікрофронтендами, але ця проблема уже вирішена у сучасних системах збірки, таких як, наприклад, Webpack за допомогою Module Federation.

Module Federation

Module Federation - це сучасний підхід в системі збірки Webpack, який дозволяє кільком окремим збіркам формувати єдиний застосунок. Ці окремі збірки діють як контейнери та можуть надавати та споживати код один одному, створюючи єдиний, уніфікований застосунок. Це часто називають мікрофронтендами. [4]

Переваги:

1. Справжня незалежність розгортання: різні команди можуть працювати та розгортати зміни паралельно.
2. Швидкий цикл зворотного зв’язку: зміни доступні одразу після розгортання одного модуля, без необхідності розгортання всього застосунку.
3. Підтримка A/B тестування та ранніх релізів.Можна завантажувати різні версії для різних користувачів
4. Технологічна незалежність. Кожна команда сама вибирає технології з якими буде працювати.
5. Підтримка перевикористання залежностей між різними збірками мікрофронтендів.

Недоліки:

1. Складна інфраструктура.
2. Потрібно створити механізми для комунікації між модулями.

3. Потенційні проблеми з продуктивністю, велика кількість збірок, які підвантажуються під час виконання програми - можуть сповільнити її.

Стратегії розділення мікрофронтендів

Є дві основні стратегії розподілу системи на мікрофронтенди: горизонтальне і вертикальне розділення. Горизонтальне розподілення часто використовується для застосунків без навігації, коли весь функціонал знаходиться в межах одного інтерфейсу чи одної сторінки(SPA). Такі веб застосунки мають схожий до нативних додатків досвід користувача, наприклад Google Docs, Figma, тощо. При горизонтальному розподіленні один інтерфейс ділиться на кілька мікрофронтендів. На рисунку 1 зображений схематичний приклад такого веб застосунку. Серед переваг можна виділити максимальну модульність, один мікрофронтенд можна перевикористати в інших місцях чи на інших сторінках. А серед недоліків це складність у координації стану застосунку між різними мікрофронтендами.

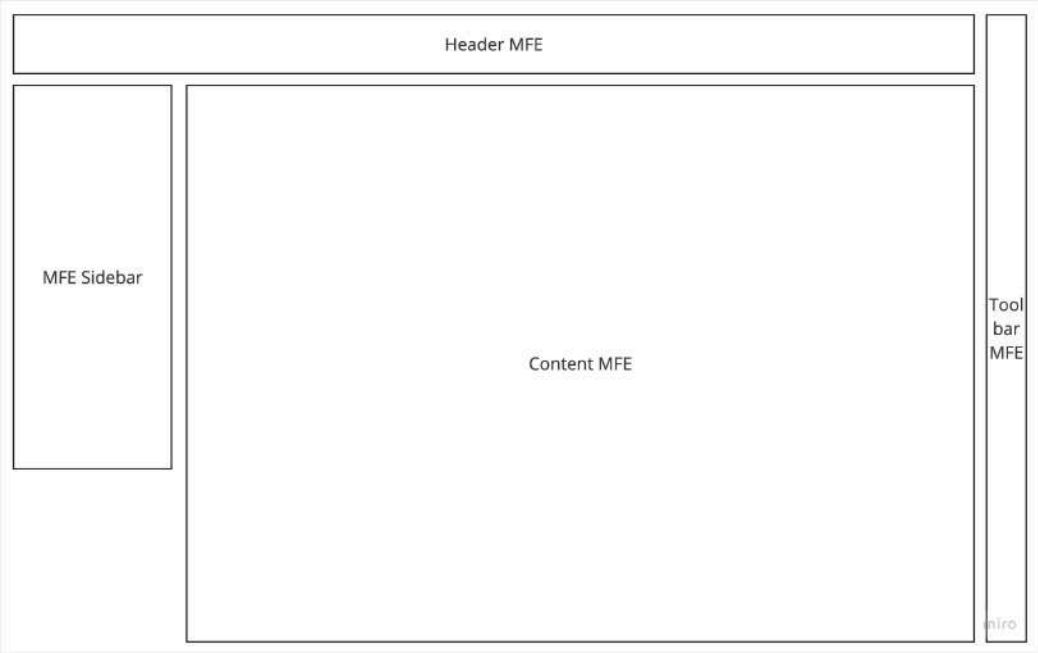

Рисунок 1. Горизонтальний поділ на мікрофронтенди

Вертикальне розподілення у свою чергу часто застосовується для багатосторінкових застосунків. У такому випадку розподіл відбувається за бізнес-доменами, часто кожен інтерфейс чи сторінка є окремим мікрофронтендом. На рисунку 2 зображений схематичний приклад такого поділу. Серед переваг такого підходу можна виділити природне роздідення по доменах та простіша кординація між мікрофронтендами, адже не потрібно думати про координацію стану. Також такий підхід може мати кращу продуктивність, адже можна динамічно підвантажувати лише потрібні мікрофронтенди в залежності від адреси користувача. Проте виникає складність як перевикористовувати спільні компоненти в інших мікрофронтендах. Наприклад один компонент “Header” має бути присутній на різних сторінках сайту, тому щоб запобігти дублюванню коду, потрібно придумати спосіб перевикористання компонентів. Це може бути, наприклад, спільна бібліотека компонентів інтерфейсу.

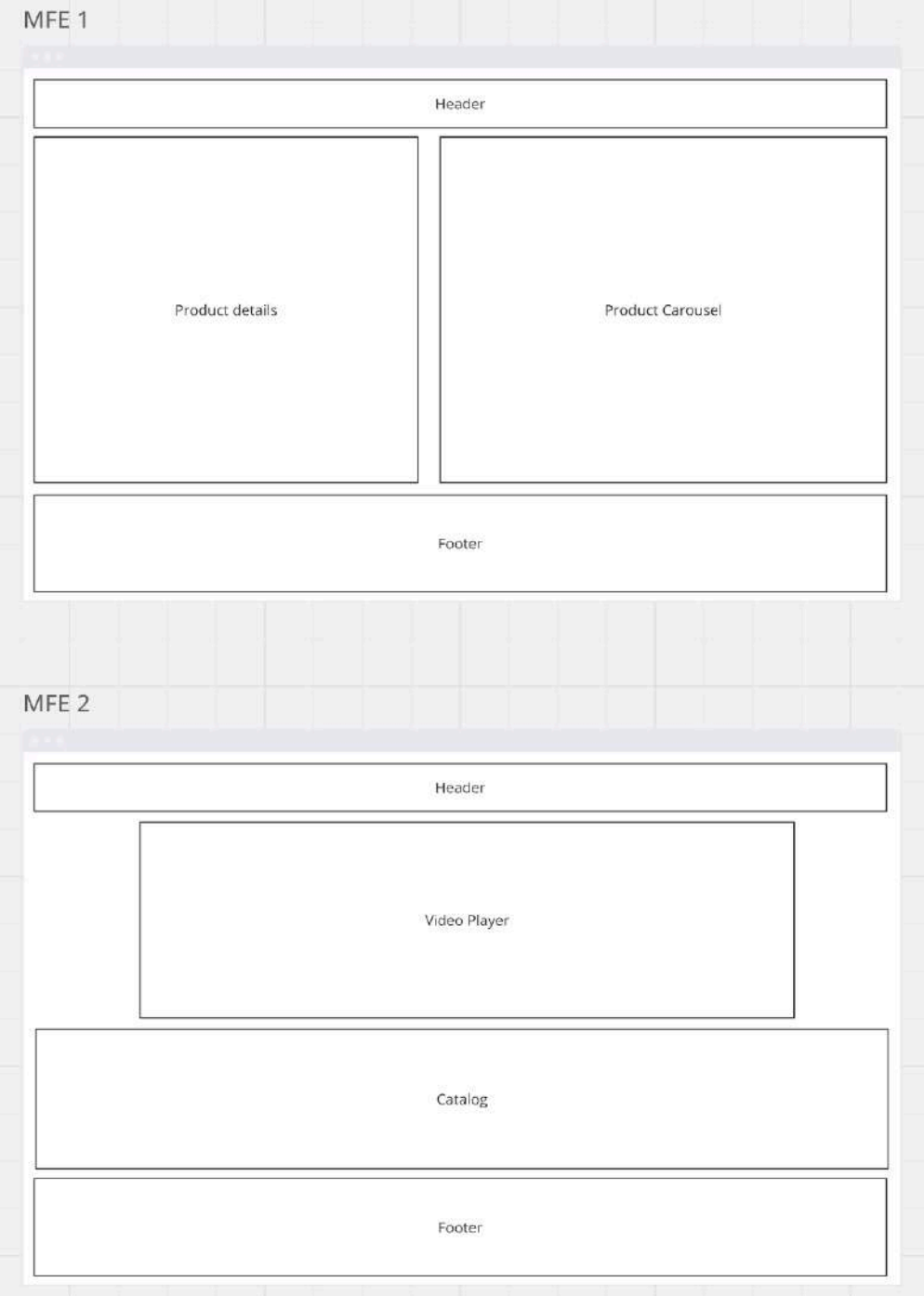

Рисунок 2. Вертикальний поділ на мікрофронтенди

Існує також і гібридна стратегія, суть полягає у комбінуванні вертикального та горизонтального поділу. Так, наприклад, у нас може бути

вертикальний розподіл за сторінками, проте в межах сторінки ми перевикористовуємо інші мікрофронтенди. На Рисунку 3 зображене схематичне представлення такого підходу. Бачимо два окремих мікрофронтенди: “Сторінка продукту” та “Сторінка списку продуктів” і кожен з них перевикористовує спільні мікрофронтенди “Header” та “Footer”. Таким чином можна отримати переваги одразу двох стратегій поділу, проте потребує більше часу та зусиль для налаштування інфраструктури.

Рисунок 3. Гібридний поділ на мікрофронтенди

Патерни комунікації в розподілених системах

Подійно-орієнтована архітектура

Подійно-орієнтована архітектура є архітектурним патерном, в якому компоненти системи взаємодіють через створення та обробку подій. Так на відміну від традиційної синхронної комунікації, де компоненти очікують на відповідь від інших компонентів, в event-driven підході взаємодія відбувається асинхронно через публікацію та підписку на події. На Рисунку 4 зображено схематичне представлення цього патерну.

Подійно-орієнтована архітектура складається з трьох основних компонентів: Event Bus, генератори подій та обробники подій.
Шина подій (Message Broker) - це центральний компонент, який виступає посередником між двома іншими. Він відповідає за отримання, зберігання та доставку подій всім зацікавленим обробникам подій.
Генератори подій (Видавці) - генерують події при зміні стану компонентів.
Події - це об’єкти, які містять інформацію про те, що саме сталось. Дальше події передаються в Event Bus, який розсилає їх обробникам подій.
Обробники подій (Підписники) - це компоненти, які зацікавленні в певних видах подій і реєструють свої обробники цих подій в шину подій.

Основною перевагою такої архітектури є слабка зв'язаність. Компоненти системи не мають прямих залежностей один від одного і не викликають методи одне одного безпосередньо. Натомість вони взаємодіють через абстракцію подій, що дозволяє змінювати та оновлювати компоненти без впливу на інші частини системи. Новий компонент може бути доданим чи

видаленим за допомогою підписки чи відписки на існуючі події, без необхідності змін у існуючих компонентах системи.

Також така архітектура сприяє відмовостійкості системи. Якщо один з обробників подій недоступний, це не впливає на роботу генераторів подій та інших обробників. Event bus може зберігати необроблені події та повторно доставляти їх після відновлення роботи компонента.

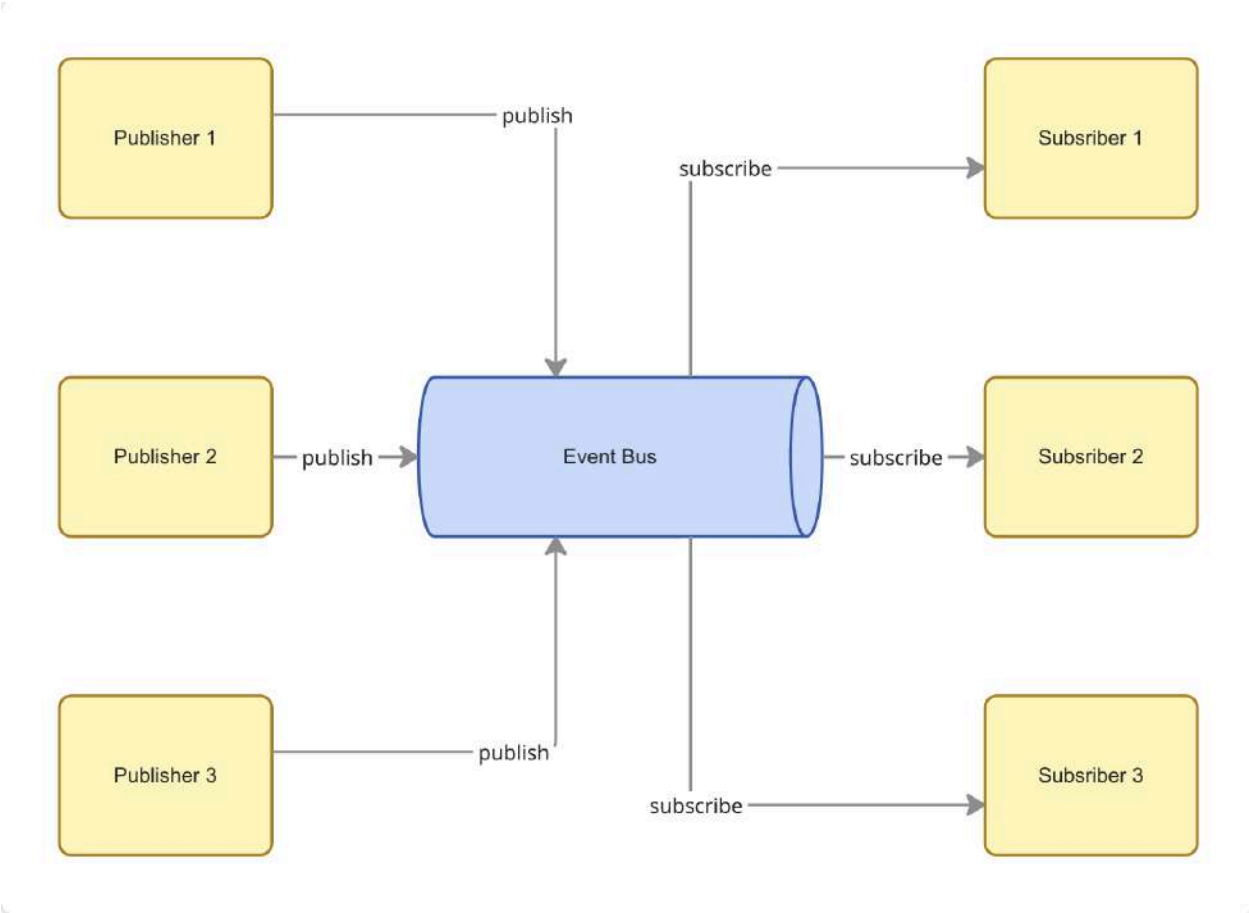

Рисунок 4. Схематичне зображення патерну pub/sub

В архітектурі мікрофронтендів, подійно-орієнтована архітектура дозволяє незалежним модулям користувацького інтерфейсу (мікрофронтендам) комунікувати між собою без прямих залежностей.

Наприклад, модуль кошика може публікувати подію, що товар було додано до кошика, на яку підписується модуль навігації (для оновлення лічильника товарів), тощо. При цьому модуль кошика залишається повністю незалежним і не знає про існування інших модулів.

Синхронна комунікація за допомогою контрактів

Синхронна комунікація на основі контрактів - це підхід в якому взаємодія між компонентами відбувається за допомогою явно визначених контрактів, які описують інтерфейси, типи даних та правила їх взаємодії. На відміну від неявної комунікації через події, де структура даних може бути невизначеною, або слабо типізованою, цей підхід забезпечує гарантії щодо форми даних, які передаються між компонентами.

Контракт - це щось, на кшталт, угоди між двома компонентами системи, а саме між постачальником функціональності та її споживачем. Постачальник зобов'язується надавати певний набір операцій з визначеними сигнатурами, а споживач в свою чергу використовує ці операції згідно з домовленим контрактом. Така модель схожа на концепцію інтерфейсів в об'єктно-орієнтованому програмуванні, але застосована на рівні незалежних модулів або сервісів.

Застосування в мікрофронтенд архітектурі

В контексті мікрофронтендів комунікація на основі контрактів вирішує проблему координації між незалежними UI модулями, які розробляються різними командами та навіть можуть використовувати різні технології. Коли

один мікрофронтенд потребує функціональності від іншого, то замість прямого доступу до внутрішньої реалізації, ці модулі взаємодіють через публічний контракт. Це створює чітку межу між публічним API модуля та його внутрішньою реалізацією.

Інтеграція з Ін’єкцією залежності

Комунікація на основі контрактів гарно поєднується з патерном Ін'єкції залежності формуючи архітектуру, в якій модулі не створюють свої залежності самостійно, а отримують їх ззовні через контракти. Споживач декларує залежність від абстрактного контракту, а не від конкретної реалізації, що забезпечує інверсію контролю. Таким чином модулі залежать від абстракцій, а не від конкретних реалізації. На Рисунку 5 зображене схематичне представлення цього патерну.

В розподілених середовищах мікрофронтендів, Ін'єкція залежності, зазвичай працює не на рівні окремих класів чи функцій, а на рівні цілих модулів. Модуль реєструє себе як постачальник певних контрактів при ініціалізації, а інші модулі можуть запитувати ці контракти через глобальний реєстр, не маючи прямих імпортів або залежностей від постачальника.

Основною перевагою використання комунікації за допомогою контрактів у поєднанні з ін’єкцією залежності у мікрофронтенд архітектурі є типобезпека. Використання систем статичної типізації дозволяє виявляти невідповідності між визначенням контракту та його використанням на етапі компіляції, а не під час виконання програми. Компілятор автоматично виявляє всі місця, де використовується контракт, і повідомить про невідповідності типів. Іншою перевагою є можливість зміни реалізацій без зміни в коді споживачів. Також значно покращуються можливості для unit тестування

системи. Адже щоб потестувати певний модуль в ізоляції, достатньо надати пусту імплементацію залежностей в модуль, що тестується.

Також сучасні великі продукти часто потребують використання АБ тестування, і використання такої архітектури напряму сприяє цьому, адже можна для одної групи користувачів надати одну імплементацію модуля, а для іншої іншу без змін у споживачах цих модулів.

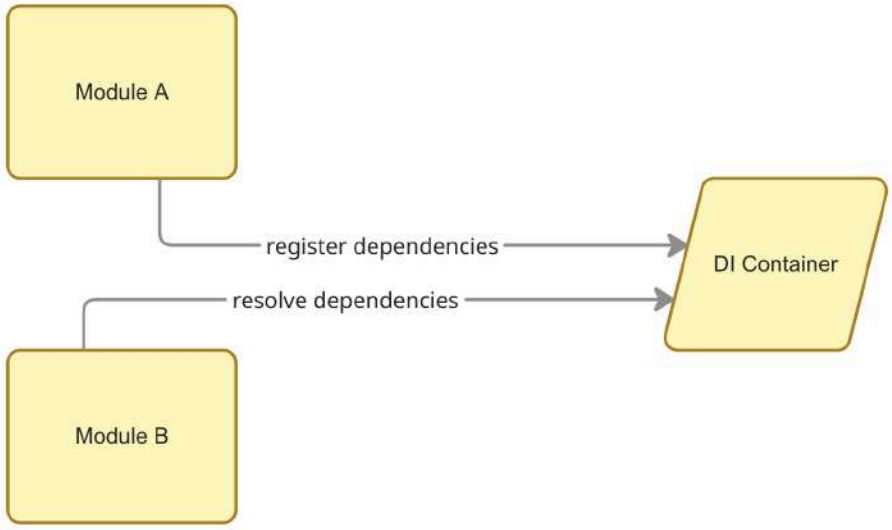

Рисунок 5. Схематичне зображення патерну Ін’єкція залежностей

Комбінування з Подійно-орієнтованою архітектурою

Комунікація на основі контрактів та подійно-орієнтована комунікація не є взаємовиключними. Вони вирішують різні типи задач і часто використовуються разом в класичній мікросервісній архітектурі.

В типовій мікрофронтенд системі модуль може використовувати контракти для отримання даних або виклику бізнес-логіки інших модулів, і одночасно публікувати події про зміни свого стану. Контракти надають

типобезпеку та явні залежності для критичних взаємодій, а події забезпечують низьку зв’язаність між модулями.

Розділ 2. Аналіз існуючих рішень та формування вимог

Огляд існуючих мікрофронтенд-фреймворків

Single-SPA

Single-SPA є одним з перших та найбільш популярних фреймворків для мікрофронтендів. Single-SPA працює як meta-framework, який керує множиною застосунків (мікрофронтендів). Кожен мікрофронтенд реєструється з lifecycle функціями (bootstrap, mount, unmount, update) та activity function, яка визначає, коли застосунок має бути активним. Single-SPA відповідає за виклик відповідних lifecycle hooks на основі URL або інших умов.[5]

Single-SPA підтримує React, Angular, Vue, Svelte та інші фреймворки через екосистему адаптерів, які обгортають фреймворк-специфічну логіку в єдиний життєвий цикл.

Основні можливості:

1. Активація мікрофронтендів в залежності від поточної адреси
2. Працює з різними фреймворками та бібліотеками
3. Підтримка “лінивого” підвантаження модулів

Обмеження:

Single-SPA не надає вбудованих механізмів для комунікації між мікрофронтендами. Розробники мають самостійно реалізовувати способи комунікації, контролем за внутрішнім станом і тд.

Piral

Piral - це фреймворк для портальних застосунків наступного покоління. Він дозволяє створювати веб-застосунки, що відповідають архітектурі мікрофронтендів, практично в найкоротші терміни. Piral дозволяє використовувати стандартні інструменти для максимальної ефективності. [6]
Основні можливості:

1. Архітектура плагінів з ін’єкцією залежностей.
2. Централізований сервіс для управління мікрофронтендами.
3. Вбудована комунікація через спільний стан.
4. Підтримка TypeScript

Обмеження: Piral нав'язує специфічну структуру застосунку та патерни, які можуть не підходити для всіх випадків. Залежність від центрального сервісу створює централізовану точку відмови та додатковий інфраструктурний компонент. Обмежена підтримка Module Federation. Piral має власну систему завантаження модулів, яка не повністю використовує можливості Webpack Module Federation. Piral також вимагає значних архітектурних змін для інтеграції.

Порівняльний аналіз обмежень

Аналіз існуючих рішень виявляє декілька архітектурних проблем. Наприклад, відсутність готових патернів комунікації між модулями. Piral надає механізми комунікації, за допомогою спільного стану, але спільний

стан суперечить одному з основних принципів мікросервісної архітектури. Відсутність типобезпеки між модулями. Навіть при використанні TypeScript всередині окремих мікрофронтендів, взаємодія між ними залишається нетипізованою.

Більшість існуючих рішень не надають архітектурних патернів для реалізації всіх принципів мікросервісної архітектури на фронтенді, зокрема принцип слабкої зв'язаності та децентралізованого управління станом. Обмежена підтримка гібридного способу розподілу мікрофронтендів. Існуючі фреймворки орієнтовані або на вертикальний поділ або на горизонтальний, але не надають зручних механізмів для комбінування обох підходів в одній системі для використання переваг обидвох підходів.

Формування вимог до фреймворку

На основі виявлених обмежень існуючих рішень було сформовано набір вимог для нашого мікрофронтенд фреймворку.

Функціональні вимоги

1. Підтримка різних UI фреймворків. Наш фреймворк має забезпечувати можливість одночасного використання React, Angular, Vue, Svelte та інших UI фреймворків в рамках одного застосунку.
2. Готові патерни комунікації. Фреймворк має надавати вбудовані механізми для комунікації.
3. Підтримка вертикального та горизонтального розбиття. Вертикальний поділ та активація мікрофронтендів при певній умові (наприклад зміна URL). Горизонтальний поділ через паралельне

завантаження множини мікрофронтендів на одній сторінці. Та гібридний підхід, що комбінує обидві стратегії.

4. Абстракція над Module Federation. Фреймворк має використовувати всі переваги Module Federation та надавати абстракцію над ним. Щоб забезпечити легку та централізовану конфігурацію, автоматичну генерацію типів для мікрофронтендів, тощо.
5. Керування життєвим циклом мікрофронтендів. Фреймворк має забезпечувати коректне керування життєвим циклом мікрофронтендів, та вирішувати проблему динамічних залежностей між мікрофронтендами.
6. Фреймворк має надавати механізм для композиції різних мікрофронтендів.

Нефункціональні вимоги

1. Типобезпека. Вся комунікація між мікрофронтендами має бути строго типізованою через TypeScript.
2. Незалежність та слабка зв'язаність. Мікрофронтенди не мають мати прямих залежностей один від одного. Взаємодія відбувається виключно через абстракції (події, контракти, слоти), щоб забезпечити незалежну розробки, тестування та розгортання.
3. Гнучкість у виборі технологій.
4. Проста конфігурація. Створення нового мікрофронтенду має вимагати мінімальної конфігурації.
5. Покращений досвід розробки. Фреймворк має надавати інструменти для розробників, для спрощення відлагодження програми, візуалізації

стану програми, моніторингу та інспектування залежностей, тощо. А також інструменти для зручного тестування модулів.

6. Фреймворк має відповідати всім основним принципам мікросервісної архітектури:
    - Незалежність розгортання.
    - Слабка зв'язаність.
    - Висока згуртованість
    - Організація навколо бізнес потреб.
    - Децентралізоване керування станом.
    - Ізоляція відмов.

Ці вимоги формують основу для проєктування архітектури фреймворку, яка буде описана в наступному розділі.

Розділ 3. Мікрофронтенд-фреймворк для побудови масштабованих вебзастосунків

Високорівнева архітектура системи

Архітектура фреймворку побудована таким чином, щоб сприяти максимальній гнучкості при виборі фронтенд технологій - фреймворк не диктує, як саме мікрофронтенди мають рендерити свій інтерфейс, а лише координує їх життєвий цикл та забезпечує механізми комунікації між ними. Це досягається через чітке розділення відповідальностей між компонентами системи.

Компоненти високого рівня

Архітектура складається з трьох основних компонентів, кожен з яких має чітко визначену відповідальність:

1. AppContext є центральним компонентом, який надає єдиний API для всіх форм комунікації між мікрофронтендами. Він інкапсулює в собі шину подій для асинхронної комунікації та реєстр сервісів для синхронної взаємодії через строго типізовані контракти.
2. Bootstrap Manager відповідає за оркестрацію життєвого циклу мікрофронтендів. Він аналізує залежності між модулями, визначає порядок їх завантаження за допомогою топологічного сортування, керує пріоритетністю завантаження та виконує активацію/деактивацію модулів.
3. Router забезпечує інтеграцію з браузерним History API та відповідає за визначення, які мікрофронтенди мають бути активними для

конкретного URL. Router лише повідомляє Bootstrap Manager про необхідність активації чи деактивації певних мікрофронтендів.

На Рисунку 6 зображено високорівневу архітектура фреймворку та тестового застосунку, який його використовує. Host App - це частина застосунку, яка відповідає за імпорт збірок інших мікрофронтендів та фреймворку і робить виклик Bootstrap Manager, який оркеструє мікрофронтенди.

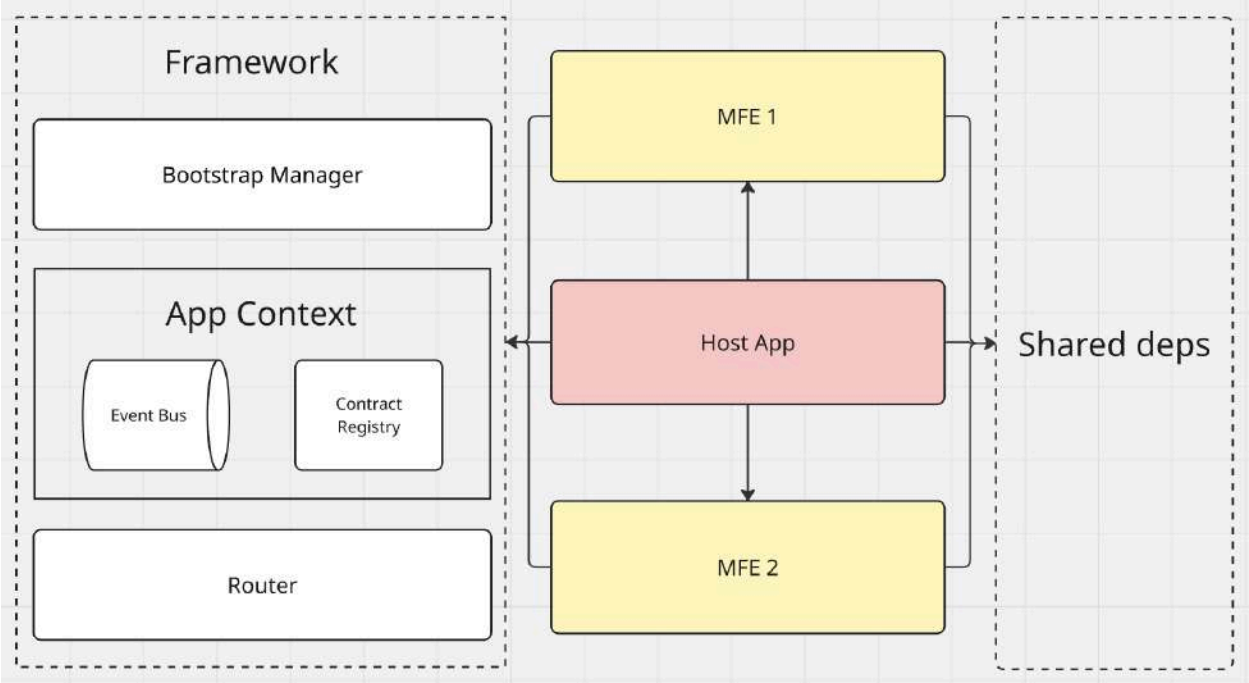

Рисунок 6. Високорівнева архітектура системи

Framework-Agnostic підхід

Ключовою архітектурною особливістю є повна агностичність щодо фронтенд фреймворків. На відміну від традиційних підходів, де фреймворк надає адаптери для React, Vue, Svelte, цей фреймворк взагалі не займається

рендерингом. Кожен мікрофронтенд самостійно обирає свій UI фреймворк та використовує його нативні API для рендерингу.

Повна відповідальність за підключення та відключення компонентів покладається на самі мікрофронтенди: при ініціалізації вони реєструють свої можливості та відображають інтерфейс, а при знищенні вони зобов'язані самостійно звільнити всі ресурси та скасувати підписки, оскільки фреймворк не має доступу до деталей конкретних технологій і не може зробити це автоматично.

Мікрофронтенди також можуть декларувати та ділитись певними слотами DOM контейнерів за допомогою API. Інші мікрофронтенди можуть використати ці слоти для рендерингу. Наприклад, React MFE отримує DOM container від певного API та самостійно займається свої рендерингом. Приклад зображений на Рисонку 7.

```
1  const container = context.getAPI(SomeUiApi).rootContainer
2  const root = ReactDOM.createRoot(container);
3  root.render(<MyComponent />);
```

Рисунок 7. Приклад відмальовки мікрофронтенду за допомого бібліотеки React

Рішення не надавати адаптери для конкретних технологій базується на принципі мінімальної абстракції. Відсутність додаткового шару абстракції надає розробникам повний доступ до можливостей обраної технології без жодних обмежень. Єдиний компроміс - це більше шаблонного коду в кожному мікрофронтенді.

Принцип інверсії залежностей

Архітектура реалізує принцип інверсії залежностей на рівні модулів. Мікрофронтенди не залежать від конкретних реалізацій інших модулів, натомість вони залежать від абстракцій (контрактів). AppContext виступає, як сервіс локатор. який забезпечує зв'язок між модулями, що надають залежності та тих що їх потребують. Коли MFE A потребує функціональності від MFE B, він не імпортує B безпосередньо. Замість цього:

1. MFE B реєструє свою реалізацію контракту в AppContext
2. MFE A запитує контракт з AppContext
3. AppContext надає реалізацію без розкриття деталей про MFE B

Така модель, зображена на Рисунку 8, забезпечує можливість заміни реалізацій без зміни коду споживачів, що є фундаментальним принципом для підтримки незалежності розгортання.

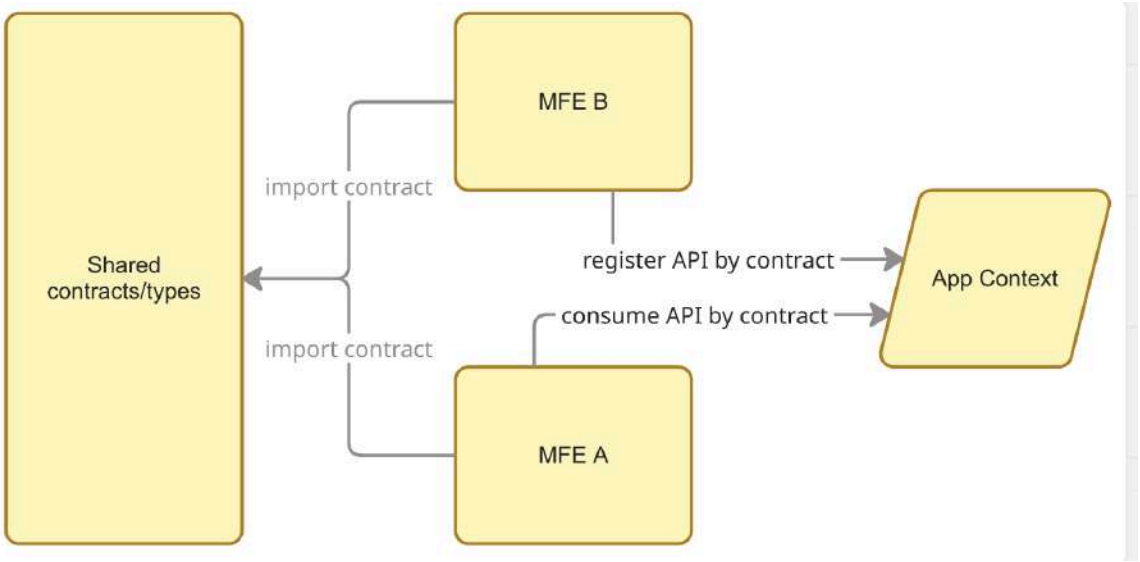

Рисунок 8. Інверсія залежностей на рівні модулів

Архітектура комунікаційного шару

Комунікаційний шар є серцем архітектури, який забезпечує взаємодію між модулями без створення прямих залежностей.

Архітектура шини подій

Event Bus реалізує класичний Pub/Sub патерн для асинхронної комунікації між мікрофронтендами. Архітектура базується на повній незалежності видавців від підписників. Видавці генерують події при виникненні певних змін в своєму домені. Публікація події є fire-and-forget операцією, тобто видавець не очікує підтвердження та навіть не знає, хто обробить цю подію. Підписник реєструє функції-обробники для подій певного типу. Один підписник може бути підписаний на множину типів подій, і одна подія може мати множину підписників. А Event Bus виступає як посередник в цій архітектурі, він не зберігає історію подій, а лише викликає відповідні обробники. Фреймворк також надає клас фабрику для створення строго типізованих подій, який генерує унікальний ідентифікатор та тип події.

Архітектура реєстру контрактів

Contract Registry - це централізований реєстр контрактів, який реалізує Сервіс Локатор взірець для синхронної комунікації через типізовані API. Контракт визначає інтерфейс без реалізації. Він є абстрактною угодою між мікрофронтендами про форму API. Фреймворк також надає клас фабрику для контрактів, який генерує унікальний ідентифікатор для контракту та зберігає його тип. Мікрофронтенд декларує свої залежності, а Bootstrap Manager гарантує, що всі необхідні контракти будуть зареєстровані до моменту виклику bootstrap методу цього мікрофронтенду.

На відміну від багатьох архітектур, які використовують JSON Schema для валідації під час виконання програми, цей фреймворк повністю покладається на типізацію TypeScript на етапі компіляції. Це рішення базується на припущенні, що всі мікрофронтенди написані на TypeScript та проходять через компіляцію. Основна перевага такого підходу - він допомагає виявляти помилки ще на етапі компіляції збірки а не під час виконання програми.

Реєстр контрактів спроєктований централізованим, адже єдина точка проходження всіх взаємодій спрощує відладку програми, дозволяє створити інструменти розробника та дозволяє легко підміняти весь контекст під час тестування і забезпечити унікальність кожного зареєстрованого контракту. Теоретично така архітектура створює єдину точку відмови, проте на практиці ризик мінімальний, оскільки реєстр є простим об'єктом у пам'яті без складної логіки.

Підтримка різних стратегій поділу застосунку

Фреймворк підтримує як вертикальний, так і горизонтальний поділ мікрофронтендів, а також їх комбінацію.

Вертикальний поділ організовує мікрофронтенди за маршрутами, де кожен маршрут активує відповідний мікрофронтенд. Для цього фреймворк надає свою імплементацію роутера. При зміні маршруту роутер визначає:

- які мікрофронтенди мають бути активовані;
- які мікрофронтенди мають бути деактивовані;

Горизонтальний поділ дозволяє множині мікрофронтендів працювати паралельно на одній сторінці, кожен відображає свою частину UI.

Горизонтальний поділ базується на простому принципі: один MFE створює DOM container та надає його іншому MFE через API контракт.

Реальні застосунки часто комбінують обидві стратегії: базові модулі завжди активні та надають інфрастуктуру для інших модулів. А доменно специфічні модулі активуються на відповідних маршрутах та відображаються в слотах, наданих базовими модулями. Фреймворк також дозволяє деяким модулям бути умовно активованими базуючись не лише на маршруті, але й на інших критеріях, які задекларує сам модуль.

Система оркестрації модулів

Bootstrap Manager є центральним компонентом оркестрації, який координує життєвий цикл усіх мікрофронтендів в системі. Bootstrap Manager керує трьома ключовими аспектами життєвого циклу: завантаженням модулів, розв'язанням залежностей та активацією на основі шляхів. Життєвий цикл мікрофронтендів складається з наступних станів (див Рисунок 9):

1. NOT_LOADED: мікрофронтенд зареєстровано, але його збірка ще не завантажена.
2. LOADING: Збірка завантажується з віддаленого сервера.
3. LOADED: Збірку завантажено, але bootstrap hook ще не викликано.
4. BOOTSTRAPPED: Bootstrap hook виконано, мікрофронтенд активний та може взаємодіяти з іншими мікрофронтендами за допомогою AppContext.
5. DESTROYED: Destroy hook виконано, мікрофронтенд деактивований (можливий повторний перехід до LOADED при re-activation).

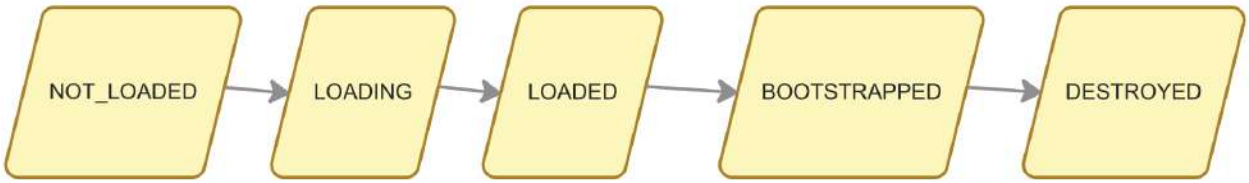

Рисунок 9. Життєвий цикл мікрофронтендів

Bootstrap Manager зберігає усі завантажені модулі та їх поточний стан. При зміні шляху за допомогою роутера, він визначає які модулі потребують переходу в інший стан та оркеструє ці переходи в правильному порядку.

Розв'язання залежностей за допомогою топологічного сортування

Для розв'язання залежностей Bootstrap Manager використовує топологічне сортування. Кожен мікрофронтенд декларує: список контрактів, які він надає та список контрактів, які він споживає. На основі цього будується направлений граф залежностей, в якому вершини це мікрофронтенди, а ребро між між вершиною A і B існує, якщо A споживає контракт, який B надає. Алгоритм топологічного сортування визначає лінійний порядок вершин графа, що для кожного ребра (від A до B), вершина B йде раніше A в порядку. Він працює наступним чином:

1. Знайти всі вершини, які надають імплементацію контрактів при цьому не мають залежностей.
2. Зберегти їх та видалити з графу.
3. Повторити процес поки граф не стане порожнім.

Якщо граф не стає порожнім, то це означає що залежності створюють циклічну залежність і Bootstrap Manager повідомить про цю помилку з описом циклу.

Пріоритетність мікрофронтендів

Архітектура фреймворку дозволяє визначати пріоритет конкретних мікрофронтендів. Високопріоритетні мікрофронтенди позначаються за допомогою `priority: 'high'` в MFE export і вони не можуть мати ніяких залежностей. Ці модулі Bootstrap Manager опрацьовує першими. Прикладом такого модулю може бути мікрофронтенд з базовою структуру користувацького інтерфейсу, який необхідно показати якнайшвидше, навіть якщо контент ще завантажується.

Абстракція над Module Federation за допомогою Webpack Plugin

Налаштування Module Federation є доволі складним та створює декілька проблем:

1. Кожен мікрофронтенд має власний файл налаштувань, де потрібно вручну вказувати залежності та точки підключення. Ця конфігурація дублюється між різними середовищами розробки, та різними мікрофронтендами.
2. Віддалені модулі не мають автоматичної типізації. Розробник змушений вручну створювати файли оголошень типів для кожного віддаленого модуля, це може призводити до застарілих типів при змінах і як результат - помилок.
3. Кожен новий мікрофронтенд потребує майже ідентичного налаштування з незначними відмінностями.

Фреймворк пропонує додаткову абстракцію над Module Federation. Це Webpack Plugin, який отримує єдиний файл конфігурації де замість розгорнутих налаштувань розробник декларативно описує назву модуля, його точки входу та спільні залежності з іншими мікрофронтендами. Плагін самостійно генерує повну конфігурацію на основі цього опису. Також конфігурація може варіюватись в залежності від конкретних середовищ виконання. Плагін автоматично підставляє правильну адресу залежно від поточного середовища. Плагін перевіряє конфігурацію ще на етапі збірки та підтверджує існування всіх залежностей, коректність шляхів та діапазонів версій спільних бібліотек.

Під час компіляції плагін зчитує файл конфігурації, формує необхідні параметри, створює екземпляр плагіна та додає його до налаштувань Webpack, а також генерує файли оголошень типів для вхідних точок мікрофронтендів, щоб забезпечити безпеку типів при імпорті мікрофронтендів у головний застосунок.

Інструменти розробника

Для покращення користувацького досвіду у фреймворці також були передбачені інструменти розробника, які надають можливість спостерігати за роботою програми в реальному часі за допомогою спеціальної панелі, яка викликається комбінацією клавіш. Вона показує список усіх подій у системі і дозволяє фільтрувати їх за типом, часом та вмістом, а також переглядати передані дані. Також можна відстежувати реєстрацію та використання контрактів. На основі цих даних будується візуальне представлення графа залежностей між мікрофронтендами. Панель розробника також візуалізує життєвий цикл кожного мікрофронтенда: його поточний стан, час

завантаження та ініціалізації, та його залежності. Це повинно дозволити користувачам фреймворку виявляти надто великі модулі чи повільну логіку ініціалізації, що блокує решту системи, та спростити процес відладки програми.

Тестування мікрофронтендів зазвичай відбувається за принципом повної ізоляції, тобто кожен модуль тестується незалежно від решти системи. Для цього фреймворком передбачено інструменти для тестування, такі як тестова версія App Context, яка імітує роботу справжнього середовища виконання та перехоплює публікацію подій, реєстрацію контрактів та звернення до залежностей, зберігаючи всі виклики в пам'яті замість реального їх виконання. Тестова версія контексту не має жодних зовнішніх залежностей, тому тести виконуються швидко та без побічних ефектів.

Розділ 4. Реалізація фреймворку

Система контрактів: Contract Factory та Event Factory

Contract Factory

Contract Factory - відповідальна за створення унікальних ідентифікаторів для API контрактів, забезпечуючи при цьому строгу типізацію API для споживачів за допомогою TypeScript. Приклад реалізація зображений на лістингу 1.

```
export class ContractFactory {
  static create<T>(id: string): Contract<T> {
    return { id } as Contract<T>;
  }
}
export interface Contract<T> {
  id: string;
  _apiType?: T;
}
```

Лістинг 1. Contract Factory

Ключовим тут є поле _apiType, яке позначене як optional та ніколи не заповнюється значенням. Це поле існує лише для того, щоб TypeScript міг отримати тип контракту. Type assertion as Contract<T> дозволяє повернути об'єкт з лише полем id, але TypeScript вважає його типом Contract<T>, тобто надає інформацію про тип API. Приклад використання наведений на лістингу 2.

```
interface DocumentAPI {
  getDocument(): Document;
  updateBlock(id: string, updates: DocumentUpdates): void;
}
const DocumentContract = ContractFactory.create<DocumentAPI>('document-api');

// Provider
context.register(DocumentContract, implementation);

// Consumer
const doc = context.use(DocumentContract);
doc.getDocument(); // ✓ Autocomplete працює
doc.invalidMethod(); // ✗ Compile error
```

Лістинг 2. Приклад використання Contract Factory

Event Factory

Event Factory використовує схожий підхід для створення строго типізованих подій. Єдиною відмінністю є те, що замість унікального ідентифікатора тут використовується поле “тип”, адже в системі може існувати безліч подій одного типу. Приклад реалізації Event Factor наведений в лістингу 3.

```
export class EventFactory {
  static create<T>(type: string): EventContract<T> {
    return { type } as EventContract<T>;
  }
}

export interface EventContract<T> {
  type: string;
  _payloadType?: T;
}
```

Лістинг 3. Event Factory

Внутрішній маршутизатор (Роутер)

Маршрутизатор забезпечує вертикальний поділ мікрофронтендів використовуючи інтеграцію з браузерним History API та простий механізм порівняння URL. Маршрутизатор слухає два типи навагаційних подій, які відправляє браузер:

1. Popstate event спрацьовує при використанні браузерної навігації за допомогою кнопок вперед/назад. Маршутизатор реагує на браузерну навігацію і активовує відповідні мікрофронтенди.
2. Click event. Маршутизатор перехоплює всі кліки по тегам “a” на сторінці. Якщо посилання є внутрішнім (same origin), то маршутизатор запобігає поведінці браузера за замовчуванням та використовує History API для навігації між мікрофронтендами. Що в свою чергу дозволяє навігувати між мікрофронтендами без перезавантаження сторінки.

Приклад імплементації підписки на ці події наведений на лістингу 4.

```
window.addEventListener('popstate', () => {
    this.handlePathChange(window.location.pathname);
});
window.addEventListener('click', (e) => {
    const target = e.target as HTMLElement;
    const anchor = target.closest('a');
    if (anchor && anchor.href && anchor.origin === window.location.origin) {
        const path = new URL(anchor.href).pathname;
        if (path !== this.currentPath) {
            e.preventDefault();
            this.navigate(path);
        }
    }
});
```

Лістинг 4. Імплементація підписки на події Роутером

Реєстр API: Взірець Типобезпечного Локатора Сервісів

Реєстр API є центральним реєстром для всіх синхронних API між мікрофронтендами. Його реалізація базується на взірці Локатор Сервісів. Основна відповідальність компонента полягає у зберіганні контрактів та їх імплементацій та забезпеченні типобезпеки під час взаємодії. Центральною структурою реєстру даних є асоціативний масив Map, який зберігає відповідності між унікальними ідентифікаторами контрактів та об'єктами реєстрації. Кожна реєстрація містить саму імплементацію API та дані про час реєстрації. Спрощена версія імплементації реєстру API наведена на лістингу 5 і складається з двох основних методів: register та use.

```
export class APIRegistry {
  private apis: Map<string, APIRegistration>;
  constructor() {
    this.apis = new Map();
  }
  register<T>(contract: Contract<T>, implementation: T): void {
    if (this.apis.has(contract.id)) {
      throw new ApiRegistredError();
    }
    const registration: APIRegistration<T> = {
      id: contract.id,
      implementation,
      registeredAt: Date.now(),
    };
    this.apis.set(contract.id, registration);
  }
  use<T>(contract: Contract<T>): T {
    const registration = this.apis.get(contract.id);
    if (!registration) {
      throw new ApiNotFoundError();
    }
    return registration.implementation as T;
  }
}
```

Лістинг 5. Імплементація реєстру API

Шина подій

Для імплементації шини подій, було використано популярну бібліотеку для роботи з потоками подій rxjs. Вибір rxjs обумовлений його великою кількістю операторів для роботи з асинхронними потоками даних, а також rxjs уже є стандартом в екосистемі Angular та також часто використовується у React додатках для зберігання стану застосунку. Основним недоліком ж є розмір цієї бібліотеки, що може суттєво вплинути на розмір збірки клієнтських застосунків, та цю проблему вирішує внутрішня оптимізація збірника, яка називається tree shaking, вона прибирає з загальної збірки той код бібліотеки, який напряму не використовується. Лістинг 6 демонструє імплементацію шини подій.

```
export class EventBus {
  private eventStream$: Subject<InternalEvent>;
  constructor() {
    this.eventStream$ = new Subject<InternalEvent>();
  }
  publish<T>(eventContract: EventContract<T>, payload: T): void {
    const event: InternalEvent<T> = {
      type: eventContract.type,
      payload,
    };
    this.eventStream$.next(event);
  }
  subscribe<T>(
    eventContract: EventContract<T>,
    handler: (payload: T) => void
  ): EventSubscription {
    const subscription: Subscription = this.eventStream$
      .pipe(
        filter((event) => event.type === eventContract.type),
        map((event) => event.payload as T)
      )
      .subscribe(handler);
    return {
      unsubscribe: () => subscription.unsubscribe(),
    };
  }
}
```

Лістинг 6. Імплементація шини подій

Контекст Застосунку(AppContext)

Контекст Застосунку є фасадом, який інкапсулює реєстр API та шину подій у єдиний інтерфейс, який будуть використовувати мікрофронтенди для комунікації. Компонент використовує класичний шаблон делегування для розподілу відповідальності між внутрішніми компонентами. А також надсилає системні події про роботу системи, це потрібно для імплементації інструментів розробника. Спрощений приклад імплементації глобального контексту наведений на лістингу 7.

```
export class AppContext implements GlobalContext {
    private eventBus: EventBus;
    private apiRegistry: APIRegistry;
    constructor() {
        this.eventBus = new EventBus();
        this.apiRegistry = new APIRegistry();
    }
    publish<T>(eventContract: EventContract<T>, payload: T): void {
        this.eventBus.publish(eventContract, payload);
    }
    subscribe<T>(eventContract: EventContract<T>,
                 handler: (payload: T) => void): EventSubscription {
        return this.eventBus.subscribe(eventContract, handler);
    }
    register<T>(contract: Contract<T>, implementation: T): void {
        this.apiRegistry.register(contract, implementation);
        this.eventBus.publish(ApiRegisteredEvent, { id: contract.id });
    }

    use<T>(contract: Contract<T>): T {
        const api = this.apiRegistry.use(contract);
        this.eventBus.publish(ApiUsedEvent, { id: contract.id });
        return api;
    }
}
```

Лістинг 7. Імплементація глобального контексту

Менеджер Завантаженнь

Менеджер завантаження координує мікрофронтенди у дві фази: спочатку всі модулі завантажуються паралельно для економії часу, потім ініціалізуються строго послідовно з урахуванням залежностей: спершу інфраструктурні модулі,які позначаються користувачами фреймворку, як модулі з високим пріоритетом, потім бізнес-логіка у порядку топологічного сортування, що наведено на лістингу 8. При зміні маршруту спочатку активуються нові модулі, і лише потім деактивуються старі, це зроблено, щоб запобігати помилкам зниклих залежностей.

При будь-якій помилці ініціалізації система обирає стратегію швидкої зупинки замість м'якої деградації. Це дозволяє одразу побачити справжню причину проблеми, не допускаючи каскаду вторинних помилок, які приховують першопричину.

```
export const topologicalSort = (mfes: LoadedMFE[], highPriorityMfes: LoadedMFE[] = []): LoadedMFE[] => {
    const sorted: LoadedMFE[] = [];
    const visited = new Set<string>();
    const visiting = new Set<string>();
    const providerMap = new Map<string, LoadedMFE>();
    for (const loaded of highPriorityMfes) {
        for (const contract of loaded.mfe.injects) {
            providerMap.set(contract.id, loaded);
        }
    }
    for (const loaded of mfes) {
        for (const contract of loaded.mfe.injects) {
            providerMap.set(contract.id, loaded);
        }
    }
    for (const loaded of highPriorityMfes) visited.add(loaded.name);
    const visit = (loaded: LoadedMFE) => {
        if (visited.has(loaded.name)) {
            return;
        }
        if (visiting.has(loaded.name)) {
            throw new CircularDependencyError(loaded.name);
        }
        visiting.add(loaded.name);
        for (const requiredContract of loaded.mfe.uses) {
            const provider = providerMap.get(requiredContract.id);
            visit(provider);
        }
        visiting.delete(loaded.name);
        visited.add(loaded.name);
        sorted.push(loaded);
    };
    for (const loaded of mfes) visit(loaded);
    return sorted;
}
```

Лістинг 8. Імплементація топологічного сортування з урахуванням пріоритетності

Плагін для спрощення конфігурації розділення модулів

Також було реалізовано webpack-плагін, який вирішує відразу три задачі: централізує конфігурацію потрібну для Module Federation в одному місці, замість дублювання її в кожному мікрофронтенді. Підтримує різні

середовища розробки, та автоматично генерує TypeScript декларації, які може використовувати хост-застосунок. Приклад такого файлу наведений на лістингу 9.

```
/**
 * Auto-generated Module Federation type definitions
 * Generated at: 2025-12-27T21:46:46.254Z
 * DO NOT EDIT THIS FILE MANUALLY
 * Run your webpack build to regenerate this file
 */
declare module 'storage-mfe/main' {
  import { MFE } from 'femodular';
  const mfe: MFE | MFE[];
  export default mfe;
  export { mfe };
}
declare module 'toolbar-mfe/main' {
  import { MFE } from 'femodular';
  const mfe: MFE | MFE[];
  export default mfe;
  export { mfe };
}
declare module 'editor-mfe/main' {
  import { MFE } from 'femodular';
  const mfe: MFE | MFE[];
  export default mfe;
  export { mfe };
}
declare module 'hierarchy-mfe/main' {
  import { MFE } from 'femodular';
  const mfe: MFE | MFE[];
  export default mfe;
  export { mfe };
}
```

Лістинг 9. Приклад авто згенерованого файлу з TypeScript деклараціями віддалених модулів

Інструменти розробника для налагодження та тестування

Окрім основних компонентів фреймворка, також було імплементовано панель інструментів розробника, яка є інтерактивним графічним інтерфейсом для візуального відстеження та налагодження складної взаємодії між мікрофронтендами у режимі реального часу. Панель організована як інтерфейс з чотирма спеціалізованими вкладками. Перша вкладка “огляд” відображає загальний стан всіх мікрофронтендів з позначками про їхній поточний статус завантаження та ініціалізації. Друга вкладка містить хронологічний журнал всіх опублікованих подій з відміткою часу їх публікації. Третя вкладка відображає повний реєстр всіх зареєстрованих сервісів з інформацією про модуль, який є постачальником, а також список поточних модулів, які є споживачами. Четверта вкладка відповідає за візуалізацію залежностей між модулями у форматі текстового графа. Інформація в панелі автоматично оновлюється без необхідності ручного оновлення розробником завдяки підписці на всі системні події контексту застосунку. Панель інтегрується до будь-якого додатку та активується при натисканні комбінації клавіш Ctrl+Shift+D розробником. Дана панель зображена на Рисунку 10.

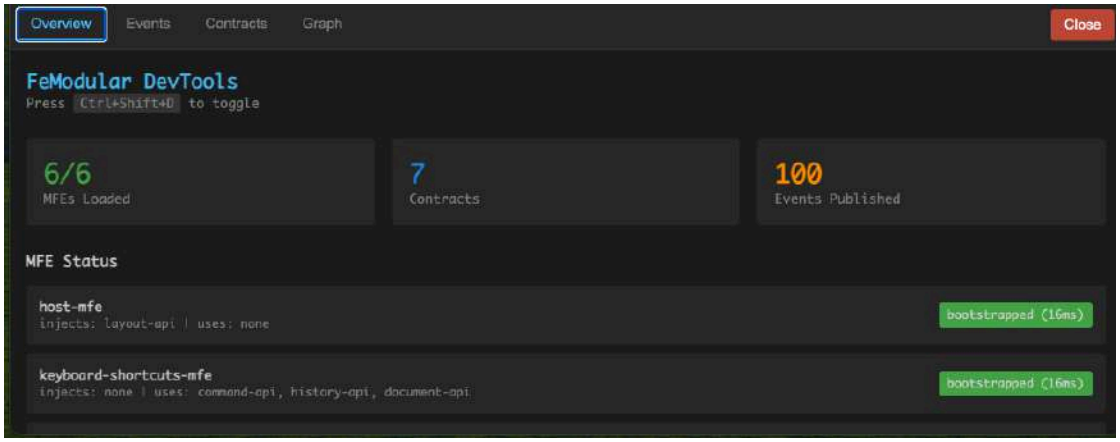

Рисунок 10. Панель інструментів розробника

Цей фреймворк був спроєктований для використання у великих системах, а такі системи майже неможливо побудувати без тестування. Тому для покращення досвіду написання тестів, було реалізовано інструменти для тестування фреймворку. Основним таким інструментом є клас макета контексту застосунку. Цей макет дозволяє тестувати окремі мікрофронтенд в ізоляції, без завантаження інших модулів.

Розділ 5. Демонстраційний застосунок: редактор документів

Цей розділ описує практичну реалізацію повноцінного вебзастосунку використовуючи розробленей фреймворк, та демонструє його можливості у реальному сценарії. Застосунок є спрощеним аналогом популярного сервісу GoogleDocs, тобто редактор текстових документів з підтримкою форматування, керування структурою документа та збереження змін.

Концепція та призначення застосунку

Основна ідея полягає у відтворенні базової функціональності популярних онлайн-редакторів документів, таких як GoogleDocs чи Coda. Весь інтерфейс користувача побудовано як набір незалежних мікрофронтендів, кожен з яких реалізовано з використанням іншого фреймворку користувацького інтерфейсу.

Застосунок надає користувачу середовище для роботи з текстовими документами, що складаються з окремих структурних блоків різних типів. Кожен блок може бути параграфом, заголовком одного з шести рівнів, елементом списку, блоком програмного коду з підсвіткою синтаксису. Користувач має можливість вільно створювати нові блоки, змінювати їхній тип, застосовувати різноманітне форматування до тексту всередині блоків, змінювати порядок блоків у документі, та виконувати інші операції редагування.

Горизонтальна панель інструментів у верхній частині екрану містить елементи форматування та дій над документом, основна центральна частина

відведена безпосередньо під редагований вміст документа, та вертикальна бічна панель зліва відображає ієрархічну структуру документа у вигляді дерева блоків. На рисунку 11 зображений цей демонстраційний застосунок, де червоним схематично зображено межі функціональних зон.

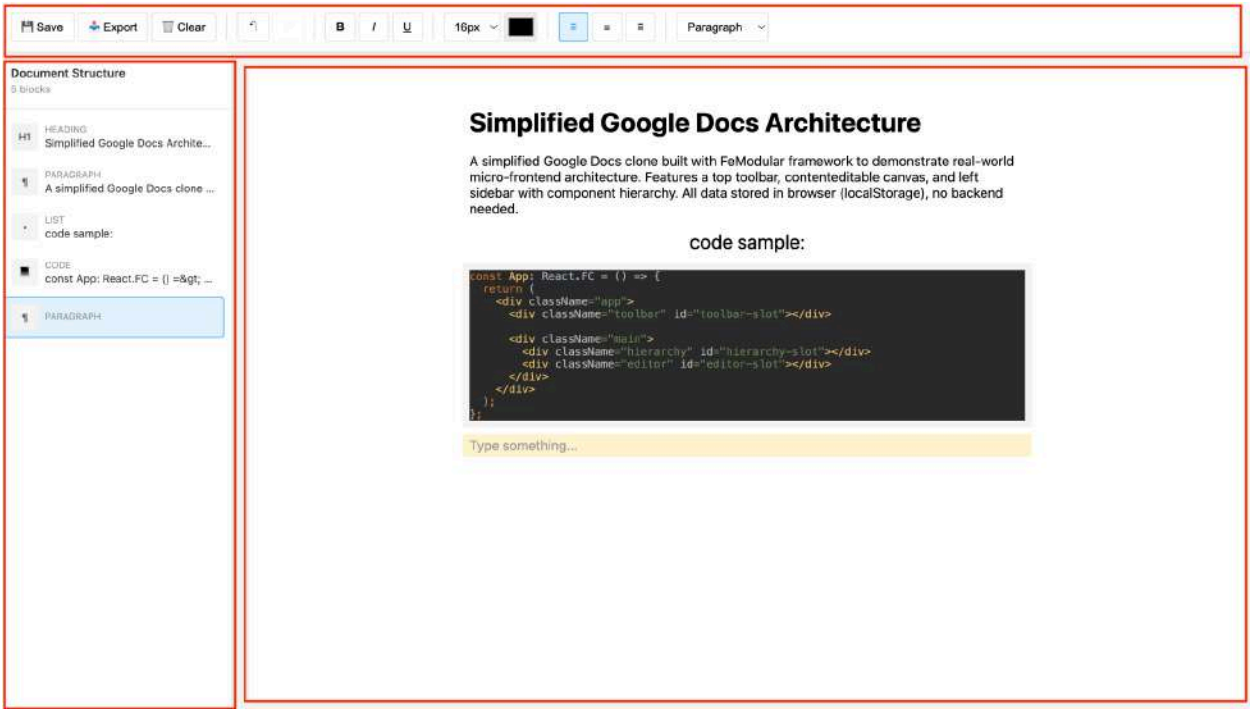

Рисунок 11.Демонстраційний застосунок

Архітектурний розподіл на мікрофронтенди

Застосунок розділено на п'ять окремих незалежних мікрофронтендів, кожен з яких має чітко визначену відповідальність. Такий розподіл відповідальностей дозволяє різним командам розробників працювати над різними частинами застосунку паралельно без конфліктів та взаємного блокування, при цьому використовуючи технологічні стеки, які їм найбільше підходять.

Демонстраційний застосунок свідомо використовує чотири різні підходи до реалізації користувацького інтерфейсу, щоб продемонструвати його можливості для інтеграцій з різними фреймворками для побудови інтерфейсу користувача. Так, наприклад, при реалізації різних мікрофронтендів було використано React, Svelte, Vue та нативний TypeScript.

Головний застосунок

Головний застосунок виконує роль контейнера для всіх інших мікрофронтендів. Його основна відповідальність полягає у створенні базової структури сторінки з трьома чітко визначеними областями розміщення модулів, ініціалізації фреймворку, та забезпеченні базових сервісів для інших модулів. Модуль реалізовано з використанням бібліотеки React.

Головний застосунок не містить складної бізнес-логіки та служить каркасом для розміщення інших модулів. Він створює три DOM-елементи з унікальними ідентифікаторами, які служать слотами для монтування користувацьких інтерфейсів інших мікрофронтендів. Ці слоти надаються іншим модулям через спеціальний контракт, що дозволяє кожному модулю отримати посилання на відповідний елемент та змонтувати у ньому свій інтерфейс з використанням власного фреймворку. Також головний застосунок обробляє глобальні комбінації клавіш клавіатури для швидкого доступу до часто використовуваних функцій, таких як збереження документа або скасування останньої дії.

Мікрофронтенд: Керування станом документа

Мікрофронтенд для керування станом редагованого документу є єдиним джерелом правди щодо вмісту та структури документу. На відміну від інших модулів, він не має візуального інтерфейсу та складається виключно з бізнес-логіки. Саме тому для його реалізації було обрано чистий TypeScript без використання жодного фреймворку користувацького інтерфейсу.

Модуль надає іншим модулям набір операцій для взаємодії з документом, зокрема створення нових блоків, видалення існуючих, оновлення вмісту та метаданих блоків, зміна порядку блоків у структурі документа.

Також додатковою функціональністю цього модуля є автоматичне збереження стану документа у сховище браузера кожні декілька секунд. При наступному завантаженні застосунку модуль автоматично відновлює збережений стан, дозволяючи користувачу продовжити роботу з того місця, де він зупинився. Також реалізовано функціональність експорту документа у файл JSON.

Мікрофронтенд: Панель інструментів

Мікрофронтенд панелі інструментів надає користувачу графічний інтерфейс для виконання операцій над документом. Він реалізований з використанням бібліотеки React. Панель інструментів містить декілька груп кнопок для форматування документу. Модуль не зберігає стан документу локально та не маніпулює ним безпосередньо. Натомість він отримує через реєстр контрактів сервіс керування станом та сервіс виконання команд і викликає їх.

Мікрофронтенд: Редактор вмісту документа

Мікрофронтенд редактора вмісту відповідає за взаємодію користувача з текстовим вмістом документа. Для його реалізації обрано фреймворк Vue.

Основна відповідальність модуля - це відображення структури документа. Модуль підписується на події зміни стану документу та перебудовує DOM дерево. Кожен блок документу відображається як окремий HTML елемент з атрибутом “contenteditable”, що дозволяє користувачу редагувати його вміст.

Мікрофронтенд також обробляє події клавіатури, так ,наприклад, натискання клавіші “Enter” всередині блоку створює новий порожній блок одразу після поточного та змінюю фокус клавіатури у цей новий блок, а натискання клавіші “Backspace” видаляє цей блок та переміщує фокус у попередній блок документа, тощо.

Мікрофронтенд: Ієрархія документу

Мікрофронтенд візуалізації ієрархії документу надає користувачам структурний вигляд документу у формі інтерактивного дерева блоків. Він розміщений у бічній панелі та реалізований з використанням фреймворку Svelte.

Основна функція цього мікрофронтенду полягає у навігації по документі. При натисканні на блок у ієрархії модуль викликає API редактора, який переміщує відповідний блок у редакторі у видиму зону та встановлює на ньому курсор.

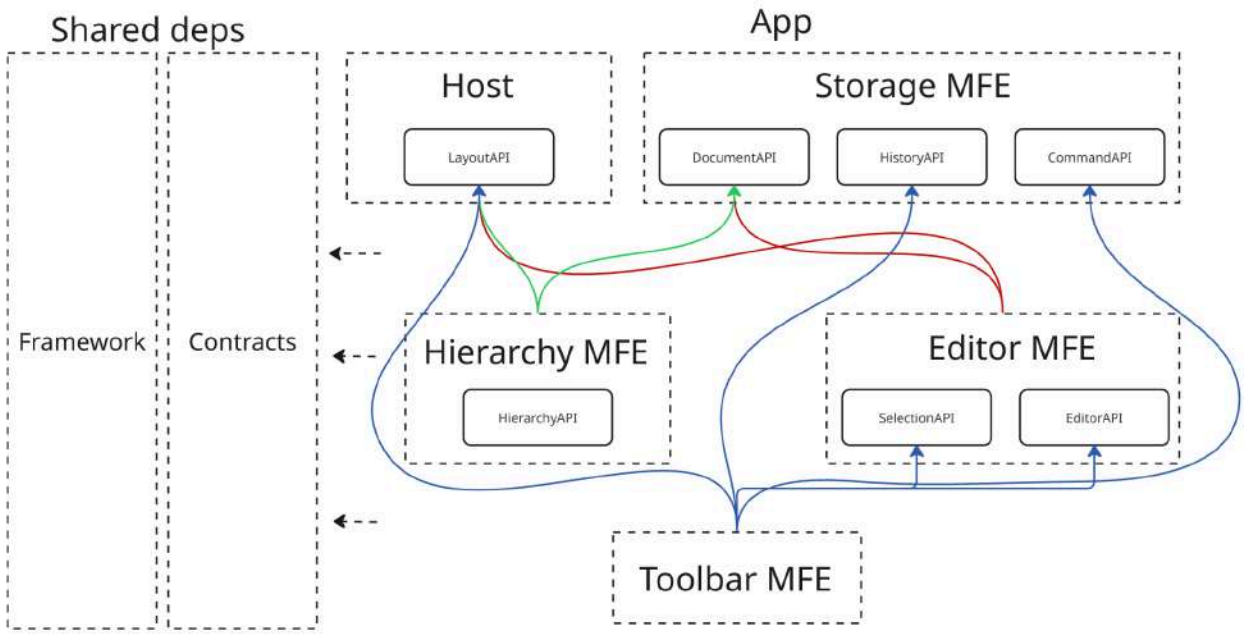

Рисунок 12.Архітектура демонстраційного застосунку

Демонстрація можливостей фреймворку

Важливою особливістю застосунку є те, що одночасно використовуються три різні фреймворки користувацького інтерфейсу у одному застосунку без адаптерів. Кожен мікрофронтенд використовує нативний API свого фреймворку для монтування компонентів у DOM. Модуль панелі інструментів викликає стандартний метод createRoot з React для рендерингу. Мікрофронтенд редактора викликає стандартний метод createApp з Vue для створення екземпляру застосунку. Модуль ієрархії використовує стандартний конструктор компонента Svelte. Фреймворк не диктує як саме має відбуватись процес рендерингу та не потребує спеціальних адаптерів. Така незалежність досягається завдяки важливому архітектурному рішенню, що фреймворк не відповідає за рендеринг користувацьких інтерфейсів, а лише надає DOM слоти через API та забезпечує механізми комунікації через контракти.

Система контрактів демонструє, як можна досягти повної типобезпеки при взаємодії між мікрофронтендами. При реєстрації API, мікрофронтенд просто викликає метод register з контрактом та імплементацією. Компілятор автоматично перевіряє, чи імплементація відповідає інтерфейсу контракту, та видає помилку при невідповідності. Для використання сервісу, міекрофронтенд викликає метод use з контрактом, та отримує назад типізований об'єкт. Аналогічно відбувається і для подій.

Також, у демонстраційному застосунку видно одну з переваг асинхронної комунікації, а саме слабке зв'язування між мікрофронтендами. Так, наприклад, модуль керування станом не знає про існування модулів редактора чи модуля ієрархії. А лише публікує подію, після кожної модифікації, і не має жодної інформації про те, хто саме підписаний на цю

подію. Такий підхід дозволяє легко додавати нові модулі, без жодних модифікацій існуючого коду.

Слабке зв'язування також значно спрощує тестування цих модулів. Так, наприклад, модуль-редактор можна тестувати в ізоляції без справжнього модуля керування станом, просто публікуючи тестові події.

Хоча демонстраційний застосунок використовує переважно горизонтальний поділ на мікрофронтенди, архітектура підтримує і вертикальний поділ на основі маршрутів. Можна легко розширити застосунок додатковими сторінками з різними наборами активних модулів. Так, наприклад, можна додати сторінку списку всіх документів, яка завантажує лише мікрофронтенд списку документів без редактора та панелі інструментів.

Висновки

У дипломній роботі вирішено актуальну науково-практичну задачу: проєктування та реалізації мікрофронтенд-фреймворку, який відповідає всім принципам мікросервісної архітектури. Створений фреймворк надає всі необхідні для побудови масштабованих веб-застосунків механізми, а саме: систему оркестрації завантаження та ініціалізації модулів, типобезпечну комунікацію за допомогою контрактів та подій, підтримку різних стратегій поділу застосунку, інструменти автоматизації конфігурації та налагодження.

Було проведено детальне дослідження шести фундаментальних принципів мікросервісної архітектури - незалежності розгортання, слабкої зв'язаності, високої згуртованості, організації навколо бізнес-можливостей, децентралізованого керування даними та ізоляції відмов. Розглянуто можливість їх застосування при розробці користувацьких інтерфейсів з урахуванням специфіки браузерного середовища виконання. Встановлено, що всі шість принципів можуть бути адаптовані та використанні для фронтенду також.

Здійснено аналіз найпоширеніших фреймворків для побудови мікрофронтенд застосунків. Виявлено ключові недоліки існуючих рішень: необхідність у спеціалізованих адаптерах для підтримки різних бібліотек інтерфейсу, відсутність типобезпечної комунікації на рівні компілятора, складність початкового налаштування, обмежену підтримку гібридного поділу застосунків. На основі виявлених недоліків сформовано вимоги до фреймворку та його архітектури.

Спроєктовано архітектуру фреймворку з чітким розподілом відповідальностей між компонентами. Реєстр інтерфейсів реалізує патерн сервіс-локатора для синхронної комунікації між модулями. Шина подій

забезпечує асинхронну комунікацію через взірець публікації-підписки на базі реактивних потоків RxJS і позбавляє необхідності створення глобального стану застосунку, який би порушував один з принципів мікросервісної архітектури, а саме: децентралізованого керування даними. Менеджер завантаження координує процес паралельного завантаження модулів та їх ініціалізації з урахуванням залежностей. Архітектура забезпечує незалежність від конкретних бібліотек інтерфейсу через делегування процесу рендерингу самим модулям. Створено Webpack плагін, який автоматизує процес налаштування федерації мікрофронтендів за допомогою централізованої конфігурації. Для головного застосунку плагін також генерує TypeScript типи віддалених модулів. Також реалізовано панель інструментів розробника, яка дозволяє відстежувати та аналізувати взаємодії між мікрофронтендами. Реалізовано інструменти для тестування систем побудованих за допомогою даного фреймворку.

Створено демонстраційний застосунок для демонстарції можливостей фреймворку.

Незважаючи на досягнуті результати, поточна реалізація фреймворку має певні обмеження, які окреслюють напрямки подальшого розвитку:
1. Відсутність підтримки серверного рендерингу. Поточна реалізація фреймворку обмежена рендерингом на стороні клієнта. Для застосунків, де критично важливим є швидкість початкового завантаження чи оптимізація для пошукових систем, відсутність серверного рендерингу може бути суттєвим обмеженням.
2. Відсутність автоматичного визначення залежностей. Поточна реалізація потребує явного оголошення всіх залежностей кожним мікрофронтендом за допомогою масивів uses та injects у конфігурації. Теоретично можна

автоматично визначати залежності за допомогою статичного аналіз коду мікрофронтендів та автоматично будувати граф залежностей.

Подальший розвиток фреймворку може бути спрямований на усунення виявлених обмежень.

Список використаних джерел

1. Mezzalira L. Building Micro-Frontends. — O'Reilly Media, 2021.
2. Newman S. Building Microservices. — O'Reilly Media, 2015.
3. Khononov V. Learning Domain-Driven Design. — O'Reilly Media, 2021.
4. Webpack. Module Federation [Електронний ресурс]. — Режим доступу до ресурсу: https://webpack.js.org/concepts/module-federation/
5. Single-SPA. Getting Started Overview [Електронний ресурс]. — Режим доступу до ресурсу: https://single-spa.js.org/docs/getting-started-overview
6. Piral. Documentation [Електронний ресурс]. — Режим доступу до ресурсу: https://piral.io/
7. RxJS. Overview [Електронний ресурс]. — Режим доступу до ресурсу: https://rxjs.dev/guide/overview
8. Martin R. C. Clean Architecture: A Craftsman's Guide to Software Structure and Design. — Prentice Hall, 2017.
9. Geers M. Micro Frontends in Action. — Manning Publications, 2020.
10. Evans E. Domain-Driven Design: Tackling Complexity in the Heart of Software. — Addison-Wesley, 2003.
11. Burns B. Designing Distributed Systems. — O'Reilly Media, 2018.
12. Fowler M. Patterns of Enterprise Application Architecture. — Addison-Wesley, 2002.
13. van Deursen S., Seemann M. Dependency Injection Principles, Practices, and Patterns. — Manning Publications, 2019.
14. Jackson C. Micro Frontends [Електронний ресурс]. — Режим доступу до ресурсу: https://martinfowler.com/articles/micro-frontends.html
15. Richardson C. Microservices Patterns [Електронний ресурс]. — Режим доступу до ресурсу: https://microservices.io/patterns/