Міністерство освіти і науки України
Харківський національний університет радіоелектроніки

Факультет *Комп'ютерних наук*
(повна назва)

Кафедра *Системотехніки*
(повна назва)

КВАЛІФІКАЦІЙНА РОБОТА
Пояснювальна записка

рівень вищої освіти *другий (магістерський)*

Дослідження трансформації архітектурного підходу розробки веб додатків від моноліту до мікро-фронтенду
(тема)

Виконав: студент *II* курсу, групи *СПРм-22-2*
Коваленко Г.С.
(прізвище, ініціали)

Спеціальність *122 Комп'ютерні науки*
(код і повна назва спеціальності)

Тип програми *освітньо-наукова*

Освітня програма *Системне проектування*
(повна назва освітньої програми)

Керівник *доцент каф. СТ Ребезюк Л.М.*
(посада, прізвище, ініціали)

Допускається до захисту

Зав. кафедри _____ *Гребеннік І.В.*
(підпис) (прізвище, ініціали)

2024 р.

Я як студент ХНУРЕ розумію і підтримую політику закладу із академічної доброчесності. Я не надавав і не одержував недозволену допомогу під час підготовки кваліфікаційної роботи. Використання ідей, результатів і текстів інших авторів мають посилання на відповідне джерело.

Студент _____ Коваленко Г.С.

Кваліфікаційна робота не містить відомостей заборонених до відкритого опублікування.

Керівник кваліфікаційної роботи _____ доц. Ребезюк Л.М.

Кваліфікаційна робота виконана у відповідності до діючих стандартів в Україні.

Керівник кваліфікаційної роботи _____ доц. Ребезюк Л.М.

Попередній захист проведено 21 червня 2024 р.

Керівник кваліфікаційної роботи _____ доц. Ребезюк Л.М.

Харківський національний університет радіоелектроніки

(назва закладу вищої освіти)

Факультет *Комп'ютерних наук*

Кафедра *системотехніки*

Рівень вищої освіти *другий (магістерський)*

Спеціальність *122 Комп'ютерні науки*
(код і повна назва)

Тип програми *освітньо-наукова*

Освітня програма *Системне проектування*
(повна назва освітньої програми)

ЗАТВЕРДЖУЮ:

Зав. кафедри *СТ*

проф. Гребеннік І.В.

" _____ " *2024 р.*

ЗАВДАННЯ

НА КВАЛІФІКАЦІЙНУ РОБОТУ

студентові *Коваленку Генріху Сергійовичу*
(прізвище, ім'я, по батькові)

1. Тема роботи *Дослідження трансформації архітектурного підходу розробки веб додатків від моноліту до мікро-фронтенду*

затверджена наказом університету від " *01* **"** *квітня* **2024р. №** *259Ст*

2. Термін подання студентом роботи до екзаменаційної комісії *19 червня 2024 р.*

3. Вихідні дані до роботи. *Дослідити процес, методи та засоби трансформації архітектурного підходу розробки веб додатків від моноліту до мікро-фронтенду. Перелік використовуваних програмних засобів: MacOs, Microsoft Word, WebStorm, NodeJS server, GoogleChrome.*

4. Перелік питань, що потрібно опрацювати в роботі
4.1 Вступ 4.2. Аналіз предметної галузі та постановка задач дослідження 4.2.1 Аналіз монолітної та мікросервісної архітектури 4.2.2 Огляд існуючих методів у реалізації мікросервісів web додатків та методів переходу web додатку від моноліту до мікро-фронтенду 4.2.3 Постановка задач дослідження 4.3 Дослідження архітектурної трансформації та моделей оцінки 4.3.1. Аналіз архітектурних особливостей монолітних додатків та їх реалізацій 4.3.2. Аналіз основних характеристик та реалізації мікросервісної архітектури 4.3.3 Аналіз технологій трансформацій web додатку 4.3.4 Аналіз проблем та викликів при переході до мікросервісів 4.3.5 Аналіз проблем та викликів при переході від моноліту до мікро-фронтенду 4.3.6 Аналіз критеріїв оцінки результатів трансформації 4.4 Реалізація переходу від моноліту до мікро-фронтенду 4.4.1. Аналіз архітектурних підходів Angular додатку 4.4.2 С 4.4.4. Аналіз отриманого застосунку 4.4.5 Створення дизайн бібліотеки 4.4.6 Перехід на мікро-фронтенд за використанням iframe технології 4.4.7 Перехід на мікро-фронтенд за допомогою бібліотеки simgle-spa 4.5 Оцінка трансформації додатку 4.6 Висновки

5. Перелік графічного матеріалу із зазначенням креслеників, схем, плакатів, комп'ютерних ілюстрацій (п.5 включається до завдання за рішенням випускової кафедри)
Титульний слайд кваліфікаційної роботи (слайд 1). Об'єкт, предмет дослідження, мета роботи (слайд 2). Характеристики монолітної архітектури (слайд 2). Монолітна архітектура застосунку (слайд 3). Характеристики мікро-фронтенд архітектури (слайд 4). Мікро-фронтенд архітектура застосунку (слайд 5). Причини трансформації в мікро-фронтенд (слайд 6). Поблеми при переході в мікро-фронтенд (слайд 7). Методи реалізації мікро-фронтенд (слайд 8). Підходи до зміни архітектури на мікро-фронтенд (слайд 9). Технології трансформації в мікро-фронтенд (слайд 10). Мікрофронтенд за допомогою IFRAME (слайд 11,12). Висновки (слайд 13).

6. Дата видачі завдання *01 квітня 2024 р.*

КАЛЕНДАРНИЙ ПЛАН

№	Назва етапів роботи	Терміни виконання етапів роботи	Примітка
1.	*Отримання завдання кваліфікаційної роботи*	*01.04.2024*	
2.	*Аналіз завдання, літератури та аналогів з теми кваліфікаційної роботи*	*01.04 — 10.04.2024*	
3.	*Аналіз монолітної та мікросервісної архітектури*	*26.04 — 05.05.2024*	
4.	*Огляд існуючих методів у реалізації мікросервісів web додатків та методів переходу від моноліту до мікро-фронтенду. Постановка задач дослідження*	*06.05 — 12.05.2024*	
5.	*Дослідження архітектурної трансформації та моделей*	*13.05 — 15.05.2024*	
6.	*Реалізація переходу від моноліту до мікро-фронтенду*	*16.05 — 23.05.2024*	
7.	*Перехід від одномодульної до багатомодульної*	*24.05 — 26.05.2024*	
8.	*Оцінка трансформації додатку*	*27.05 — 01.06.2024*	
9.	*Оформлення пояснювальної записки*	*02.06 — 12.06.2024*	
10.	*Створення комп'ютерної презентації та оформлення графічних матеріалів кваліфікаційної роботи*	*13.06 — 17.06.2024*	
11.	*Представлення на рецензування*	*18.06.2024*	
12.	*Представлення кваліфікаційної роботи до ЕК*	*19.06.2024*	

Студент _____ *Коваленко Г.С..*
(підпис)

Керівник роботи _____ *доц. Ребезюк Л.М.*
(підпис)

ABSTRACT

Master’s thesis: 129 p., — tabl., 17 fig., 1 app., 28 sources.

WEB APPLICATION ARCHITECTURE, MONOLITHIC ARCHITECTURE, TRANSFORMATION, MICROSERVICE APPROACH, MICRO-FRONTEND, FRAMEWORK, ANGULAR

Object of research - web application architecture.

Subject of study - transformation of the architectural approach to the development of web applications in order to increase their functional efficiency.

The purpose of the work - research on the transformation of the architectural approach to the development of web applications from a monolith to a micro-frontend in order to increase their functional efficiency.

Research methods – systematic approach, methods of functional analysis, comparative analysis of web applications using different architectural approaches, as well as experimental research on ways to implement microservice and microfrontend architectures in web development

To solve the problem, an analysis of scientific and technical literature and publications was carried out in order to compare the monolithic and microservice approach to the development of web applications. An analysis of the existing tools for the implementation of the microservice architecture of the web application was carried out. An analysis of the requirements for the transition from a monolithic architecture to a micro-frontend architecture was also carried out. As part of the implementation of the transition from a monolith to a micro-frontend in the Angular application, the creation of a library design, a transition to a micro-frontend using iframe technology, and a transition to a micro-frontend using the simgle-spa library were demonstrated.

РЕФЕРАТ

Кваліфікаційна робота: 129 с., — табл., 17 рис., 1 дод., 28 джерел.

АРХІТЕКТУРА ВЕБ-ДОДАТКІВ, МОНОЛІТНА АРХІТЕКТУРА, ТРАНСФОРМАЦІЯ, МІКРОСЕРВІСНИЙ ПІДХІД, МІКРО-ФРОНТЕНД, ФРЕЙМВОРК, ANGULAR

Об'єкт дослідження − архітектура веб-додатків.

Предмет дослідження – трансформації архітектурного підходу розробки веб-додатків з метою підвищення їх функціональної ефективності.

Мета роботи – дослідження трансформації архітектурного підходу розробки веб-додатків від моноліту до мікро-фронтенду з метою підвищення їх функціональної ефективності.

Методи дослідження – системний підхід, методи функціонального аналізу, порівняльний аналіз веб-додатків, що використовують різні архітектурні підходи, а також експериментальне дослідження шляхів впровадження мікросервісної та мікрофронтендної архітектур у веб-розробці.

Для вирішення поставленого питання був проведений аналіз науко-технічної літератури та публікацій з метою порівняння монолітного і мікросервісного підходу для розробки веб-додатків. Проведено аналіз існуючих інструментів для реалізації мікросервісної архітектури веб-застосунку. Також проведено аналіз вимог для переходу від монолітної архітектури до мікрофронтенд архітектури. В рамках реалізації переходу від моноліту до мікро-фронтенду у Angular додатку продемонстровано: створення дизайн бібліотеки, перехід на мікро-фронтенд за використанням iframe технології та перехід на мікро-фронтенд за допомогою бібліотеки simgle-spa.

ЗМІСТ

СКОРОЧЕННЯ ТА УМОВНІ ПОЗНАКИ

API – Application Programming Interface;

HTML – HyperText Markup Language (мова гіпертекстової розмітки);

MVP – minimum Viable Product;

DRY – Do not Repeat Yourself;

SSR – Server-Side Rendering;

SPA – Single Page Applications.

ВСТУП

XXI сторіччя відзначається постійним зростанням програмної індустрії. В наш час комп'ютерна техніка покликана не лише для професійного використання чи вирішення специфічних завдань, а й для вирішення повністю побутових питань кожної людини в суспільстві. Вже стало досить звично носити з собою документи в мобільному додатку "Дія" чи відкривати банківський рахунок і оплачувати комунальні послуги за допомогою смартфону. Одним з найлегших способів запропонувати користувачу певні послуги є web застосунок, адже він не потребує інтеграцію в систему користувача додаткових програм крім браузера, не потребує специфічних вимог від комп'ютерної техніки та стає доступним вже зразу. Через це web застосунки постійно еволюціонують і почавши з просто інформаційних ресурсів зростають до повноцінних систем управління продажами природних ресурсів населенню, банківської системи, систем нагляду та контролю за безпекою у містах, соціальних мереж для обміну інформацією і та інше.

Таке підвищення складності web додатків потребує постійної зміни та еволюції у підходах розробки програмного забезпечення, постійно з'являються нові фреймворки для написання коду, модернізується та покращується мова програмування JavaScript. Разом з тим ми можемо відмітити еволюцію проектування web додатків бо через навантаження та більший об'єм застосунків починається велика кількість питань які пов'язані з розмірами команд які будуть працювати над певним проектом, підтримуваністю коду (оскільки через постійну еволюцію код швидко застаріває, а команда хоче використовувати новітні підходи до написання та імплементації функціоналів), безпеки застосунку, його модульності і можливості перевикористання окремих частин продукту. Одним з способів комплексного вирішення цих питань є перехід від звичної всім, монолітної архітектури web застосунків до більш гнучкої мікросервісної архітектури. Оскільки монолітна архітектура характеризується тим що вся функціональність знаходиться в одній кодовій

базі, написана на одному фреймворку і за допомогою одного підходу вона може викликати певні незручності з масштабуванням команди, імплементації нових технологій у розробку і іншим. З іншого боку мікросервісна архітектура розбиває додаток на певні окремі частини які не залежать одна від одної. Це дозволяє вести паралельну розробку загального функціоналу різними командами, використання зовсім різних методів розробки, а також різних фреймворків. Також слід окремо наголосити, що якщо ці додатки незалежні один від одного вони можуть бути використані окремо і по суті можна сказати що це є окремі продукти які виконують певні функції і в певному випадку збираються в один великий продукт.

При певних перевагах мікросервісної архітектури перехід на неї з моноліту досить тяжке завдання (особливо в період пост продакшн), оскільки може викликати чисельні проблеми такі як:

– розбиття програми на окремі модулі та сервіси руйнує побудовані зв’язки моноліту і швидше за все призведе до повної втрати функціональності;

– зміна фреймворку імплементації додатку (певного сервісу) призведе до повного переписування сорс коду;

– потрібно добре планувати процес переходу та процес розгортання мікросервісів для побудови підтримуваних зв'язків

Попри такі незручності перехід від монолітної архітектури web додатків до мікросервісної може призвести до значного покращення ефективності розробки, стабільності, безпеки та підтримуваності застосунку.

1 АНАЛІЗ ПРЕДМЕТНОЇ ГАЛУЗІ ТА ПОСТАНОВКА ЗАДАЧ ДОСЛІДЖЕННЯ

1.1 Аналіз монолітної та мікросервісної архітектури

Монолітна архітектура є традиційною уніфікованою моделлю для розробки програмного забезпечення. Монолітний, у цьому контексті, означає «компонований все в одному шматку». Відповідно до Кембриджського словника, прикметник монолітний також означає «занадто великий» і «неможливий для змін» [1].

Основною рисою монолітної архітектури є те що компоненти такого додатку тісно пов’язані між собою, як правило клієнтський і серверний код є частиною однієї кодової бази, займають один і той самий репозиторій і за великим рахунком команда працює лише з ним. Гарним прикладом фреймворку який зі старту реалізує моноліт є фреймворк від компанії Vercel, а саме NextJs. Цей фреймворк досить новий і приходить на заміну бібліотеці React, він пропонує використовувати підхід SSR (підходу в тому, що вся сторінка рендериться на сервері при першому запиті [2]) та пропонує свої методи написання бекенд частини замість використання сторонніх фреймворків чи інших мов програмування.

Розглянемо мінуси монолітної архітектури. Додатки які спроектовані за монолітною архітектурою як правило не роздільні і їх частини тісно пов’язані одна з одною, з цього випливає, що якщо потрібно змінити певний функціонал то повністю можливо, що доведеться вносити зміну в один компонент і підлаштовувати інші оскільки можлива не правильна робота застосунку в цілому, також після подібних змін може знадобитись регресійне тестування (це набір тестів, спрямованих на виявлення дефектів у вже протестованих модулях додатка [3]).

В монолітній архітектурі є проблеми з масштабуванням і пост продакшн підтримкою, оскільки всі компоненти настільки тісно зв’язані на етапі

проектування, що дописати навіть не дуже великий функціонал у вже робочий додаток складно і може зайняти багато часу.

Монолітні додатки мають певні проблеми зі швидкістю розробки, хоча на початкових етапах робота над монолітом набагато легша і швидша ніж над мікросервісом, але через деякий час коли кодова база зростає швидкість розробки починає падати, оскільки потрібний додатковий час на планування архітектури і планування можливих ризиків, які можуть виникнути в інших частинах продукту.

Монолітні додатки дуже чутливі до зміни технологій і технологічних рішень. Так як монолітні додатки проектуються і пишуться з використання певного одного фреймворку вони стають дуже чутливими до його версії, підвищення версії фреймворку може стати певною проблемою оскільки в ньому може бути змінений підхід до певної функціональності (наприклад, підхід до моделі формування запиту до API) і це призведе до повної втрати функціональності всього застосунку. Повна заміна фреймворку буде взагалі неможлива оскільки це призведе до повного переписування сосркоду за новими правилами. Причому неможливо знайти легкого вирішення цієї проблеми оскільки, якщо не оновлювати фреймворк то кодова база дуже швидко застаріє і підтримувати її стане складно не тільки з точки зору часу а й з точки зору людських ресурсів, оскільки знайти програміста який знає як використовувати певний фреймворк буде ставати складніше і складніше. Як приклад можна навести таку мову програмування як CoffeScript. CoffeeScript – мова програмування, що транслюється в JavaScript. CoffeeScript додає синтаксичний цукор у дусі Ruby, Python і Haskell для того, щоб покращити читання коду і зменшити його розмір. В середньому для виконання однакових дій на CoffeeScript потрібно в два рази менше рядків, ніж JavaScript [4]. В свій час до оновлення JavaScript до версії ES-6, CoffeeScript був досить популярною мовою програмування, але після цього оновлення CoffeeScript стали менше використовувати і більше забувати оскільки він більше не додавав ніяких бонусів поряд з класичним JavaScript. Як підсумок, в 2024 році рекрутери

продуктових компаній зіштовхуються з певними труднощами оскільки для підтримки застарілого моноліту потрібні співробітники які можуть використовувати CoffeeScript, а подібних кандидатів на ринку наразі дуже мало. Іншим прикладом може бути досить популярна в свій час бібліотека Backbone JS, що базується на MVP інфраструктурі з досить жорсткими зв'язками [5]. Свого часу через популярність цієї бібліотеки на ній було написано досить багато великих проектів, але після втрати нею популярності кількість спеціалістів знайомих з нею швидко скоротилось. Це призвело до того, що великі продуктові компанії вимушені самостійно готувати спеціалістів, а також шукати шляхи переходу продукту на інший фреймворк/бібліотеку без втрати функціональності.

Розглянемо плюси монолітної архітектури. Основним плюсом монолітної архітектури є простота її розробки, на проектах які не великі за об'ємом кодової бази (такі як інформаційні ресурси) немає необхідності в переході на мікрофронтенд оскільки при моноліт набагато легше налаштовувати для роботи, процес деплою також не ускладняється, також можна сказати, що процес введення нових працівників в робочий процес на монолітній архітектурі не такий складний, як при мікросервісах.

Продуктивність – у централізованій базі коду та сховищі один API часто може виконувати ту саму функцію, яку виконують численні API з мікросервісами [6].

Легкий доступ до даних – в пов'язаній структурі моноліту досить легкий доступ до даних інших компонентів і дуже часто не потрібно навіть робити запит до API для отримання даних, через те що дані вже були отримані іншим компонентом чи сторінкою. В якості прикладу можна навести бібліотеку Redux якщо вона використовується в монолітній структурі то забезпечує безпроблемний доступ до стейту всього застосунку і через стейт є повна можливість отримати будь-які дані в будь-якій частині застосунку.

Як висновок, можна сказати, що монолітна архітектура для веб-додатку буде ідеальна лише в тому випадку, якщо:

– додаток планується невеликої кодової бази;

– продукт не планує зростання і великих пост продакшн модифікацій;

– не планується залучення великої кількості розробників.

Мікросервісна архітектура web-додатків (мікрофронтенд) – це архітектура яка базується на розгортанні незалежних додатків які збираються в один. Іншими словами, за допомогою мікрофронтенду ми можемо розподілити між різними командами певні частини глобального функціоналу і кожна команда буде приймати участь в розробці своєї частини, при цьому кожна з команд буде працювати атомарно від інших і по суті розробляти свій окремий продукт. Зауважимо що кожна команда може окремо вибрати для себе не тільки стиль написання коду, а й бібліотеку/фреймворк для написання певного додатку. Після написання двох або більше додатків вони інтегруються один в одний за допомогою певного вибраного підходу і представляють собою окремий продукт, не втрачаючи при цьому можливості за потреби бути використаним окремо один від одного.

Розглянемо деякі патерни, які властиві мікрофронтенду:

а) Module Federation – об’єднання модулів є основною концепцією мікрофронтендової архітектури. Це дозволяє динамічно завантажувати код і ресурси з різних мікроінтерфейсів. Цей підхід сприяє спільному використанню коду та повторному використанню, зменшуючи розмір початкового завантаження сторінки та покращуючи продуктивність [7];

б) Shell and Module Approach – підхід оболонки та модуля передбачає створення програми оболонки, яка діє як хост для всіх мікроінтерфейсів. Ця оболонка забезпечує інтеграцію мікроінтерфейсів, що робить її важливою частиною архітектури [7];

в) Microservices and Micro Frontends – мікросервіси та мікроінтерфейси йдуть рука об руку. Мікросервіси забезпечують серверну логіку та дані, тоді як мікроінтерфейси обробляють інтерфейс користувача. Такий розподіл завдань дозволяє командам працювати незалежно, сприяючи ефективній розробці та обслуговуванню [7].

Розглянемо у загальному мінуси мікрофронтенду:

а) важкий обмін даними і зв’язки. Як описувалось вище продукти які створюються за допомогою архітектури мікрофронтенду це певні додатки які використовуються окремо один від одного і після об’єднуються в один великий продукт. Дивлячись на це виникає проблема як саме встановити зв’язок між цими частинами. Для вирішення цієї проблеми потрібно підібрати ефективні стратегії комунікації щоб покращити досвід юзера від користування загальним продуктом;

б) стиль написання і стиль коду. Також як писалось раніше, незалежні додатки які розробляють різні команди можуть не підтримувати загальний стиль коду продукту, якщо в моноліті ми можемо налаштувати лінтер та або пре коміт хуки то в мікросервісному підході це доведеться налаштовувати в кожному проекті окремо. В будь-якому випадку потрібен постійний контроль за стилем коду, оскільки в іншому випадку ми можемо отримати проблеми з людським капіталом підприємства, а саме співробітник компанії який знаходиться в одній команді не зможе швидко (без курсу “onboarding”) тимчасово замінити іншого працівника, що в подальшому може призвести до ще більшого розростання команд;

в) версіонування. В мікрофронтенді постає питання версіонування і сумісності версій, через те що версії додатків можуть відрізнятись, то час від часу вони можуть бути не сумісні одна з одною (Наприклад в одному додатку була реалізована частина функціоналу, а в іншому ні. це може призвести до неможливості використання юзером певної частини функціоналу). Через це необхідно дуже ретельно відслідковувати версії і зміни в частинах продукту, особливо під час розгортання основного продукту;

г) усунення несправностей. Виявлення та усунення проблем в архітектурі мікрофронтенду може бути складнішим, ніж у монолітних програмах. Ефективні інструменти моніторингу та налагодження необхідні для спрощення процесу усунення несправностей;

д) збільшення загальної складності. Поряд з тим що окремі додатки розділені і не настільки складні для розуміння ніж один великий моноліт, виникає проблема складності додатку в цілому. Через те що загальний додаток має комплексний вигляд в перший час роботи на проекті співробітники компанії можуть плутатись яка сторінка відповідає тому чи іншому проекту. Більш явною ця проблема постає при втраті можливості обміну знаннями між співробітниками (як приклад компанія купляє вже написаний продукт і співробітників які його писали не можливо знайти) в цей час виникає проблема встановлення середовища для розробки, оскільки співробітники компанії не можуть знати які саме проекти за що відповідають і буде необхідний час для вивчення і розгортання мережі;

е) порушення принципу DRY (Do not repeat yourself). Під час написання мікрофронтендових додатків в будь’якому випадку виникне проблема з дублюванням коду, оскільки час від часу функції і методу для конвертування даних потрібно буде повторювати в декількох різних окремих проектах і винести певні функції в окремий модуль, як в моноліті не можливо;

ж) розмір бандлу. Загальний розмір бандлу моноліту буде завжди менший за сукупність бандлів додатків мікрофронтенду через те що в кожному з додатків буде встановлений свій фреймворк і свій набір бібліотек, більше того встановлені бібліотеки можуть дублюватись оскільки в кожному з додатків може бути встановлена бібліотека для роботи з датами та часом та або бібліотеки для запитів на бекенд;

з) відсутність всього глобального. Під час розробки програмного забезпечення прийнято виносити значення що повторюються між компонентами (такі як певні кольори елементів, певні числові значення і таке інше) в глобальні змінні, робиться це для того, щоб в разі необхідності зміни цих значень можливо було змінити це в одному місці і зміни запрацювали по всьому проекту зразу. В підході мікрофронтенду про це слід забути, оскільки через те що додатки не пов’язані один з одним між ними не може бути нічого глобального.

Плюси мікросервісної архітектури (мікрофронтенду):

а) масштабованість і повторне використання. Перш за все ми можемо розглянути, що команди розробників можуть збільшувати чи зменшувати функціонал певного конкретно додатку не впливаючи на продукт в цілому. Також тут ми можемо винести в окремий блок, що одна з команд може написати свою інтерфейсну бібліотеку додати storybook, внести до неї окремі компоненти для використання по всьому продукту і ре використовувати їх де потрібно;

б) самостійна розробка і розгортання. Через те що кожен з компонентів структури продукту є незалежним це дає розробникам можливість писати конкретний додаток окремо від інших. Також розробники можуть розгортати певний додаток окремо від інших. Така можливість є безперечною перевагою від моноліту, оскільки час від часу моноліт не може бути розгорнутий для юзерів через критичні помилки окремого модуля чи окремого функціоналу, в варіанті ж мікросервісної архітектури додатки які не мають критичних помилок можуть бути розгорнуті, ті ж що їх мають йдуть на доопрацювання і можуть бути розгорнуті окремо як тільки команда закінчить виправлення і функціонал буде протестовано;

в) зменшення проблемних місць під час розробки. В монолітній архітектурі з’являються певні проблемні місця проекту, через збільшення кодової бази з’являється місце в коді (частині моноліту) яке може потенційно впливати на моноліт в цілому. Причому допущена помилка саме в цій частині призводить до не стабільності роботи і можливої втрати функціоналу моноліту в цілому. При мікросервісній архітектурі подібні місця або не виникають, або впливають локально на один додаток не зачіпаючи функціонал інших.

З наведеного вище, можна зробити основний висновок, що як монолітна так і мікросервісна архітектура мають свої суттєві мінуси і свої суттєві плюси. Я б сказав що це набір можливостей і кожен з архітектурних підходів, мало того що має право на існування, більш того один підхід не зможе до кінця

витіснити інший підхід. Оскільки монолітна архітектура прекрасно підходить для простих проектів або не великих проектів, або проектів яким не потрібна підтримка в майбутньому. З іншого ж боку мікросервісна архітектура повністю покриває можливість постійної розробки і підтримки функціоналу, зміну фреймворків і перехід на нові, переписування окремих додатків не зупиняючи функціоналу основного продукту.

При цьому використання будь-якої з цих архітектур потрібно розуміти, що для того щоб отримати бонуси від архітектури вона повинна бути правильно спроектована і повинні бути включені відповідні технологічні рішення для підтримки вибраної архітектури. Оскільки неправильний підхід до проектування і виконання може призвести до проблем (як, наприклад, неправильне розбиття на мікросервіси може призвести до виникнення проблем в розробці настільки великого масштабу, що доведеться змінювати архітектуру в цілому і переписувати продукт).

1.2 Огляд існуючих методів у реалізації мікросервісів web додатків та методів переходу web додатку від моноліту до мікро-фронтенду

Для реалізації мікрофронтенд архітектури використовують різні методи які можна поділити на наступні методи:

– методи фреймворків та бібліотек;

– методи JavaScript;

– методи html;

– методи розділення баз і бекенд коду.

Методи фреймфорків і бібліотек – з розвитком складності веб застосунків розробники почали використовувати фреймворки та бібліотеки для реалізації коду Single Page Application. Фреймворки пропонують розробникам такі підходи як роутінг та модульна структура. Якщо уважніше розглянути роутінг у фреймворках то за певним URL розташований певний контекст веб додатку загалом при використанні фреймворку весь код для всіх сторінок знаходиться в

одній кодовій базі, але за допомогою DevOps інструментів на стороні сервері можна розмістити більше одного додатку які будуть пов’язані роутінгом в цьому вигляді ми можемо побачити класичну мікрофронтенд архітектуру в якій певні сторінки додатку розгортаються окремо від інших. Що до модульності, то не можна сказати що підхід модульності фреймворків і мікрофронтенд це тотожні поняття, скоріше модульність це перша сходинка до мікрофронтенду. Модуль це окрема частина функціоналу веб застосунку яка не пов’язана з іншими модулями чи їх частинами. Теоретично модуль може бути окремим додатком і мати власну структуру компонентів, модуль може бути винесений окремо від іншої частини застосунку без втрати функціональності і це нагадує архітектурний підхід мікрофронтенду. Однак одною з особливостей мікрофронтенд архітектури і загалом мікросервісної архітектури є окреме розгортання частин продукту, модуль не може бути разгорнутий окремо від загального сорс коду продукту адже пов’язаний архітектурою фреймворку, отже для реалізації мікрофронтенду на основі модулю потрібно винести модуль з фреймворку і додати окремі інструкції збірників (таких як webpack) для розгортання його сорскоду.

Методи JavaScript – так само як і з попереднім методом полягає в використання клієнтського JavaScript для завантаження різних мікрофронтендів на основі URL адреси, також починаючи зі стандарту ES-6 JavaScript підтримує модулі на які можна розділити застосунки не використовуючи фреймворк

Методи HTML – під методами HTML розуміється використання тегу iframe який дозволяє відкривати певне посилання в середені вікна веб застосунку і показувати сторонній ресурс[19].

Методи розділення баз і бекенд коду - більше підходить для використання під час SSR, під час такої імплементації сторінка генерується на стороні бекенд і приходоить на клієнтську сторону вже зенерованою, в такому випадку так само як і з методами модулів додаток розділяється на різні частини після чого база даних яка біла монолітною розділяється для кожного застосунку окремо.

Підходами до переходу від моноліту до мікросервісної архітектури є наступні:

а) розбиття на компоненти або на модулі, тобто монолітний застосунок розбивається на свої логічні частини такі як модулі (оскільки модуль повинен мати все необхідне для роботи окремо від інших частин застосунку) після чого окремі модулі інтегруються в новий застосунок і розгортаються на сервері;

б) реорганізація за доменами, тобто розділення функціональності за доменами або функціональними зонами, щоб кожен мікрофронтенд відповідав за певний аспект додатку;

в) використання мікросервісів, тобто:використання мікросервісної архітектури для фронтенду, де кожен мікрофронтенд може мати свій власний сервер та базу даних;

г) поступове впровадження, тобто впровадження мікрофронтендів поступово, починаючи з нових функцій або компонентів, і поступово переносити старий функціонал

1.3 Постановка задач дослідження

Після порівняння характеристик монолітної архітектури і мікрофронтенду можна зробити висновок, що питання переходу від моноліту до мікрофронтенду є досить актуальним оскільки буд-який продукт може зіткнутися з проблемами які були описані вище і їх необхідно буде вирішити максимально зберігши функціональність і час. Також необхідно знайти оптимальний підхід для переходу від моноліту до мікрофронтенду. Для досягнення цієї задачі буде зроблено:

– створити тестовий додаток для екранування проблем монолітної інфраструктури;

– проаналізувати певні можливі мінуси цього додатку які, можуть виникнути при подальшому розростанні;

– переробити тестовий додаток на мікросервісну архітектуру за допомогою iFrame підходу;

– проаналізувати труднощі, які виникли і можуть виникнути в майбутньому при iFrame підході;

– переробити тестовий додаток на мікросервісну архітектуру за допомогою підходу бібліотек;

– проаналізувати труднощі, які виникли і можуть виникнути в майбутньому при застосуванні підходу бібліотек;

– проаналізувати отримані результати і зробити висновок про оптимальний підхід для реалізації переходу від моноліту до мікрофронтенду.

Очікується, що це дослідження дасть розуміння процесу перетворення веб-додатків із моноліту на мікросервіс та його вплив на різні аспекти розробки. Результати дослідження будуть корисними для розробників, які розглядають перехід на архітектуру мікросервісів, допомагаючи їм приймати зважені рішення та ефективно впроваджувати цю трансформацію.

2 ДОСЛІДЖЕННЯ АРХІТЕКТУРНОЇ ТРАНСФОРМАЦІЇ ТА МОДЕЛЕЙ ОЦІНКИ

2.1 Аналіз архітектурних особливостей монолітних додатків та їх реалізацій

Як розглядалось вище монолітна архітектура веб-додатків є традиційним підходом до розробки, де весь функціонал додатку реалізований як єдиний, великий блок коду.

Основні характеристики:

– монолітний моноліт, тобто весь функціонал додатку розгортається як єдиний моноліт, що означає, що всі його компоненти, такі як інтерфейс, бізнес-логіка та база даних, знаходяться разом;

– централізоване управління, тобто управління функціоналом та взаємодіями між компонентами здійснюється централізовано в межах одного додатку;

– простота розгортання, тобто монолітні додатки зазвичай досить прості у розгортанні, оскільки вони вимагають лише одного середовища для запуску;

– збільшення складності з ростом, тобто при збільшенні обсягу функціоналу та розміру додатку стає складніше розуміти та підтримувати його.

До переваг слід віднести:

– простота розробки, а саме великий блок коду спрощує розробку, оскільки розробники можуть швидко отримати доступ до всіх компонентів;

– простота відлагодження, а саме відлагодження помилок може бути простішим, оскільки всі компоненти додатку взаємодіють між собою без виклику віддалених служб.

До недоліків слід віднести:

– складність масштабування, а саме зі збільшенням обсягу додатку та навантаження на нього, може виникнути проблема масштабування, оскільки всі компоненти знаходяться в одному моноліті;

– швидкість впровадження змін, а саме внесення змін у великий монолітний додаток може бути складним і ризикованим, оскільки зміни в одній частині можуть вплинути на інші;

– супровідність, а саме підтримка та розвиток монолітних додатків може стати проблемою з часом, оскільки складність коду зростає, архітектура може стати застарілою.

Монолітні додатки є простими для розробки та відлагодження на початкових стадіях проекту, але з часом вони можуть зазнати проблем з масштабуванням та підтримкою. Ці архітектурні особливості важливо враховувати при вивченні процесу переходу до мікросервісної архітектури.

Для початку розглянемо модульність в архітектурі застосунку, хоча вона і не є архітектурним підходом але саме модульність в архітектурі додатків передувала мікрофронтенду.

Модуль в веб додатку – це певна частка функціоналу, яка є незалежна від інших частин функціоналу, імплементована в загальну структуру і має набір своїх компонентів і бізнес логіки.

Кожен модуль складається з компонентів які в свою чергу являють собою будівельні блоки додатку і повинні містити певну візуальну частину яка демонструється користувачу. Компоненти поділяються на ті які можна перевикористати в інших частинах додатку (необхідні для виконання принципу Do not repeat yourself), та ті які являються основними для певної сторінки і куди підключаються інші.

Бізнес логіка це певна частину функціоналу яка пов’язана з роботою додатку така як запити до арі певні обчислиння чи робота з даним. Відносно використовуємого в додатку фреймворку/бібліотеки бізнес логіка може знаходитись або в сервісах (фреймворк Angular) або в хуках (бібліотека React).

Структура модулів в веб додатку базується або на правилах фреймворку, або розробляється розробками додатку. Розглянемо на прикладі Angular і React структуру побудови веб додатку.

Angular – це веб-фреймворк, розроблений компанією Google, який

використовується для створення динамічних односторінкових додатків (Single Page Applications, SPA) [8]. Він дозволяє розробникам будувати високопродуктивні веб-додатки з використанням мови програмування TypeScript.

До основних особливостей Angular слід віднести:

– компонентна архітектура, тобто Angular побудований на компонентній моделі, де веб-програма розділена на невеликі незалежні частини. При цьому кожен компонент містить свій власний шаблон, стиль та логіку, які дозволяють розробникам легко організовувати код та керувати ним;

– двостороння прив'язка даних, тобто Angular надає механізм двосторонньої прив'язки даних для автоматичної синхронізації даних між компонентами та їх моделями, що дозволяє легко обробляти та оновлювати дані в режимі реального часу;

– залежність та впровадження залежностей, тобто Angular має вбудовану систему впровадження залежностей, яка дозволяє легко керувати залежностями між компонентами, що спрощує розробку, тестування та обслуговування коду;

– маршрутизація, тобто Angular надає механізм маршрутизації, який дозволяє створювати багатосторінкові програми. Його можна використовувати для визначення того, який компонент відображається для кожної URL-адреси;

– розширюваність, тобто Angular має широкий спектр функцій, які можна розширити за допомогою сторонніх модулів. Існує велика кількість сторонніх бібліотек та модулів, що розширюють можливості фреймворку.

Загальну структуру веб-додатку на основі Angular наведено на рисунку 2.1.

Як ми бачимо веб додаток написаний за допомогою Angular як і за загальним правилом будується за допомогою модулів в які входить певна множина компонентів і сервіси. Що цікаво на цьому графіку ми можемо побачити додаткові елементи такі як App Module, Shared Module. App Module в даному випадку буде відповідати за основний функціонал застосунку, в ньому будуть знаходитись компоненти і сервіси котрі не перевикористовуються в

застосунку і відповідають патерну програмування Singleton який означає що в додатку буде створений лише один інстанс певного класу [9].

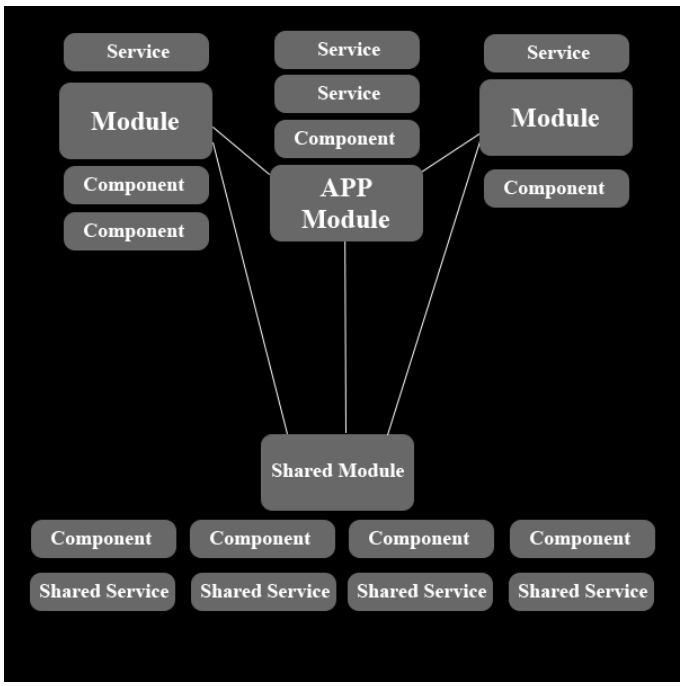

Рисунок 2.1 – Структура веб додатку, написаного за допомогою Angular

Shared Module це модуль в який підключаються компоненти та сервіси з бізнес логікою які будуть перевикористовуватись в застосунку і будуть мати більше одного інстансу. До цього модулю можуть бути додані модальні вікна, певні написані кнопки, сервіси для математичних операцій, сервіси для операцій з LocalStorage, загальні реквести і таке інше.

React JS – це відкритий JavaScript-фреймворк, а точніше, бібліотекою JavaScript, яка використовується для розробки інтерфейсів користувача. Він був створений компанією Facebook і швидко набув популярності серед розробників з усього світу. Реакт дозволяє ефективно створювати застосунки з високою продуктивністю і масштабованістю. Одним з ключових концепцій у React JS є компоненти. Вони представляють собою незалежні блоки коду, які відповідають за рендеринг певної частини користувацького інтерфейсу[10]. На відміну від Angular, React як бібліотека не має певної особливої структури застосунку і передає структуру на розсуд команди розробників. За

домовленістю на проекті загальні компоненти виносяться на директорію вище, а локальні компоненти знаходяться в одній директорії з певним компонентом з яким вони використовуються. Що до сервісів, то як таких їх в реакті немає і за домовленістю між розробниками певна логіка можу бути винесена в окремі javaScript файли (helpers) або в реатк хуки (Hooks – API, що дозволяє писати функціональні компоненти зі станом та використовувати інші можливості реакту без написання класів. Доступно в прев'ю версі [11]).

Як було описано вище підходи в модульній архітектурі додату можуть здатись схожими на мікрофронтенд оскільки ми маємо певну архітектуру модулі якої виокремлюють певний функціонал і є незалежними один від іншого, але насправді це не так оскільки всі ці модулі при відкритті додатку все ще завантажуються на ккомп'ютер користувача разом, що може спричинити певні незручності з швидкістю завантаження, більше того навіщо завантажувати певний контент додатку якщо користувач може цю частину додатку ніколи і не відкрити, або відкривати в край рідко. В якості прикладу можна навести сторінку користувача в таких продуктах як Приват 24 або ж Monobank, через те що ці додатки є додатками банків користувачу немає необхідності дуже часто заходити в свій профіль і змінювати аватар або прикріплений емейл, через це постає питання навіщо завантажувати ці дані при відкритті користувачем додатку і чи можна їх завантажувати лише в той момент коли користувач захоче потрапити на конкретну стоорінку? Ця проблема була вирішена на рівні фреймворка/бібліотеки за допомогою лінивого завантаження модулів. Ліниве завантаження Angular - ліниве завантаження в Angular було реалізоване на рівні роутінгу і представляло собою що певні модулі будуть викликані для завантаження лише тоді коли користувач буде перенаправлений за певним посиланням (перейде на певний роут) додатку [12]. В React ця проблема була вирішена я допомогою функції вищого порядку (функції високого порядку (high order function) – це функції, які або приймають функцію як параметр, або повертають функцію, або і те, й інше [13]) lazy та компоненту Suspense [14]. Необхідна частина функціоналу імпортується за

допомогою функції вищого порядку lazy після чого вставляється певна умова і імпорт огортається в компонент Suspense [15]. Після таких маніпуляцій в React огорнута в Suspense частина коду буде завантажуватись лише за умовою що передує цьому к компоненту.

Як ми бачимо деякі проблеми які вирішує архітектура мікрофронтенду можна вирішити і за рахунок фреймворку/бібліотеки, однак не всі.

2.2 Аналіз основних характеристик та реалізації мікросервісної архітектури

Мікросервісна архітектура веб-додатків є підходом до розробки, який полягає в розбитті функціоналу на невеликі, незалежні мікросервіси, що можуть бути розвиваними, впроваджуваними та масштабованими окремо. Нижче ми розглянемо основні характеристики та переваги/недоліки мікросервісної архітектури порівняно з монолітною.

Основні характеристики:

– розбиття на мікросервіси, тобто функціонал додатку розбивається на невеликі, незалежні мікросервіси, які можуть бути розгорнуті та масштабовані окремо;

– децентралізована архітектура, тобто кожен мікросервіс може бути розгорнутий на власному сервері або контейнері, що дозволяє їм бути незалежними один від одного;

– ізольованість та незалежність, тобто мікросервіси мають свою власну базу даних та можуть використовувати різні технології та мови програмування;

– гнучкість та масштабованість, тобто кожен мікросервіс може бути масштабованим незалежно від інших, що дозволяє ефективно використовувати ресурси.

До переваг слід віднести:

– гнучкість у відділенні функціональності, тобто мікросервіси дозволяють розробникам гнучко розподіляти функціональність між ними, що полегшує

розвиток та підтримку системи;

– легкість масштабування, тобто кожен мікросервіс може бути масштабований окремо в залежності від навантаження, що полегшує управління ресурсами.

До недоліків слід віднести:

– складність управління, а саме, з великою кількістю мікросервісів може виникнути складність у керуванні, моніторингу та координації між ними;

– комплексність тестування та відлагодження, а саме тестування та відлагодження мікросервісів може бути складнішим через їх розподілену природу та залежності.

Мікросервісна архітектура дозволяє створювати більш гнучкі та масштабовані системи в порівнянні з монолітною архітектурою, проте вона вимагає більшої уваги до керування та координації між окремими компонентами. Вибір між цими двома архітектурними підходами залежить від потреб продукту та замовника.

2.3 Аналіз технологій трансформацій web додатку

Мікрофронтенд дозволяє не тільки окремо завантажувати певні частини, а і розгортати їх окремо в підході ж фреймворків ми бачимо, що разгортання всеодоно буде проходити разом всіх частин. Також ми бачимо, що при виборі підходів з фреймворків ми все одно зав’зуємось на певний окремий фреймворк чи бібліотеку як то React чи Angular і не можемо відійти від неї на період написання всього продукту, цю проблема як раз і була вирішена за допомогою мікрофронтенду.

Існує декілька технологій трансформації до мікрофронтенду серед яких:

– підключення додатків як бібліотек за допомогою node_modules;

– технологія Iframe;

– бібліотека TailorJs;

– Single-spa;

– nx.dev.

Отже, підключення додатків як бібліотек – це один з видів імплементації сторонніх додатків в архітектуру продукту. При розробці певного додатку час від часу виникає необхідність спростити розробку і для цього використовують npm пакети – це певні частини функціоналу, які написали інші розробники, які мають певну версію і прописуються в файлі package.json. Після того, як ці бібліотеки написані і встановлені, вони додаються в папку node_modules і можуть бути використані по всьому додатку, де вони встановлені, як правило, такі бібліотеки мають набір компонентів та або функцій, які команда розробників може використовувати в своїй роботі. При такому підході ми можемо зробити певну частину продукту і написати певний функціонал після чого розгорнути його як бібліотеку і додати в основний продукт. Це підхід не є класичним мікрофронтендом оскільки, по-перше, не може бути розгорнутий окремо як продукт оскільки являє собою бібліотеку, по-друге, при встановлені бібліотеки в основний продукт вона буде додана в загальний банд продукту і буде ініціалізуватись при вході до продукту. Хоча цей підхід не є повністю мікрофронтендом час від часу розробники приходять до нього що б вирішити певні задача та завдання за допомогою нового фреймворку і імплементувати в старий сорскод. Цей перехід також може бути використаним в тому випадку, коли команда розробників ставить за ціль поетапного переведення сорскоду моноліту на інший фреймворк/бібліотеку і не може зупиняти функціонал продукту в цілому. Тоді в такий бібліотеці починають писати новий функціонал, а також при наявності часу переводять старий функціонал застосунку в сорс код цієї бібліотеки. На певному етапі більшість коду вже знаходиться в іншому застосунку, після чого туди дописується те що залишилось, видаляється файл бібліотеки і вона розгортається як окремий продукт.

Мікрофронтед за допомогою технології iFrame. iFrame або плаваючий фрейм – окреме вікно, HTML-документ, який відображається разом з іншим вмістом сторінки у вікні браузера [16]. iFrame досить стара технологія

розробки і використовувалась вона з початку для того щоб додати в сайт інший ресурс (наприклад відео плеєр чи щось інше) і по суті ця технологія відкриває у браузері вікно іншого браузера де показує певний контент. Контент який викликається в середині iFrame не контролюється зі сторони сорскоду основного продукту. Для того, щоб налагодити певну комунікацію між продуктом і iFrame, як варіант, можна використовувати метод об’єкту window (основний об'єкт базового середовища виконання javaScript коду, який має в собі певний набір методів, що і можуть бути використані при вирішенні задач розробки [17]).

Метод postMessage дозволяє забезпечити перехресний зв’язок між об’єктами Window; наприклад, між сторінкою та спливаючим вікном, яке вона створила, або між сторінкою та iframe, у якому вона вбудована. Зазвичай сценарії на різних сайтах мають доступ один до одного тоді й лише тоді, коли сайти, з яких вони походять, мають однаковий протокол, номер порту та хост (також відомий як «політика однакового походження»). window.postMessage() забезпечує контрольований механізм для безпечного обходу цього обмеження (якщо використовується правильно). Загалом, одне вікно може отримати посилання на інше (наприклад, через targetWindow = window.opener), а потім опублікувати в ньому MessageEvent за допомогою targetWindow.postMessage(). Потім вікно отримання може обробити цю подію за потреби. Аргументи, передані у window.postMessage() (тобто «повідомлення»), надаються вікну прийому через об’єкт події [18].

Серед плюсів iFrame підходу для реалізації мікрофронтенд архітектури ми можемо виокремити повну ізоляцію функціоналу і стилей одного додатку від іншого. Серед мінусів можна виокремити складність підтримки і швидкість цього підходу. Потрібно розуміти, що підхід iFrame був одним з перших в спробах розробників перейти до мікрофронтенду і наразі є більш зручні та безпечні методики реалізації мікрофронтенд архітектури.

Мікрофронтенд за допомогою бібліотеки Tailor.js. Tailor – це служба макетування, яка використовує потоки для створення веб-сторінки зі служб

фрагментів [19]. Деякі функції та переваги Tailor [19]:

– створює попередньо відтворену розмітку на сервері, що важливо для SEO та прискорює початковий рендер;

– забезпечує швидкий час до першого байта, при цьому Tailor запитує фрагменти паралельно та транслює їх якомога швидше, не блокуючи решту сторінки;

– виконує виконання бюджету. В іншому випадку це досить складно, оскільки немає єдиної точки, де можна контролювати продуктивність;

– відмовостійкість, відображаючи значущий вихід, навіть якщо фрагмент сторінки вийшов з ладу або минув час очікування.

Як ми бачимо з наведеного вище то Tailor.js це пакет для Node.js вона необхідна для того щоб побудувати мікрофронтенд архітектуру основану на SSR підході (SSR (Server-Side Rendering) генерує повний HTML для сторінки на сервері у відповідь на запит (перехід по посиланню). Це дозволяє уникнути додаткових запитів даних, наповнення шаблонів на стороні клієнта, оскільки вони обробляються до того, як браузер отримує відповідь [20]. У Tailor.js пишемо макети, які визначають, який Micro Frontend слід викликати в кожному представленні. Приблизно це може виглядати так [21]:

```
<!doctype html>
<html lang="en">
  <head></head>
  <h1>A website</h1>
  <body>
    <fragment slot="content" src="http://app1.micro-frontends.com"></fragment>
    <fragment slot="content" src="http://app2.micro-frontends.com"></fragment>
  </body>
</html>
```

На мою думку основним як мінусом так і плюсом є SSR підхід, оскільки цей підхід доволі специфічний для реалізації фронтенд додатків і може не підійти для конкретного проекту. Також у контексті переходу від монолітної

архітектури до мікросервісної користуватись підходом Tailor.js буде досить складно оскільки потрібно буде переписати додаток з фреймворку на NodeJs, через це можуть початись проблеми з деякими функціями і бібліотеками які були використані в базовому сорс коді моноліту і тепер можуть бути недоступні в NodeJs, також, слід згадати, що NodeJs хоча і використовує JavaScript більше пристосована для написання бекенд коду, через що частина команди розробників, яка працювала над монолітом може не вміти використовувати NodeJs. На відміну від підходу iFrame, який розглядався раніше, оскільки iFrame не потребує переписування всього продукту на інший фреймворк і більшість з програмістів frontend хоча б колись працювали з ним.

Мікрофронтенд за допомогою Single-spa. Single-spa – це фреймворк, який об’єднує кілька мікрофронтендів Javascript в одному фронтенді додатку [22]. Отже, як ми бачимо наразі для переходу на мікрофронтенд ми можемо використати фреймворк Single-spa, для коректного переходу нам потрібно лише розподілити монолітний продукт на бажанні частини, переписати частини в окремі фреймворки або бібліотеки налаштувати зв’язки і задеплоїти. Поряд з попередніми варіантами Single-spa має досить конкретні переваги адже дозволяє використовувати різні фреймворки або бібліотеки для написання (React, AngularJS, Angular, Ember або що зручно), дозволяє розвертати мікрофронтенди незалежно один від одного і за необхідністю дозволяє не змінювати попередній код моноліту, також підтримує описану вище ліниве завантаження. Також цей фреймворк повністю покриває проблеми розробників в разі “непередбачуваного росту”, це питання постає у випадку, коли моноліт при реалізації проектувався за допомогою конкретного фреймворку і був повністю розроблений за допомогою цього фреймворку, при реалізації не закладались можливості на додатковий функціонал і розробка завершилась, через певний час у замовника виникає бажання додати додатковий функціонал до вже існуючого продукту але фреймворк на якому було написано продукт більше не використовується, в такому випадку рішенням може бути використання нового фреймворку і нового підходу до написання і після цього

об’єднання за допомогою Single-spa. Після цього продукт буде мати таку структуру, як представлено на рисунку 2.2.

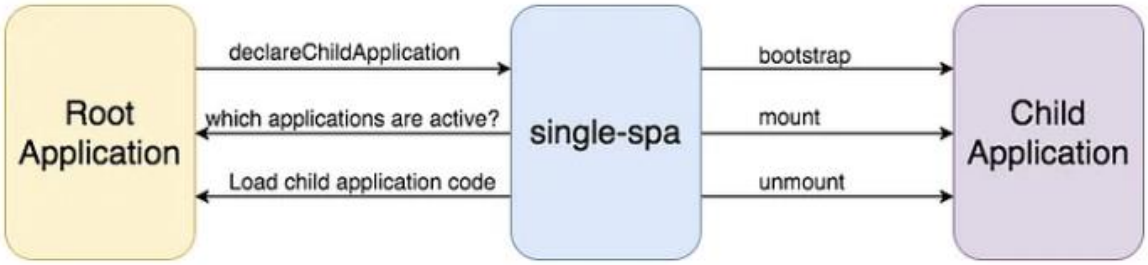

Рисунок 2.2 – Структура Single-spa

З огляду на вище вказане можна зробити висновок, що інтеграція Single-spa буде найбільш прийнятним підходом для переходу від моноліту до мікрофронтенду.

Мікрофронтенд за допомогою Nx, де Nx –це певний набір інструментів який дозволяє як писати мікрофронтендовий продукт з самого початку так і дозволяє перехід моноліту на мікрофронтед [23]. Модульність коду, яку пропонує nx, покращує повторне використання коду, а також дозволяє різним мікрофронтам повторно використовувати повторюваний код, також nx дозволяє окреме розгортання своїх компонентів. Виділяють наступні типи проектів Nx:

а) на основі пакетів, в якому кожен проект у робочій області є окремим незалежним пакетом. Кожен пакет має власний набір залежностей, скриптів. Вони публікуються окремими пакетами;

б) інтеграція – це коли кілька проектів об’єднані в одну робочу область і взаємодіють один з одним. Проекти в робочій області можуть залежати один від одного, спільно використовувати код, ресурси та налаштування, а їхніми залежностями керують централізовано у файлі package.json усього сховища;

в) автономний – це коли проект, який може функціонувати та використовуватися незалежно від інших проектів або компонентів у системі. Такі проекти найчастіше створюються для випробування інструменту Nx, але мікроінтерфейсів немає.

Підхід Nx є досить новим і, на мою думку, більше підходить для випадків, коли мікрофронтен архітектура була передбачена в застосунку з самого початку, а не в випадку коли переписується моноліт, оскільки на етапі інтеграції сторонніх сервісів можуть виникнути певні труднощі.

2.4 Аналіз проблем та викликів при переході до мікросервісів

З самого початку потрібно зрозуміти чи раціонально буде переводити монолітну архітектуру на мікрофронтед. Існує ряд причин за яких перехід проекту на мікрофронтенд може лише зашкодити проекту в цілому:

а) не великий розмір проекту, тобто мікрофронтенд досить добре працює на великих проектах, якщо ж проект не виликого розміру то перехід на міккрофронтенд скоріш заплутає розробку продукту ніж її полегшить;

б) кадровий склад компанії, тобто монолітна архітектуру набагато легша і, як правило, з нею зможе впоратись програміст з буд-яким рівнем. Чого не можна сказати про мікросервісну архітектуру, більше того для гарної підтримки мікросервісної архітектури ще потрібні девопс-інженери для розгортання середовищ;

в) порушення системи модульності моноліту та не підтримка принципу Dependency Inversion Principle, тоді в такому випадку зв’язки в продукті побудовані таким чином, що виділити певну частину для переходу досить важко і можуть зашкодити певним частинам продукту.

Тому, перш ніж прийняти рішення про перехід на мікросервіси, організації повинні оцінити свою готовність до переходу. Важливо розуміти, чи має команда необхідні навички для роботи з новою архітектурою, як мікросервіси вписуються в існуючий стек технологій і чи готова компанія до змін у методах ведення бізнесу, які неминуче відбуваються протягом дня. за дрібні послуги. Оцінка цих факторів допоможе уникнути потенційних проблем у майбутньому.

До початку процесу архітектурної трансформації і переходу з моноліту

на мікрофронтенд потрібно виділити цілі трансформації і відповісти на питання для чого трансформувати додаток серед яких:

– моноліт став занадто великий і його важко підтримувати;

– потрібно перейти на більш новий фреймворк або бібліотеку;

– новий функціонал не може безперешкодно доданий в моноліт і може привести до втрати функціональності моноліту;

– важко знайти співробітників через застарілу кодову базу;

– потрібно використовувати частину моноліту як окрему структуру.

Відповідь на поточне питання допоможе зрозуміти, яким саме чином повинен бути розділений моноліт. Якщо відповіддю на питання буде “Новий функціонал не може безперешкодно доданий в моноліт і може привести до втрати функціональності моноліту”, то, в даному випадку, необхідно просто винести частину функціоналу, яка повинна бути додана, до кодової бази в окремий додаток і підключити за допомогою одного з підходів до імплементації, судячи від поточних завдань для частини, що підключається можна вибрати або цілком простий iFrame підхід або більш складний Single-spa. Якщо передумовою переходу стає “Важко знайти співробітників через застарілу кодову базу”, то окрему увагу потрібно приділити вибору фреймворкку/бібліотеки для додатку, провести вивчення та аналіз ринку для того, щоб зрозуміти, який саме фреймворк/бібліотеку обрати.

Після готовності до переходу потрібно вибрати стратегію переходу, а саме, за яким принципом буде переведений моноліт на мікросервіс.

Першим шляхом буде різка зміна архітектури, якщо в компанії є певний період часу на перехід до мікросервісної архітектури, то команда припиняє підтримку монолітної архітектури і повністю переключається на розробку мікрофронтенду. Частково код може братись з моноліту чи з моноліту береться тільки структура окремих модулів, а інша частина просто переписується. Якщо реалізується цей підхід, то розробка моноліту повинна бути повністю зупинена, оскільки інакше може з’явитись перспектива циклічної розробки.

Принцип модульного переходу (раніше розглядали модульність системи в моноліті). Для модульного переходу на мікрофронтенд беруть певний модуль моноліту, відключають його і переписують в інший додаток. Після чого розгортають його і за допомогою вибраного підходу до організації мікрофронтенду імплементують назад в моноліт але вже як окремий сервіс.

При виборі будь-якого з підходів для переходу на мікросервіс окрему увагу потрібно звернути на технічну частину і вибір технічного рішення як для продукту, так і для частин мікрофронтенду. Так, не дуже бажано:

– змішувати підходи, такі як SSR з рендерінгомна клієнті, тому що, перехід одної з частин на SSR без переходу головної, як правило, можливий але це досить різні підходи до написання веб архітектури і, якщо їх змішувати то, можуть виникнути проблеми з підтримкою коду та функціоналу;

– не змішувати фреймворки, так як, взагалі хоч і можливо використовувати і більше ніж один фреймворк при мікроофронтенді, але тут потрібно пам’ятати, що фреймворк важчий за бібліотеку оскільки з коробки має всі функції які потрібні (роутінг, інтерсептори і таке інше) при цьому не завжди ці функції необхідні в структурі одної з частин мікрофронтенду;

– не змішувати стилі написання коду, так як, для того, що б робота з мікрофронтенда була швидша і ефективніша, не потрібно реалізовувати зміни в стилі коду і мові програмування, оскільки для роботи з кожною частиною мікрофронтенду потрібно буде мати час на адаптацію.

Незалежно від того який підхоід до реалізації переходу був вибраний можливим варіанттом також буде для кожної з части мікрофронтенду створити окрему базу даних, в такому випадку частини стануть ще більш незалежними.

2.5 Аналіз проблем та викликів при переході від моноліту до мікро-фронтенду

При переході від моноліту до мікрофронтенду одною з основних проблем може бути версіонність, оскільки в момент переходу додатку ми по суті

вирізаємо сорс код з одного додатку і додаємо в інший. При цьому, як правило, версії бібліотек які використовувались в монолітному додатку не будуть співпадати з версіями нового фреймфорку, що призведе до переписування сорс коду, який переноситься.

За великим рахунком, способів подолання цієї проблеми існує два. В першому варіанті можливо розгорнути фреймворк тої самої версії, що і фреймворк моноліту, в такому випадку перехід пройде досить безпроблемно і не забере багато часу, однак, цей підхід є не дуже гарним, оскільки це не принесе користі застосунку в цілому. Як правило перехід до моноліту планується для покращення кодової бази і підвищення продуктивності, якщо додавати фреймворк старої версії, то вийде, що замість прогресу у розробці ми приходимо до регресу, оскільки суттєвих змін, окрім окремого розгортання ми не отримуємо. Більш того, продукт з підключеним мікросервісом буде займати сумарно більше місця на сервері, оскілки ми повинні будемо зібрати розміри двох бандлів і бібліотек, а в цьому разі в нас виходить, що ми маємо два фреймворки плюс до того подвійний набір бібліотек для кожного з додатків.

Другим варіантом є використання більш нових версій існуючих бібліотек і виправлення існуючого коду (рефакторинг коду) під них. Цей варіант є більш важким і потребує додатковий час на розробку, але разом з мінусами попереднього варіанту (сукупним розміром бандлу) ми будемо мати бонус, у вигляді більш читабельного коду, який легше підтримувати в майбутньому, а також можливості використання повних функцій фреймворку, які з’являються в його більш пізніх версіях.

Іншою досить вагомою проблемою, можемо назвати управління станом продукту. В більшість монолітів при розробці включають загальний стан для всього продукту (state), особливо часто це зустрічається при використанні архітектурного патерна FLUX [24]. Цей архітектурний патерн, який передбачає кругову залежність, в якій відображення залежить від даних, а дані від відображення (див.рис.2.3). За цим патерном дані зберігаються в сторі (певному загальному стані застосунку) і напряму передаються на відображення, в разі

якоїсь зміни включається action, що запускає dispatcher, який, в свою чергу, змінить дані в середині стори.

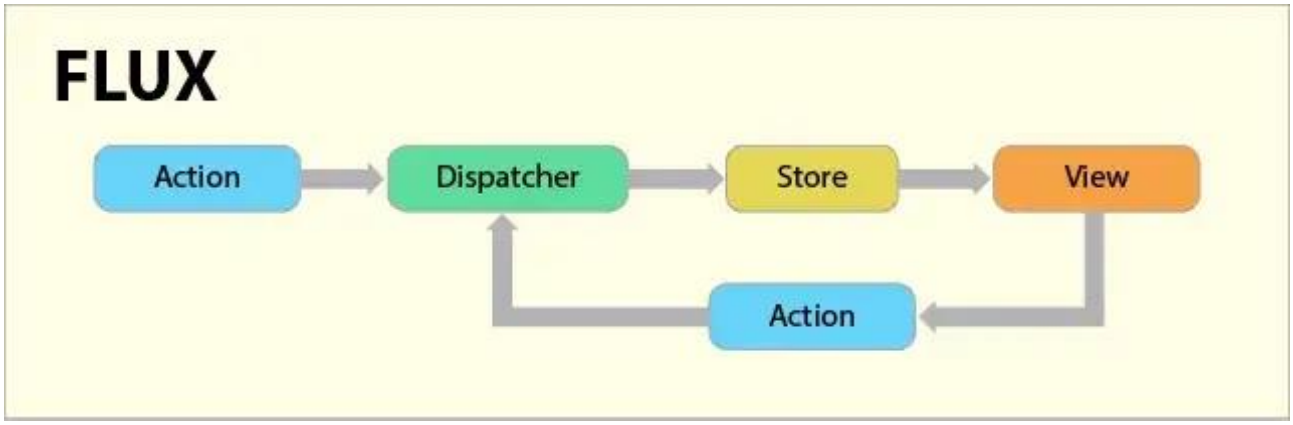

Рисунок 2.3 – FLUX підхід

Використання цього підходу дуже часте для рішень в монолітних архітектурах, через те, що всі дані завжди доступні по всьому продукту і в будь-який час ми можемо звернутись до стори за певними даними. При цьому при переході на мікрофронтенд може виникнути певна кількість питань, оскільки ми втрачаємо підключення до стори. В такому разі, найкращим варіантом буде розподілення навантаження між загальним станом та бекендом і базою. Частину даних краще буде перенести в базу і, за необхідності постійного оновлення, імплементувати на стороні бекенд технологію WebSocket (це протокол, що призначений для обміну інформацією між браузером та вебсервером в режимі реального часу). Він забезпечує двонаправлений повнодуплексний канал зв'язку через один TCP-сокет. WebSocket спроектовано для втілення у веббраузерах та вебсерверах, але може також використовуватись будь-яким клієнт-серверним застосунком. Прикладний програмний інтерфейс WebSocket був стандартизований W3C. Окрім того, протокол WebSocket стандартизований IETF як RFC [25]. Після чого мікрофронтенд підключається до бекенду і буде отримувати необхідні дані вже напряму з бази.

Швидкість розгортання та масштабування – це наступна проблема, яка може виникнути при переході, адже при використанні моноліта необхідно

розгорнути лише один додаток, при використанні ж мікрофронтенд підходу з окремими додатками ми повинні розуміти, що розгортати доведеться кожний додаток окремо і стежити за його версійністю.

За великим рахунком, проблема розгортання додатку вирішується шляхом імплементування додаткових DevOps інструментів таких як Docker (Docker – це програмне забезпечення з відкритим кодом, найпопулярніша платформа для управління контейнерами. Він потрібен для більш ефективного використання системи і ресурсів, швидкого розгортання готових програмних продуктів, а також для їх масштабування і перенесення в інші середовища з гарантованим збереженням стабільної роботи [26]) та Jenkins (система з відкритим вихідним кодом, тобто продукт доступний для перегляду, вивчення та зміни, який створений на базі Java). Дженкінс дозволяє автоматизувати частину процесу розробки програмного забезпечення без участі людини. Ця система призначена для забезпечення процесу безперервної інтеграції програмного забезпечення [27]. Ці інструменти повністю закривають проблеми розгортання і налаштовуються один раз, після цього розгортання можна проводити лише за допомогою команди в корпоративному месенджері такому як, Slack, більше того, за допомогою тегів, які були створені в процесі розгортання, ми можемо продивитись історію змін кожної частини і в разі необхідності відкотити версію певної мікрофронтенд частини (як приклад, в разі, якщо була знайдена нестабільність в роботі певного функціоналу).

Складність інтеграції раніше ця проблема виникала частіше ніж зараз, оскільки мікрофронтенд був зовсім новою архітектурою застосунків і було досить важко інтегрувати мікрофронтенд сервіс в “батьківську” частину застосунку. Наразі є багато шляхів для вирішення цього питання, починаючи з HTML підходу до створення мікрофронтенду за допомогою iFrame і закінчуючи готовими рішеннями, такими як бібліотека Single SPA, за допомогою документації якої досить швидко реалізувати перехід.

2.6 Аналіз критеріїв оцінки результатів трансформації

Для оцінки результатів переходу від моноліту до мікрофронтенду можна застосувати порівняння між системою моноліту і отриманим в результаті трансформації продукту, до якого входять мікрофронтеди. Серед основних характеристик, на які можна звернути увагу, може бути розмір бандлу, швидкість деплою, швидкість рендеру окремих компонентів продукту.

При цьому основною ціллю переходу на мікрофронтенд є стабільність застосунку і можливість його підтримки і розширення, за допомогою стандартних способів оцінити це результат неможливо, однак можливо оцінити якість переходу можна через певний час (місяць - два) за допомогою метрик в Jira.

Перш за все варто приділити увагу статистиці відкритих багів, при позитивному сценарії переходу статистика повинна почати падати, оскільки структурно тепер частини продукту не так сильно пов’язані між собою. Другою метрикою, на яку варто звернути увагу, буде “наповненість” команди по сторі поінтам (сторі поінти це одиниці виміру, які показують складність певної задачі). Для сторіпоінтів використовують будь що, наприклад, числа Фібоначі, чи певних звірів, наприклад, собак, чи розміри в футболках. Основна ідея сторіпоінтів полягає в тому, що б розділити задачі на дошці від малих до великих. Якщо перехід був успішним, то “наповненість” команди почне зростати. Третьою метрикою може бути кількість деплої окремих застосунків, в разі якщо деякі мікрофронтенд деплоять частіше ніж інші. Можемо сказати, що компанія позитивно економить кошти на процес розгортання, оскільки раніше розгортався повний моноліт, а після переходу розгортаються окремі його частини які менші за розміром.

3 РЕАЛІЗАЦІЯ ПЕРЕХОДУ ВІД МОНОЛІТУ ДО МІКРО-ФРОНТЕНДУ

3.1 Аналіз архітектурних підходів Angular додатку

Для реалізації тестового додатку було вибрано фреймфорк Angular версії 9, для найкращого ілюстрування змін при переході від моноліту до мікро-фронтенду. Сам фреймворк на відміну від бібліотеки має готову структуру з усіма необхідними засобами (такими як роутінг, форми і так далі) для розробки. Ця структура надає як плюси при розробці додатків так і мінуси, оскільки з одного боку у розробника є всі необхідні інструменти, для того що б написати додаток, а з іншого боку розробник може не використовувати частину інструментів, а вона буде включена в фінальну збірку додатку.

При розробці додатку за допомогою Angular є дві основні структури програми.

Перша це одномодульна структура застосунку (див.рис.3.1), полягає в тому, що додаток Angular складається з одного модулю в який підключено дерево компонентів які вкладені один в інший.

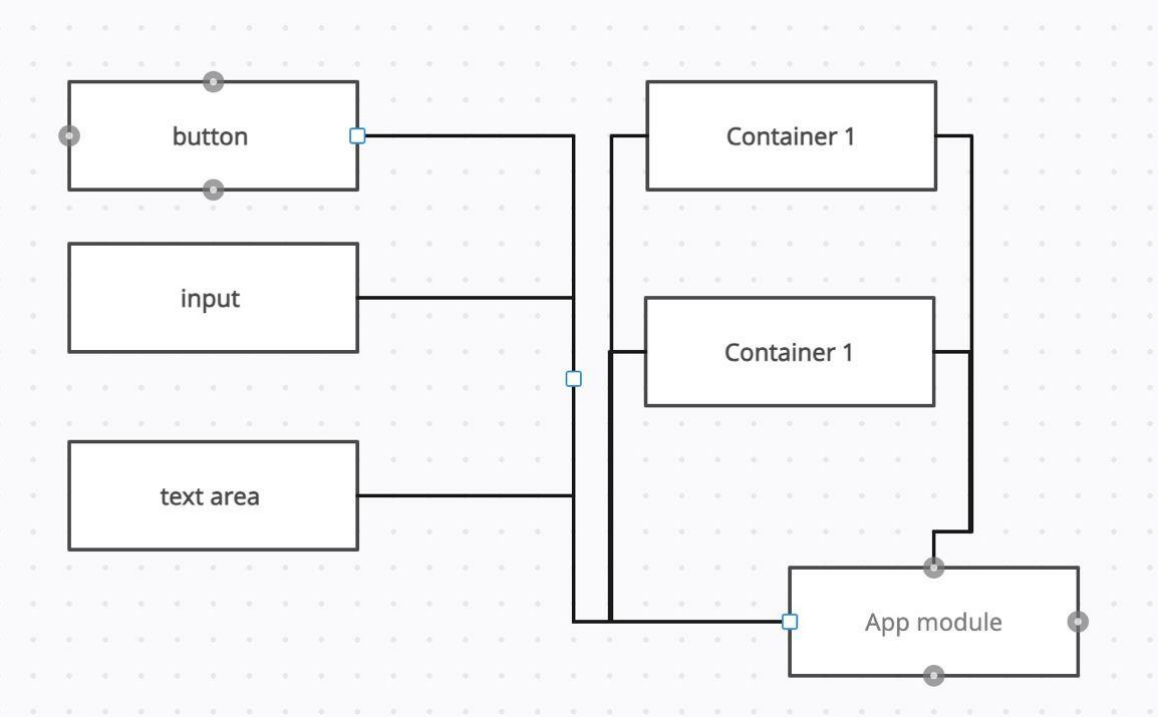

Рисунок 3.1 – Одномодульна структура Angular

Умовно компоненти можуть бути поділені на контейнери – компоненти які знаходяться на верхній частині ієрархії і використовують інші; та компоненти що перевикористовуються – це компоненти, котрі використовуються як візуальні чи функціональні блоки застосунку які роблять конкретну функцію і перевикористовуються в інших його частинах де є потреба в певній функціональності для якої вони призначені, суть цих компоненті полягає в тому, що якщо потрібно зробити певну зміну в візуальному оформлені чи функціоналі якогось блоку (наприклад змінити колір грані у інпута) ця зміна робиться в одному компоненті і за рахунок його перевикористання буде автоматично змінено по всьому додатку, без необхідності змінювати якусь конкретну частину (див.рис.3.1).

Друга структура це багатомодульна структура (див.рис.3.2), яка полягає в тому, що компоненти контейнери видокремлюються в окремі модулі застосунку і викликаються за допомогою роутінгу (конструкція lazy loading).

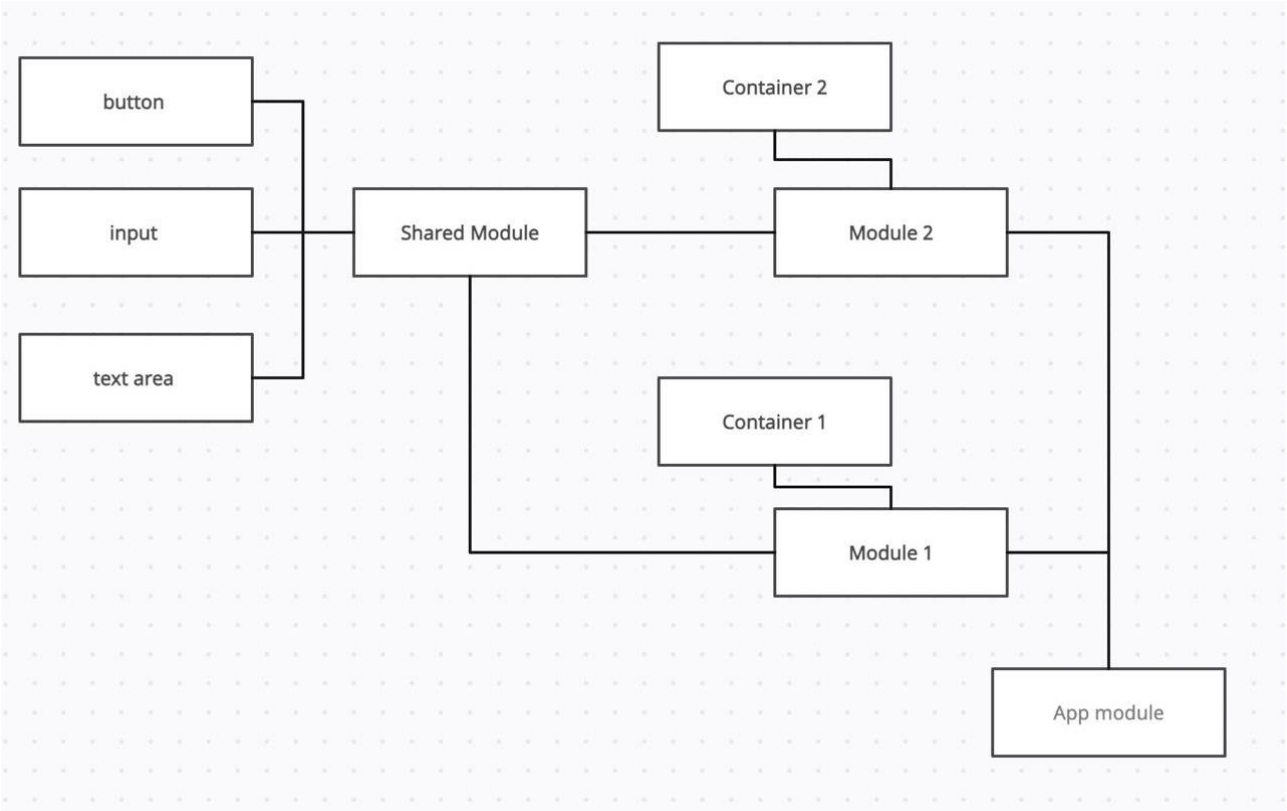

Рисунок 3.2 – Багатомодульна структура Angular

При цьому компоненти які перевикористовуються підключаються в

Shared module цей модуль в свою чергу буде підключений до модулів контейнерів. Така структура компонентів і модулів необхідна з декількох причин, таких як незалежність модулів (зміни в одному модулі не повинні змінювати інший модуль, а також при такій структурі файл скриптів і розмітки застосунку буде завантажуватись тільки в той момент коли користувач перейде до конкретного роуту, в першій же структурі файл буде завантажуватись одразу на весь застосунок. Для великих проектів частіше використовується багатомодульна структура застосунку, оскільки при такій структурі застосунок легше підтримувати і він працює на багато швидше, оскільки не тримає в оперативній пам’яті додаткові дані які не потрібні наразі користувачу (див.рис.3.2).

Для візуалізації переходу від моноліту до мікро-фронтенду спершу розглянемо перехід від одномодульної архітектури до багатомодульної.

3.2 Реалізація одномодульної архітектури додатку Angular

Для візуалізації напишемо тестовий додаток пошуку роботи на Angular який буде складатись з сторінок: дошка вакансій, сторінка компанії, сторінка кандидата.

Командою ng new < ім’я застосунку > ми ініціалізуємо створення додатку, після цього вибираємо за допомогою чого ми будемо писати стилі, в цьому випадку це css. Далі питання, чи потрібен нам ssr відповідаємо що ні. Після цього Angular cli побудує необхідну стартову структуру для застосунку

```
├── package-lock.json
    ├── package.json
    ├── src
    │   ├── app
    │   │   ├── app.component.css
    │   │   ├── app.component.html
    │   │   ├── app.component.spec.ts
```

```
│   │   ├── app.component.ts
│   │   ├── app.config.ts
│   │   └── app.routes.ts
│   ├── assets
│   ├── favicon.ico
│   ├── index.html
│   ├── main.ts
│   └── styles.css
├── tsconfig.app.json
├── tsconfig.json
└── tsconfig.spec.json
```

Стартова структура складається з файлу залежностей і скриптів package.json в якому вказані команди запуску застосунку, директорії src в який лежить index.html файл, який являє собою точку входу в застосунок і директорія app в якій будуть лежати файли застосунку.

Для початку роботи над застосунком нам потрібно розширити директорію app. А саме ми додамо директорію _pages (в цій директорії будуть розміщенні компоненти контейнери які будуть сторінками застосунку), директорію components (в цій директорії розміщаються компоненти які будуть перевикористовуватись в контейнерах) і директорія Interfaces (в цій директорії будуть розміщуватись опис даних).

Заповнимо директорію components такими компонентами:

1) input

- input.component.ts

-- import {Component, EventEmitter, Input, OnInit, Output} from '@Angular/core';

--

-- @Component({

-- selector: 'app-input',

-- templateUrl: './input.component.html',

```
  --   styleUrls: ['./input.component.css']
  -- })
  -- export class InputComponent implements OnInit {
  --   @Input() header
  --   @Input() value
  --   @Input() placeholder
  --   @Output() valueChange = new EventEmitter();
  --   constructor() {
  --   }
  --
  --   ngOnInit(): void {
  --   }
  --
  --   valueChanged() {
  --     this.valueChange.emit(this.value);
  --   }
  -- }
- input.component.css
  -- .input-header {
  --   font-family: "Roboto", sans-serif;
  --   color: var(--input-header-color);
  --   font-size: 14px;
  --   font-weight: 400;
  --   line-height: 20px;
  --   text-align: left;
  -- }
  --
  -- .input-container input {
  --   width: calc(100% - 8px);
  --   margin-top: 8px;
```

-- padding: 4px;
-- outline: none;
-- background-color: var(--background-color);
-- border: 1px solid var(--border-color);
-- border-radius: var(--border-radius);
-- }
--
-- .input-container input:focus {
-- border-color: var(--focus-border-color);
-- background-color: var(--focus-background-color);
-- }
--
-- .input-container input.filled {
-- background-color: var(--focus-background-color);
-- }

- input.component.html

--
--
-- {{header}}
--
-- <input type="text" [(ngModel)]="value" (input)="valueChanged()" [placeholder]="placeholder">
--

2) vacancies-list

- vacancies-list.component.css

-- .table {
-- display: grid;
-- grid-gap: 8px;
-- grid-template-columns: 1fr 1fr 1fr;
-- margin-top: 24px;

```
--   margin-bottom: 8px;
-- }
--
-- .table .list-item {
--   display: flex;
--   flex-direction: column;
--   justify-content: space-between;
--   height: 200px;
--   background-color: white;
--   border-radius: 8px;
--   padding: 12px;
--   border: 1px solid #E2E5E7;
--   box-shadow: rgba(100, 100, 111, 0.2) 0px 7px 29px 0px;
-- }
--
-- .table .list-item:hover {
--   background-color: #3A66E5;
-- }
-- .table .list-item:hover .name,
-- .table .list-item:hover .description,
-- .table .list-item:hover .stack {
--   color: white;
-- }
--
-- .name {
--   padding-bottom: 8px;
--   border-bottom: 1px solid #CECECE;
--   font-weight: 600;
--   font-size: 16px;
-- }
```

```
--
-- .description {
--   padding-bottom: 8px;
--   padding-top: 8px;
--   font-style: italic;
--   font-size: 14px;
-- }
--
-- .stack {
--   padding-bottom: 8px;
--   margin-top: 24px;
--   border-bottom: 1px solid #CECECE;
--   font-weight: 600;
--   font-size: 16px;
--   font-style: italic;
-- }
--
-- .cost {
--   font-weight: 600;
--   font-size: 16px;
--   color: red
-- }
```

- vacancies-list.component.html

```
-- <div class="table">
--   <div class="list-item" *ngFor="let vacancv of vacancies">
--     <div class="main-block">
--       <div class="name"> Vacancy Name: {{vacancv.name}}</div>
--       <div class="description">{{vacancv.decription}}</div>
--       <div class="stack">stack: {{vacancv.stack}}</div>
--       <div class="stack">Experience: {{vacancv.yearsOfExpirience}}</div>
```

--

-- Cost: ${{vacancv.yearsOfExpirience}}

--

--

- vacancies-list.component.ts

-- import {Component, Input, OnInit} from '@Angular/core';

--

-- @Component({

-- selector: 'app-vacancies-list',

-- templateUrl: './vacancies-list.component.html',

-- styleUrls: ['./vacancies-list.component.css']

-- })

-- export class VacanciesListComponent implements OnInit {

-- @Input() vacancies

-- constructor() { }

--

-- ngOnInit(): void {

-- }

--

-- }

3) button

- button.component.ts

-- import {Component, EventEmitter, Input, OnInit, Output} from '@Angular/core';

--

-- @Component({

-- selector: 'app-button',

-- templateUrl: './button.component.html',

-- styleUrls: ['./button.component.css']

-- })

-- export class ButtonComponent implements OnInit {
-- @Input() name: string
-- @Output() onClickCallback = new EventEmitter();
--
-- constructor() { }
--
-- ngOnInit(): void {
-- }
--
-- onClickCallbacked() {
-- this.onClickCallback.emit();
-- }
--
-- }

- button.component.css

-- .button {
-- border: 1px solid #3A66E5;
-- background-color: #FFFFFF;
-- border-radius: 6px;
-- color: #3A66E5;
-- font-weight: 400;
-- font-size: 14px;
-- width: 100%;
-- padding: 8px;
-- cursor: pointer;
-- }
--
-- .button:hover {
-- color: white;
-- background-color: #3A66E5;

-- border: 1px solid #FFFFFF;

-- }

- button.component.html

-- <button class="button" (click)="onClickCallbacked()">{{name}}</button>

.//table

- table.component.html

-- <div *ngIf="tableData" class="table">

-- <div class="list-item" *ngFor="let item of tableData"
(click)="itemCallBack(item.id)">

-- <img *ngIf="tableData" class="avatar" [src]="item.img" alt="pic">

-- {{item.name}}

--

-- <ng-content></ng-content>

--

4) table

- table.component.ts

-- import {Component, Input} from '@Angular/core';

-- import {TableData} from 'src/app/Interfaces/TableData';

--

-- @Component({

-- selector: 'app-table',

-- templateUrl: './table.component.html',

-- styleUrls: ['./table.component.css']

-- })

--

--

-- export class TableComponent {

-- @Input() tableData: TableData[]

-- @Input() itemCallBack: (companyId: number) => void

-- }

- table.component.css

```
-- .table  {
--   width: calc(100% - 48px);
--   padding: 24px;
--   border-radius: 8px;
--   background-color: #FAFAFA;
-- }
--
-- .table .list-item {
--   display: flex;
--   align-items: center;
--   background-color: white;
--   border-radius: 8px;
--   padding: 12px;
--   border: 1px solid #E2E5E7;
--   box-shadow: rgba(100, 100, 111, 0.2) 0 7px 29px 0;
-- }
--
-- .table .list-item:hover {
--   background-color: #3A66E5;
--   color: white;
-- }
--
-- .table .list-item:hover .name {
--   color: white;
-- }
--
-- .table .list-item:not(:first-child) {
--   margin-top: 16px;
-- }
```

--
-- .avatar {
-- width: 40px;
-- height: 40px;
-- border-radius: 50%;
-- margin-right: 24px;
-- }
--
-- .name {
-- font-weight: 500;
-- font-size: 16px;
-- color: #1c232b;
-- }

5) textarea-block

- textarea-block.component.html

--
--
-- {{header}}
--
-- <textarea [(ngModel)]="value" (input)="valueChanged()" [placeholder]="placeholder"></textarea>
--

- textarea-block.component.ts

-- import { Component, EventEmitter, Input, OnInit, Output } from '@Angular/core';
--
-- @Component({
-- selector: 'app-textarea-block',
-- templateUrl: './textarea-block.component.html',
-- styleUrls: ['./textarea-block.component.css']

```
  -- })
  -- export class TextareaBlockComponent implements OnInit {
  --   @Input() header
  --   @Input() value
  --   @Input() placeholder
  --   @Output() valueChange = new EventEmitter();
  --   constructor() {
  --   }
  --
  --   ngOnInit(): void {
  --   }
  --
  --   valueChanged() {
  --     this.valueChange.emit(this.value);
  --   }
  -- }
- textarea-block.component.css
  -- .textarea-header {
  --   font-family: "Roboto", sans-serif;
  --   color: var(--textarea-header-color);
  --   font-size: 14px;
  --   font-weight: 400;
  --   line-height: 20px;
  --   text-align: left;
  -- }
  --
  -- .textarea-container textarea {
  --   width: calc(100% - 8px);
  --   margin-top: 8px;
  --   padding: 4px;
```

-- outline: none;
-- background-color: var(--background-color);
-- border: 1px solid var(--border-color);
-- border-radius: var(--border-radius);
-- resize: none;
-- }
--
-- .textarea-container textarea:focus {
-- border-color: var(--focus-border-color);
-- background-color: var(--focus-background-color);
-- }
--
-- .textarea-container textarea.filled {
-- background-color: var(--focus-background-color);
-- }

6) avatar

- avatar.component.ts

-- import { Component, Input, OnInit } from '@Angular/core';
--
-- @Component({
-- selector: 'app-avatar',
-- templateUrl: './avatar.component.html',
-- styleUrls: ['./avatar.component.css']
-- })
-- export class AvatarComponent implements OnInit {
-- @Input() url
--
-- constructor() { }
--
-- ngOnInit(): void {

-- }
--
-- }
- avatar.component.css
-- .personal-block-avatar {
-- display: block;
-- box-sizing: border-box;
-- height: 128px;
-- width: 124px;
-- border: 1px solid var(--border-color);
-- }
- avatar.component.html
-- <img
-- class="personal-block-avatar"
-- [src]="url"
-- alt="avatar"
-- />
7) block
- block.component.ts
-- import { Component, Input, OnInit } from '@Angular/core';
--
-- @Component({
-- selector: 'app-block',
-- templateUrl: './block.component.html',
-- styleUrls: ['./block.component.css']
-- })
-- export class BlockComponent implements OnInit {
-- @Input() header
--
-- constructor() { }

```
  --
  --   ngOnInit(): void {
  --     console.log(this.header);
  --   }
  --
  -- }
- block.component.html
  -- <div class="block-container">
  --   <div class="block-header">
  --     {{header}}
  --   </div>
  --   <div class="content-container">
  --     <ng-content></ng-content>
  --   </div>
  -- </div>
- block.component.css
  -- .block-container {
  --   padding: var(--block-padding);
  --   background-color: white;
  --   box-shadow: var(--block-shadow);
  --   border-radius: var(--border-radius);
  -- }
  --
  -- .block-header {
  --   font-family: "Roboto", sans-serif;
  --   font-size: 32px;
  --   font-weight: 500;
  --   line-height: 38px;
  -- }
  --
```

-- .content-container {
-- margin-top: 24px;
-- }

Тепер заповнимо директорію _pages контейнерами

1) find-work-list

- find-work-list.component.ts

-- import { Component, OnInit } from '@Angular/core';
-- import {FindWorkListService} from 'src/app/_pages/find-work-list/find-work-list.service';
--
-- @Component({
-- selector: 'app-find-work-list',
-- templateUrl: './find-work-list.component.html',
-- styleUrls: ['./find-work-list.component.css']
-- })
-- export class FindWorkListComponent implements OnInit {
-- data: any = []
-- constructor(private findWorkListService: FindWorkListService) { }
--
-- ngOnInit(): void {
-- console.log(1);
-- this.findWorkListService.getVacancies().subscribe((data) => {
-- this.data = data;
-- })
-- }
--
-- }

- find-work-list.component.html

--
-- <app-vacancies-list [vacancies]="data"></app-vacancies-list>

--

- find-work-list.service.ts

-- import { Injectable } from '@Angular/core';

-- import {HttpClient} from '@Angular/common/http';

--

-- @Injectable({

-- providedIn: 'root'

-- })

-- export class FindWorkListService {

-- public API_URL: string = 'http://localhost:5000/api/v1/'

--

-- constructor(private http: HttpClient) { }

--

-- getVacancies():any {

-- return this.http.get(this.API_URL + 'vacancies/query?limit=all')

-- }

-- }

- find-work-list.component.css

2) employee/employee-page

- employee-page.component.html

-- <app-block header="Candidate">

--

-- <app-avatar url="url"></app-avatar>

--

--

-- <app-input header="Name" [value]="employeeData.name" placeholder="Email" (valueChange)="valueChange($event, 'employeeEmail')"></app-input>

--

--

-- <app-input header="Second name" [value]="employeeData.sacondName" placeholder="Email" (valueChange)="valueChange($event, 'employeeEmail')"></app-input>
--
--
-- <app-input header="Email" [value]="employeeData.employeeEmail" placeholder="Email" (valueChange)="valueChange($event, 'employeeEmail')"></app-input>
--
--
-- <app-input header="Phone" [value]="employeeData.employeePhone" placeholder="Phone" (valueChange)="valueChange($event, 'employeePhone')"></app-input>
--
--
--
-- </app-block>

- employee-page.component.ts

-- import { Component, OnInit } from '@Angular/core';
--
-- @Component({
-- selector: 'app-employee-page',
-- templateUrl: './employee-page.component.html',
-- styleUrls: ['./employee-page.component.css']
-- })
-- export class EmployeePageComponent implements OnInit {
-- employeeData: any = {
-- name: '',
-- secondName: '',
-- employeeEmail: '',

```
  --     employeePhone: '',
  --   }
  --
  --   constructor() { }
  --
  --   ngOnInit(): void {
  --   }
  --
  --   valueChange(value, field) {
  --     this.employeeData[field] = value
  --   }
  --
  -- }
- employee-page.component.css
  -- .employee-container {
  --   display: flex;
  --   align-items: center;
  --   width: 100%;
  -- }
  --
  -- .employee-container .item-block:not(:first-child) {
  --   margin-top: 8px;
  -- }
  --
  -- .inside-block {
  --   width: 100%;
  --   margin-left: 24px;
  --   box-shadow: 0 1px 8px 0 #1C232B26;
  --   padding: var(--block-padding);
  --   border-radius: var(--border-radius);
```

-- }

4) company

- company.component.html

-- <app-table [tableData]="companiesList"
[itemCallBack]="redirectToCompanyPage">

--

-- <app-button name="Add Company +"
(onClickCallback)="onAddCompany()"></app-button>

--

-- </app-table>

--

- company.service.ts

-- import { Injectable } from '@Angular/core';

-- import { HttpClient } from '@Angular/common/http';

-- import {Observable} from 'rxjs';

-- interface CompanyData {

--

-- companyName: string,

-- companyLogo: string

-- }

--

-- interface CompanyDataResponse {

-- company: CompanyData[]

-- }

--

--

-- @Injectable({

-- providedIn: 'root'

-- })

--

-- export class CompanyService {
-- public API_URL: string = 'http://localhost:5000/api/v1/'
--
-- constructor(private http: HttpClient) { }
--
-- getCompanies():any {
-- return this.http.get(this.API_URL + 'company/query?limit=all')
-- }
-- }

- company.component.css

-- .buttons-container {
-- margin-top: 16px;
-- }

- company.component.ts

-- import { Component, OnInit } from '@Angular/core';
-- import { CompanyService } from 'src/app/_pages/company/company.service';
-- import {TableData} from 'src/app/Interfaces/TableData';
-- import { Router } from '@Angular/router';
--
-- @Component({
-- selector: 'app-company',
-- templateUrl: './company.component.html',
-- styleUrls: ['./company.component.css']
-- })
-- export class CompanyComponent implements OnInit {
-- companiesList: TableData[]
--
-- constructor(private companyService: CompanyService,
-- private router: Router) { }

```
        --
        --   ngOnInit(): void {
        --     this.companyService.getCcompanies().subscribe(({ company }) => {
        --       this.companiesList = company.map(({companyName, companyLogo,
...rest}) => ({
        --         name: companyName,
        --         img: companyLogo,
        --         ...rest
        --       }))
        --     })
        --   }
        --
        --   onAddCompany = () => {
        --     console.log('works');
        --   }
        --
        --   redirectToCompanyPage(companyId) {
        --     window.location.href =
'http://localhost:4200/company/companyPage?companyId=' + companyId
        --   }
        -- }
       .//company/company-page
        - company-page.component.css
        -- .container {
        --   width: calc(100% - 24px);
        --   height: calc(100vh - 24px);
        --   margin: 12px;
        --   padding: 12px;
        --   background-color: #e0dfdf;
        --   border-radius: 4px;
```

-- }
--
-- .logo {
-- width: 140px;
-- height: 250px;
-- border-radius: 4px;
-- }
--
-- .information {
-- display: flex;
-- align-items: flex-start;
-- }
--
-- .company-data {
-- width: calc(100% - 48px);
-- padding-left: 24px;
-- padding-right: 24px;
-- }
--
-- .company-data .item-block:not(:first-child) {
-- margin-top: 8px;
-- }

5) company/company-page

- company-page.component.ts

-- import { Component, OnInit } from '@Angular/core';
-- import {ActivatedRoute} from '@Angular/router';
-- import {CompanyPageService} from 'src/app/_pages/company/company-page/company-page.service';
-- import {retry} from 'rxjs/operators';
--

```
-- @Component({
--   selector: 'app-company-page',
--   templateUrl: './company-page.component.html',
--   styleUrls: ['./company-page.component.css']
-- })
-- export class CompanyPageComponent implements OnInit {
--   companyData: any
--   companyDataValid: any
--   isVacancyModalOn: boolean = false
--
--   constructor(
--     private route: ActivatedRoute,
--     private companyPageService: CompanyPageService
--   ) {
--
--   }
--
--   ngOnInit(): void {
--     this.route.queryParams.subscribe((params) => {
--
this.companyPageService.getCompany(params.companyId).subscribe((data) => {
--
--         this.companyData = data
--         this.companyDataValid = {...data}
--       })
--     })
--   }
--
--   valueChange(value, field) {
--     this.companyData[field] = value
```

```
-- }
--
-- onUpdateCompany(value) {
--
this.companyPageService.updateCompany(this.companyData).subscribe((data) =>
console.log(data))
-- }
--
-- toggleVacancyModal() {
--   this.isVacancyModalOn = !this.isVacancyModalOn
-- }
--
-- onCreatVacancy(data) {
--   this.companyPageService.createVacancy({id: this.companyDataValid.id,
...data}).subscribe((data) => console.log(data))
-- }
--
--
-- }
```

- company-page.component.html

```
-- <app-block header="Company">
--  <div class="information">
--   <img class="logo" [src]="companyData.companyLogo" alt="img">
--   <div class="company-data">
--     <div class="item-block">
--       <app-input
--         header="Company name"
--         [value]="companyData.companyName"
--         placeholder="Name"
--         (valueChange)="valueChange($event, 'companyName')">
```

```
--         </app-input>
--       </div>
--       <div class="item-block">
--         <app-input header="Company email"
[value]="companyData.companyEmail" placeholder="Email"
(valueChange)="valueChange($event, 'companyEmail')"></app-input>
--       </div>
--       <div class="item-block">
--         <app-input header="Company phone"
[value]="companyData.companyPhone" placeholder="Phone"
(valueChange)="valueChange($event, 'companyPhone')"></app-input>
--       </div>
--       <div class="item-block">
--         <app-input header="HR name" [value]="companyData.hrName"
placeholder="hrName" (valueChange)="valueChange($event, 'hrName')"></app-
input>
--       </div>
--       <div class="item-block">
--         <app-input header="Company logo"
[value]="companyData.companyLogo" placeholder="hrName"
(valueChange)="valueChange($event, 'companyLogo')"></app-input>
--       </div>
--       <div class="item-block">
--         <app-button name="Update data"
(onClickCallback)="onUpdateCompany($event)"></app-button>
--       </div>
--     </div>
--   </div>
--   <app-vacancies-list [vacancies]="companyData.vacancies"></app-
vacancies-list>
```

```
        --   <app-vacancy-modal *ngIf="isVacancyModalOn"
(onCreatVacancy)="onCreatVacancy($event)"></app-vacancy-modal>
        --   <app-button name="Create Vacancy"
(click)="toggleVacancyModal()"></app-button>
        -- </app-block>
       .//company/company-page
        - company-page.service.ts
        -- import { Injectable } from '@Angular/core';
        -- import {HttpClient} from '@Angular/common/http';
        --
        -- @Injectable({
        --   providedIn: 'root'
        -- })
        -- export class CompanyPageService {
        --
        --   public API_URL: string = 'http://localhost:5000/api/v1/'
        --
        --   constructor(private http: HttpClient) { }
        --
        --   getCompany(companyId):any  {
        --     return this.http.get(this.API_URL + 'company/companyData', { params:
{ companyId } })
        --   }
        --
        --   updateCompany(companyData: any) {
        --     return this.http.post(this.API_URL + 'company/updateCompany',
companyData)
        --   }
        --
        --   createVacancy(vacancyData: any) {
```

-- return this.http.post(this.API_URL + 'vacancies/createVacancy', vacancyData)

-- }

-- }

6) company/vacancy-modal

- vacancy-modal.component.ts

-- import {Component, EventEmitter, OnInit, Output} from '@Angular/core';

--

-- @Component({

-- selector: 'app-vacancy-modal',

-- templateUrl: './vacancy-modal.component.html',

-- styleUrls: ['./vacancy-modal.component.css']

-- })

-- export class VacancyModalComponent implements OnInit {

-- @Output() onCreatVacancy = new EventEmitter();

-- vacancyData: any = {}

--

-- constructor() { }

--

-- ngOnInit(): void {

-- }

--

-- valueChange(value, field) {

-- this.vacancyData[field] = value

-- }

--

-- onCreatVacancyClicked() {

-- this.onCreatVacancy.emit(this.vacancyData);

-- }

--

-- }

.//company/vacancy-modal

- vacancy-modal.component.html

--

--

--

-- <app-input [value]="vacancyData.name" placeholder="Name" (valueChange)="valueChange($event, 'name')"></app-input>

--

--

-- <app-input [value]="vacancyData.decription" placeholder="decription" (valueChange)="valueChange($event, 'decription')"></app-input>

--

--

-- <app-input [value]="vacancyData.stack" placeholder="stack" (valueChange)="valueChange($event, 'stack')"></app-input>

--

--

-- <app-input [value]="vacancyData.yearsOfExpirience" placeholder="yearsOfExpirience" (valueChange)="valueChange($event, 'yearsOfExpirience')"></app-input>

--

--

-- <app-input [value]="vacancyData.cost" placeholder="cost" (valueChange)="valueChange($event, 'cost')"></app-input>

--

-- <app-button name="Create Vacancy" (click)="onCreatVacancyClicked()"></app-button>

--

--

- vacancy-modal.component.css

-- .background {

-- position: fixed;

-- left: 0;

-- right: 0;

-- top: 0;

-- bottom: 0;

-- background-color: rgba(91, 91, 91, 0.37);

-- z-index: 1;

-- }

--

-- .content {

-- width: 700px;

-- height: 500px;

-- background-color: white;

-- border-radius: 6px;

-- top:50%;

-- left:50%;

-- padding: 24px;

-- transform:translate(30%, 50%);

-- z-index: 2;

-- }

--

-- .content .item-block:not(:first-child) {

-- margin-top: 8px;

-- }

Всі ці компоненти імпортуються в app.module.ts import { BrowserModule } from '@Angular/platform-browser';

import { NgModule } from '@Angular/core';

```
import { AppRoutingModule } from './app-routing.module';
import { AppComponent } from './app.component';
import { CompanyComponent } from './_pages/company/company.component';
import { CompanyPageComponent } from './_pages/company/company-page/company-page.component';
import { VacancyModalComponent } from './_pages/company/vacancy-modal/vacancy-modal.component';
import { EmployeePageComponent } from './_pages/employee/employee-page/employee-page.component';
import { FindWorkListComponent } from './_pages/find-work-list/find-work-list.component';
import { CompanyPageService } from './_pages/company/company-page/company-page.service';
import { FindWorkListService } from './_pages/find-work-list/find-work-list.service';
import { CompanyService } from './_pages/company/company.service';
import { AvatarComponent } from './components/avatar/avatar.component';
import { BlockComponent } from './components/block/block.component';
import { ButtonComponent } from './components/button/button.component';
import { InputComponent } from './components/input/input.component';
import { TableComponent } from './components/table/table.component';
import { TextareaBlockComponent } from './components/textarea-block/textarea-block.component';
import { VacanciesListComponent } from './components/vacancies-list/vacancies-list.component';
import { HttpClientModule } from '@Angular/common/http';
import { FormsModule } from '@Angular/forms';

@NgModule({
```

```
  declarations: [
    AppComponent,
    CompanyComponent,
    CompanyPageComponent,
    VacancyModalComponent,
    EmployeePageComponent,
    FindWorkListComponent,
    AvatarComponent,
    BlockComponent,
    ButtonComponent,
    InputComponent,
    TableComponent,
    TextareaBlockComponent,
    VacanciesListComponent
  ],
  imports: [
    BrowserModule,
    AppRoutingModule,
    HttpClientModule,
    FormsModule
  ],
  providers: [CompanyPageService, FindWorkListService, CompanyService],
  bootstrap: [AppComponent]
})
export class AppModule { }
```

і підключаємо роутінг застосунку в файлі routing.module.ts

```
import { NgModule } from '@Angular/core';
import { Routes, RouterModule } from '@Angular/router';
import { FindWorkListComponent } from './_pages/find-work-list/find-work-list.component';
```

```
import { CompanyComponent } from './_pages/company/company.component';
import { EmployeePageComponent } from './_pages/employee/employee-page/employee-page.component';
import { CompanyPageComponent } from './_pages/company/company-page/company-page.component';


const routes: Routes = [
  {
    path: 'find-work',
    component: FindWorkListComponent,
  },
  {
    path: 'company',
    component: CompanyComponent,
  },
  {
    path: 'employee',
    component: EmployeePageComponent
  },
  {
    path: 'company-page',
    component: CompanyPageComponent
  },
  {
    path: '',
    component: FindWorkListComponent
  }
];
```

```
@NgModule({
  imports: [RouterModule.forRoot(routes)],
  exports: [RouterModule]
})
export class AppRoutingModule { }
```

Тепер ми маємо готовий одномодульний застосунок з декількома роутами (див.рис.3.3).

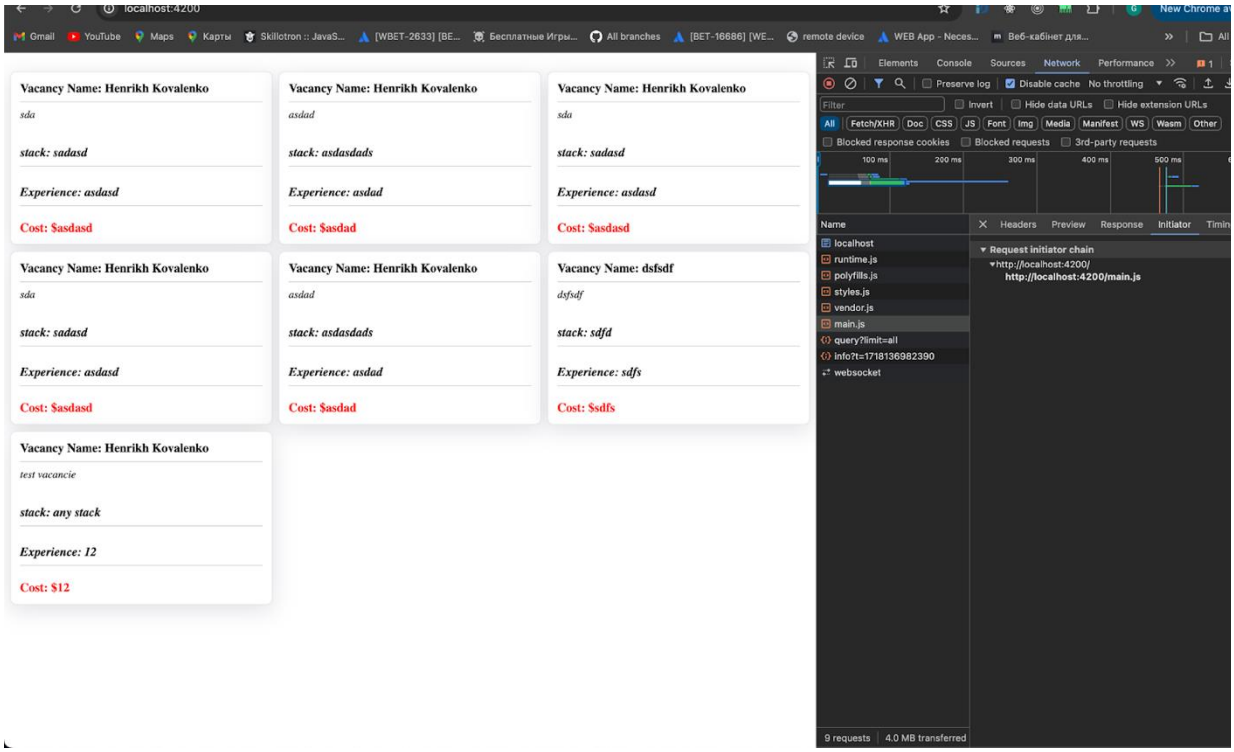

Рисунок 3.3 – Одномодульний застосунок

3.3 Перехід від одномодульної до багатомодульної архітектури в додатку Angular

Для переходу від одномодульної архітектури нам потрібно зміни архітектуру директорій. Так додаємо shared-module директорію в якій створюємо директорію components. В цю директорію необхідно перенести всі компоненти які будуть перевикористовуватись. Далі створюємо файл shered-

module.ts і реєструємо в ньому ці компоненти

```
import { NgModule } from '@Angular/core';
    import { CommonModule } from '@Angular/common';
    import { TableComponent } from './components/table/table.component';
    import { ButtonComponent } from './components/button/button.component';
    import { InputComponent } from './components/input/input.component';
    import {FormsModule} from '@Angular/forms';
    import { VacanciesListComponent } from './components/vacancies-
list/vacancies-list.component';
    import { BlockComponent } from './components/block/block.component';
    import { AvatarComponent } from './components/avatar/avatar.component';
    import { TextareaBlockComponent } from './components/textarea-
block/textarea-block.component';



    @NgModule({
      declarations: [
        TableComponent,
        ButtonComponent,
        InputComponent,
        VacanciesListComponent,
        BlockComponent,
        AvatarComponent,
        TextareaBlockComponent
      ],
        imports: [
            CommonModule,
            FormsModule
        ],
```

```
  exports: [
    TableComponent,
    ButtonComponent,
    InputComponent,
    VacanciesListComponent,
    BlockComponent,
    AvatarComponent
  ]
})
export class SharedModule { }
```

Зауважимо, що для того що б використовувати компоненти з shared-module в інших модулях необхідно їх додати в масив exports

Розподіляємо всі сторінки і додаємо в кожну сторінку <page_name>.routing.module.ts

```
import { NgModule } from '@Angular/core';
import { Routes, RouterModule } from '@Angular/router';
import { EmployeePageComponent } from 'src/app/_pages/employee/employee-page/employee-page.component';
const routes: Routes = [
  {
    path: '',
    component: EmployeePageComponent
  }
];

@NgModule({
  imports: [RouterModule.forChild(routes)],
  exports: [RouterModule],
  providers: [ ]
})
```

```
export class EmployeeRoutingModule { }
```

Також додаємо <page_name>.module.ts

```
import { NgModule } from '@Angular/core';
import { CommonModule } from '@Angular/common';
import { EmployeePageComponent } from
'src/app/_pages/employee/employee-page/employee-page.component';
import { EmployeeRoutingModule } from
'src/app/_pages/employee/employee.routing.module';
import { SharedModule } from 'src/app/shared-module/shared.module';
import { FormsModule } from '@Angular/forms';


@NgModule({
  declarations: [EmployeePageComponent],
  imports: [
    CommonModule,
    EmployeeRoutingModule,
    SharedModule,
    FormsModule
  ]
})
export class EmployeeModule {
}
```

Після цього модифікуємо app-routing.module.ts

```
import { NgModule } from '@Angular/core';
import { Routes, RouterModule } from '@Angular/router';


const routes: Routes = [
  {
```

```
      path: 'find-work',
      loadChildren: () => import('./_pages/find-work-list/find-work-
list.module').then((m) => m.FindWorkListModule)
    },
    {
      path: 'company',
      loadChildren: () => import('./_pages/company/company.module').then((m)
=> m.CompanyModule)
    },
    {
      path: 'employee',
      loadChildren: () => import('./_pages/employee/employee.module').then((m)
=> m.EmployeeModule)
    },
    {
      path: '',
      loadChildren: () => import('./_pages/find-work-list/find-work-
list.module').then((m) => m.FindWorkListModule)
    }
  ];

  @NgModule({
    imports: [RouterModule.forRoot(routes)],
    exports: [RouterModule]
  })
  export class AppRoutingModule { }
```

На цьому етапі перехід на багатомодульну структуру додатку завершено. Тепер ми маємо наступну структуру застосунку.

```
├── package-lock.json
├── package.json
```

```
├── src
│   ├── app
│   │   ├── Interfaces
│   │   │   └── TableData.ts
│   │   ├── _pages
│   │   │   ├── company
│   │   │   │   ├── company-page
│   │   │   │   │   ├── company-page.component.css
│   │   │   │   │   ├── company-page.component.html
│   │   │   │   │   ├── company-page.component.ts
│   │   │   │   │   └── company-page.service.ts
│   │   │   │   ├── company.component.css
│   │   │   │   ├── company.component.html
│   │   │   │   ├── company.component.ts
│   │   │   │   ├── company.module.ts
│   │   │   │   ├── company.routing.module.ts
│   │   │   │   ├── company.service.ts
│   │   │   │   └── vacancy-modal
│   │   │   │       ├── vacancy-modal.component.css
│   │   │   │       ├── vacancy-modal.component.html
│   │   │   │       └── vacancy-modal.component.ts
│   │   │   ├── employee
│   │   │   │   ├── employee-page
│   │   │   │   │   ├── employee-page.component.css
│   │   │   │   │   ├── employee-page.component.html
│   │   │   │   │   └── employee-page.component.ts
│   │   │   │   ├── employee.module.ts
│   │   │   │   └── employee.routing.module.ts
│   │   │   └── find-work-list
│   │   │       ├── find-work-list-routing.module.ts
```

```
│   │   │       ├── find-work-list.component.css
│   │   │       ├── find-work-list.component.html
│   │   │       ├── find-work-list.component.ts
│   │   │       ├── find-work-list.module.ts
│   │   │       └── find-work-list.service.ts
│   │   ├── app-routing.module.ts
│   │   ├── app.component.css
│   │   ├── app.component.html
│   │   ├── app.component.spec.ts
│   │   ├── app.component.ts
│   │   ├── app.module.ts
│   │   └── shared-module
│   │       ├── components
│   │       │   ├── avatar
│   │       │   │   ├── avatar.component.css
│   │       │   │   ├── avatar.component.html
│   │       │   │   └── avatar.component.ts
│   │       │   ├── block
│   │       │   │   ├── block.component.css
│   │       │   │   ├── block.component.html
│   │       │   │   └── block.component.ts
│   │       │   ├── button
│   │       │   │   ├── button.component.css
│   │       │   │   ├── button.component.html
│   │       │   │   └── button.component.ts
│   │       │   ├── input
│   │       │   │   ├── input.component.css
│   │       │   │   ├── input.component.html
│   │       │   │   └── input.component.ts
│   │       │   ├── table
```

```
│   │   │   ├── table.component.css
│   │   │   ├── table.component.html
│   │   │   └── table.component.ts
│   │   ├── textarea-block
│   │   │   ├── textarea-block.component.css
│   │   │   ├── textarea-block.component.html
│   │   │   └── textarea-block.component.ts
│   │   └── vacancies-list
│   │       ├── vacancies-list.component.css
│   │       ├── vacancies-list.component.html
│   │       └── vacancies-list.component.ts
│   │   └── shared.module.ts
│   ├── assets
│   ├── environments
│   │   ├── environment.prod.ts
│   │   └── environment.ts
│   ├── favicon.ico
│   ├── index.html
│   ├── main.ts
│   ├── polyfills.ts
│   ├── styles.css
│   └── test.ts
├── tsconfig.app.json
├── tsconfig.json
├── tsconfig.spec.json
└── tslint.json
```

При переході за допомогою роутінгу, ми можемо побачити, що сторінка завантажує окремий блок бандлу, і JS код розподіллено за модулями, це дає змогу не зберігати в пам’яті код який не застосовується і не потрібен користувачу в даний момент. Також слід звернути увагу, що застосунок з такою

структурою кращій для підтримки, оскільки ми маємо повністю екрановані модулі (див.рис.3.4).

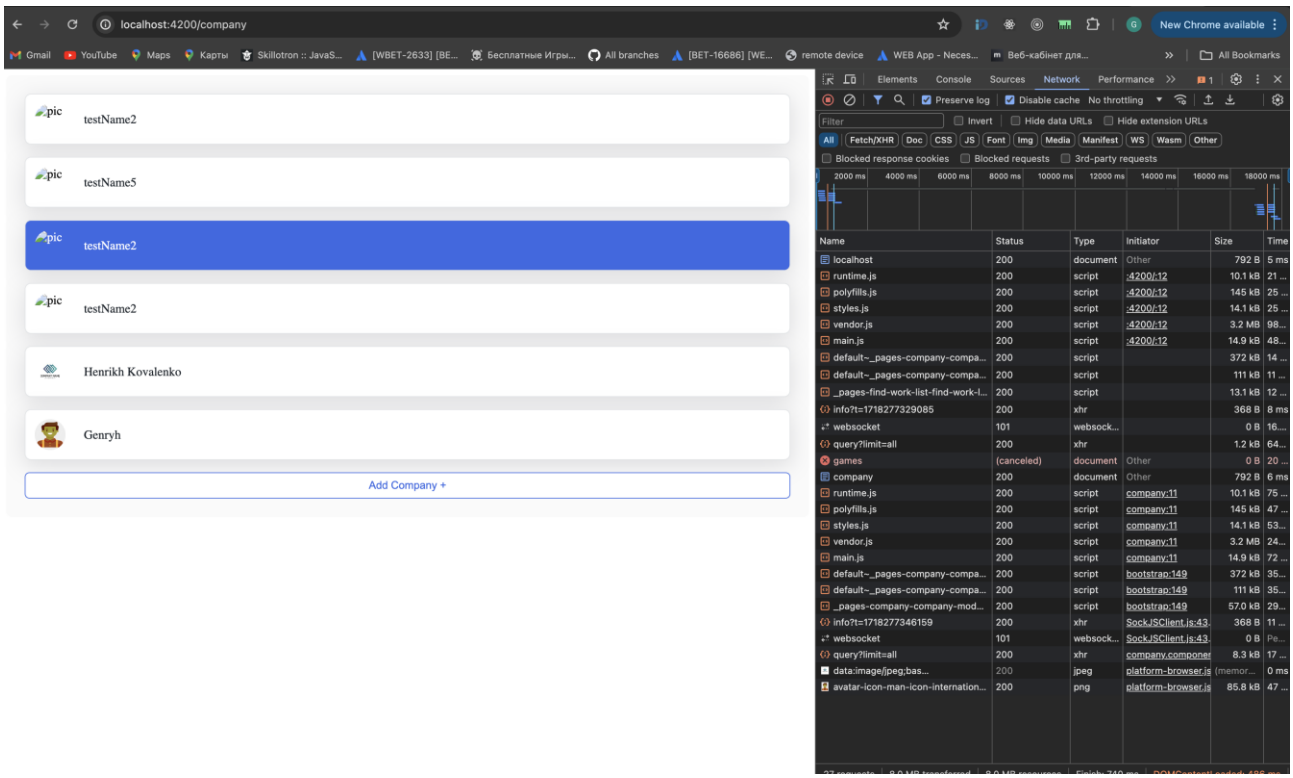

Рисунок 3.4 – Багатомодульний застосунок

3.4 Аналіз отриманого застосунку

Застосунок складається з чотирьох частин, а саме:

а) загальна сторінка вакансій (див.рис.3.5);

б) сторінка списку компаній (див.рис.3.6);

в) сторінка компанії (див.рис.3.7);

г) сторінка кандидата (див.рис.3.8).

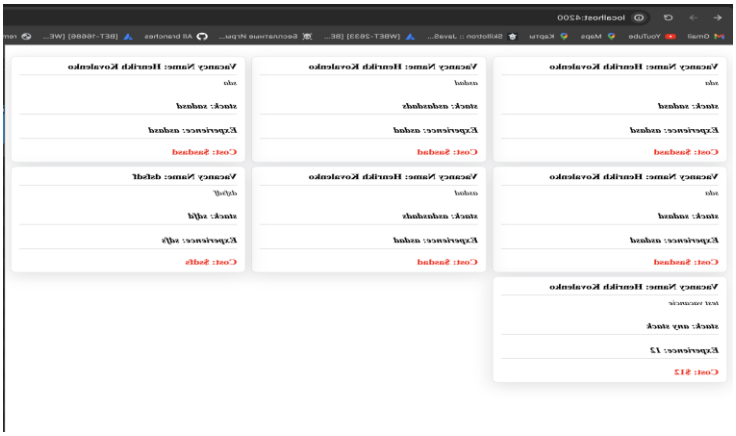

Рисунок 3.5 – Загальна сторінка вакансій

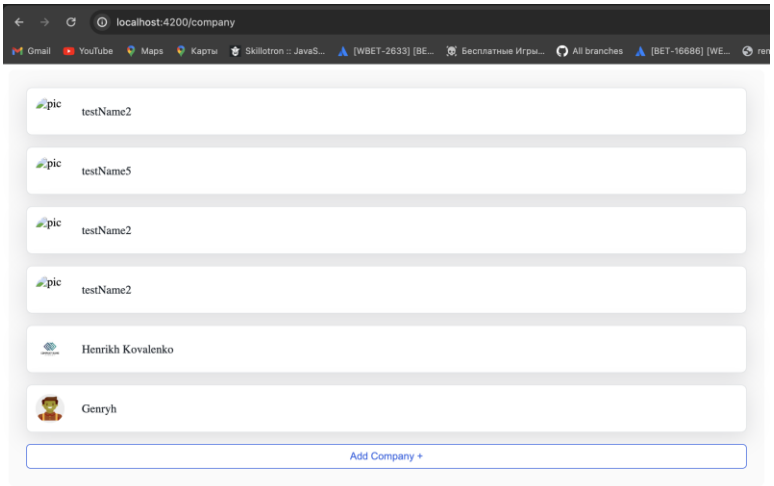

Рисунок 3.6 – Сторінка списку компаній

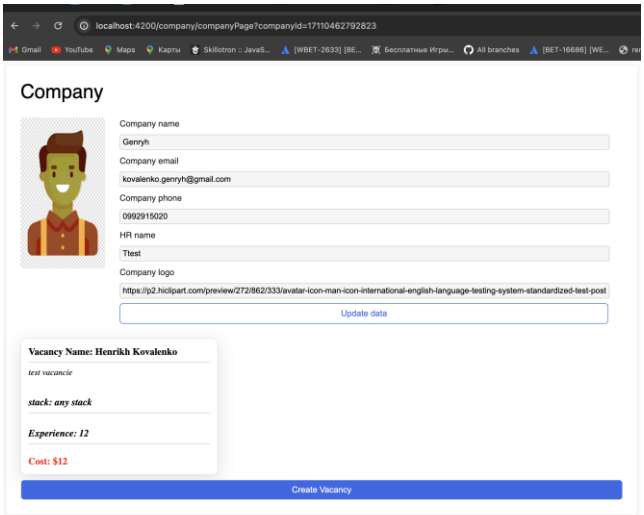

Рисунок 3.7 – Сторінка компанії

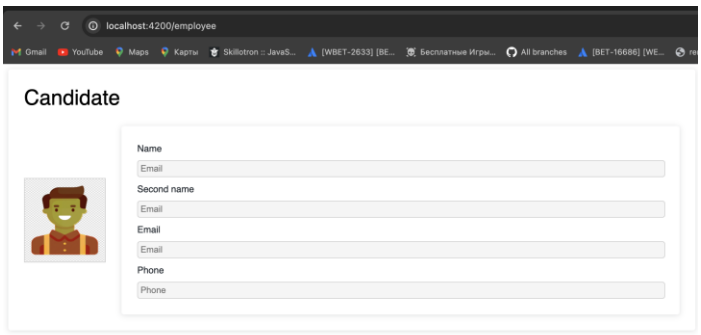

Рисунок 3.8 – Сторінка кандидата

Загальна структура застосунку має такий вигляд (див.рис.3.9).

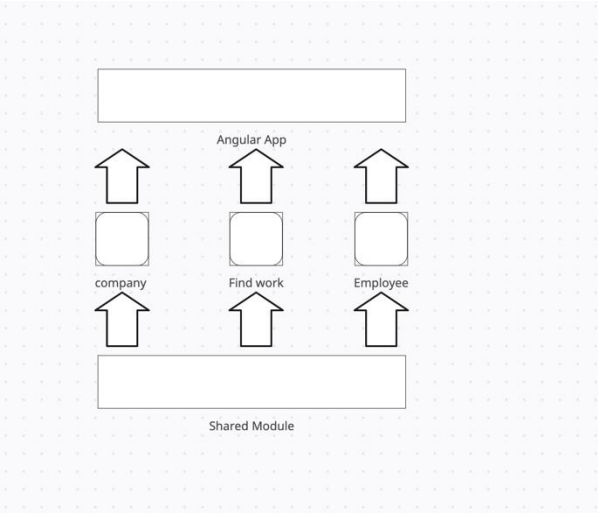

Рисунок 3.9 – Структура отриманого застосунку

Застосунок написаний з використанням вже застарілої версії фреймворку Angular, а саме 9, що не дасть змогу використовувати нові версії бібліотек, а також буде сповільняти розробку. Також можна зазначити, що окрім стартової сторінки кожна з сторінок може бути представлена як окремий проект, як наприклад, сторінка кандидата, може вміщати в себе ще й функції потенційного написання резюме для завантаження, або ж для використання на цій сторінці. Також для підвищення продуктивності розробки ці застосунки можна розділити з використанням мікро-фронтенд архітектури.

3.5 Створення дизайн бібліотеки

Для переходу до мікро-фронтенда архітектури нам необхідно, що б вигляд компонентів які застосовуються (полів вводу, кнопок і таке інше) в додатку був однаковий на всіх сторінках. В момент коли застосунок не використовує мікро-фронтенд в цьому немає проблеми, оскільки всі компоненти знаходяться в shared-module і використовуються по всьому додатку, при цьому є певні проблеми в той момент коли ми використовуємо більше ніж один додаток. Для того що б забезпечити єдиний вид і функціональність компонентів використовуються дизайн бібліотеки такі як material ui. При цьому в тому випадку коли в нас є власні компоненти ми повинні написати власну дизайн бібліотеку компонентів.

Загалом дизайн бібліотека може бути розроблена за допомогою багатьох фреймворків, а також за допомогою веб компонентів. В цій роботі буде використовуватись react для створення дизайн бібліотеки.

Для цього створюємо директорію проекту переходимо в неї і виконуємо команду npm init. Далі команду npm install react typescript @types/react --save-dev для встановленя реакт. Далі необхідно перенести компоненти які знаходяться в shared-module основного застосунку до директорії зберігаючи основну структуру застосунку який було створено виконанням попередньої команди.

```
├── package-lock.json
├── package.json
└── src
    ├── components
    │   ├── Button
    │   │   ├── Button.css
    │   │   ├── Button.tsx
    │   │   └── index.ts
```

```
│   └── index.ts
├── index.css
└── index.ts
```

Через те, що ми змінили Angular на реакт то компоненти з shared-module повинні бути переписані на реакт. Загальний вигляд компонентів бібліотеки буде такий

```
.//components/Input
 - Input.css
  -- .input-header {
  --    font-family: "Roboto", sans-serif;
  --    color: #1C232B;
  --    font-size: 14px;
  --    font-weight: 400;
  --    line-height: 20px;
  --    text-align: left;
  -- }
  --
  -- .input-container input {
  --    width: calc(100% - 8px);
  --    margin-top: 8px;
  --    padding: 4px;
  --    outline: none;
  --    background-color: #F5F5F5;
  --    border: 1px solid  #CECECE;
  --    border-radius: 4px;
  -- }
  --
  -- .input-container input:focus {
  --    border-color: #6582dc;
```

```
 --     background-color: rgba(101, 130, 220, 0.15);
 -- }
 --
 -- .input-container input.filled {
 --     background-color: rgba(101, 130, 220, 0.15);
- index.ts
.//components/Input
 - Input.tsx
 -- import React from 'react'
 --
 -- import './Input.css'
 --
 -- interface InputProps {
 --     header: string;
 --     value: string;
 --     onChangeCallback: (value: string) => void
 -- }
 --
 -- const Input = ({ header, onChangeCallback, value }: InputProps) => {
 --
 --     const onInputChange = (e: any) => {
 --         onChangeCallback(e.target.value)
 --     }
 --
 --     return (
 --         <div className="input-container">
 --             <div className="input-header">
 --                 {header}
 --             </div>
```

-- <input className={value.length ? 'filled' : ''} value={value} type="text" onChange={onInputChange}/>

--

--

--)

-- }

--

.//components/Button

- Button.css

-- .button {

-- border: 1px solid #3A66E5;

-- background-color: #FFFFFF;

-- border-radius: 6px;

-- color: #3A66E5;

-- font-weight: 400;

-- font-size: 14px;

-- width: 100%;

-- padding: 8px;

-- cursor: pointer;

-- }

--

-- .button:hover {

-- color: white;

-- background-color: #3A66E5;

-- border: 1px solid #FFFFFF;

-- }

.//components/Button

- Button.tsx

-- import React from 'react';

--

```
-- import './Button.css'
--
-- export interface ButtonProps {
--     name: string
--     onClickCallback: any
-- }
--
--
-- const Button = ({ name, onClickCallback }: ButtonProps) => {
--     return <button className="button"
onClick={onClickCallback}>{name}</button>
-- }
--
.//components/Table
 - Table.css
 -- .table  {
 --     width: calc(100% - 48px);
 --     padding: 24px;
 --     border-radius: 8px;
 --     background-color: #FAFAFA;
 -- }
 --
 -- .table .list-item {
 --     display: flex;
 --     align-items: center;
 --     background-color: white;
 --     border-radius: 8px;
 --     padding: 12px;
 --     border: 1px solid #E2E5E7;
 --     box-shadow: rgba(100, 100, 111, 0.2) 0 7px 29px 0;
```

```
  -- }
  --
  -- .table .list-item:hover {
  --     background-color: #3A66E5;
  --     color: white;
  -- }
  --
  -- .table .list-item:hover .name {
  --     color: white;
  -- }
  --
  -- .table .list-item:not(:first-child) {
  --     margin-top: 16px;
  -- }
  --
  -- .avatar {
  --     width: 40px;
  --     height: 40px;
  --     border-radius: 50%;
  --     margin-right: 24px;
  -- }
  --
  -- .name {
  --     font-weight: 500;
  --     font-size: 16px;
  --     color: #1c232b;
  -- }
- Table.tsx
  -- import React, { ReactNode } from 'react';
  --
```

```
-- import './Table.css'
--
-- export interface TableProps {
--     tableData: {id: string, img: string, name: string}[],
--     itemCallBack: (itemId: string) => void,
--     children: ReactNode
-- }
--
-- const TableComponent = ({ tableData, itemCallBack, children }:
TableProps) => {
--     return (
--         tableData && (
--             <div className="table">
--                 {tableData.map((item) => (
--                     <div
--                         key={item.id}
--                         className="list-item"
--                         onClick={() => itemCallBack(item.id)}
--                     >
--                         {item.img && <img className="avatar" src={item.img}
alt="pic" />}
--                         <div className="name">{item.name}</div>
--                     </div>
--                 ))}
--                 {children}
--             </div>
--         )
--     );
-- };
--
```

```
 -- export default TableComponent;
.//components/Table
 - index.ts
.//components/Avatar
 - Avatar.tsx
  -- import React from 'react'
  --
  -- import './Avatar.css';
  --
  -- export interface AvatarProps {
  --     url: string;
  -- }
  --
  -- const Avatar = ({url}: AvatarProps) => {
  --     return (
  --         <img
  --             className="personal-block-avatar"
  --             src={url}
  --             alt="avatar"
  --         />
  --     )
  -- }
  --
- index.ts
- Avatar.css
  -- .personal-block-avatar {
  --     display: block;
  --     box-sizing: border-box;
  --     height: 128px;
  --     width: 124px;
```

-- border: 1px solid #CECECE;

-- }

.//components/TextareaBlock

- TextareaBlock.tsx

-- import React from 'react'

-- import './TextareaBlock.css'

--

-- export interface TextareaBlockProps {

-- header: string;

-- value: string;

-- minHeight?: string;

-- onChangeCallback: (value: string) => void

-- }

--

-- const TextareaBlock = ({ header, onChangeCallback, value, minHeight = '41px' }: TextareaBlockProps) => {

--

-- const onInputChange = (e: any) => {

-- onChangeCallback(e.target.value)

-- }

--

-- return (

--

--

-- {header}

--

-- <textarea

-- style={{ minHeight }}

-- className={value.length ? 'filled' : ''}

-- value={value}

```
--                onChange={onInputChange}
--            />
--        </div>
--
--    )
-- }
--
```

- TextareaBlock.css

```
-- .textarea-header {
--     font-family: "Roboto", sans-serif;
--     font-size: 14px;
--     font-weight: 400;
--     line-height: 20px;
--     text-align: left;
-- }
--
-- .textarea-container textarea {
--     width: calc(100% - 8px);
--     margin-top: 8px;
--     padding: 4px;
--     outline: none;
--     background-color: #F5F5F5;
--     border: 1px solid #CECECE;
--     border-radius: 4px;
--     resize: none;
-- }
--
-- .textarea-container textarea:focus {
--     border-color: #6582dc;
--     background-color: rgba(101, 130, 220, 0.15);
```

```
 -- }
 --
 -- .textarea-container textarea.filled {
 --     background-color: rgba(101, 130, 220, 0.15);
 -- }
- index.ts
.//components/VacanciesList
- VacanciesList.tsx
 -- import React from 'react';
 --
 -- import './VacanciesList.css'
 --
 -- export interface VacanciesListProps {
 --     vacancies: {
 --         name: string;
 --         description: string;
 --         stack: string;
 --         yearsOfExperience: string;
 --         cost: string;
 --     }[]
 -- }
 --
 -- const VacanciesList = ({ vacancies }: VacanciesListProps) => {
 --     return (
 --         <div className="table">
 --             {vacancies.map((vacancy, index) => (
 --                 <div className="list-item" key={index}>
 --                     <div className="main-block">
 --                         <div className="name">Vacancy Name:
{vacancy.name}</div>
```

```
        --                  <div className="description">{vacancy.description}</div>
        --                  <div className="stack">Stack: {vacancy.stack}</div>
        --                  <div className="stack">Experience:
{vacancy.yearsOfExperience}</div>
        --                </div>
        --                <div className="cost">Cost: ${vacancy.cost}</div>
        --              </div>
        --            ))}
        --        </div>
        --    );
        -- };
        --
        -- export default VacanciesList;
       - index.ts
       - VacanciesList.css
        -- .table {
        --    display: grid;
        --    grid-gap: 8px;
        --    grid-template-columns: 1fr 1fr 1fr;
        --    margin-top: 24px;
        --    margin-bottom: 8px;
        -- }
        --
        -- .table .list-item {
        --    display: flex;
        --    flex-direction: column;
        --    justify-content: space-between;
        --    height: 200px;
        --    background-color: white;
        --    border-radius: 8px;
```

```
--     padding: 12px;
--     border: 1px solid #E2E5E7;
--     box-shadow: rgba(100, 100, 111, 0.2) 0px 7px 29px 0px;
-- }
--
-- .table .list-item:hover {
--     background-color: #3A66E5;
-- }
-- .table .list-item:hover .name,
-- .table .list-item:hover .description,
-- .table .list-item:hover .stack {
--     color: white;
-- }
--
-- .name {
--     padding-bottom: 8px;
--     border-bottom: 1px solid #CECECE;
--     font-weight: 600;
--     font-size: 16px;
-- }
--
-- .description {
--     padding-bottom: 8px;
--     padding-top: 8px;
--     font-style: italic;
--     font-size: 14px;
-- }
--
-- .stack {
--     padding-bottom: 8px;
```

```
 --     margin-top: 24px;
 --     border-bottom: 1px solid #CECECE;
 --     font-weight: 600;
 --     font-size: 16px;
 --     font-style: italic;
 -- }
 --
 -- .cost {
 --     font-weight: 600;
 --     font-size: 16px;
 --     color: red
 -- }
.//components/Block
 - Block.tsx
 -- import React from 'react'
 --
 -- import './Block.css'
 --
 -- export interface BlockProps {
 --     header: string
 --     children: any
 -- }
 --
 -- const Block = ({
 --    header,
 --    children
 -- }:BlockProps) => {
 --     return (
 --         <div className="block-container">
 --             <div className="block-header">
```

```
  --              {header}
  --            </div>
  --            <div className="content-container">
  --              {children}
  --            </div>
  --          </div>
  --      )
  -- }
  --
    - index.ts
- Block.css
  -- .block-container {
  --     padding: 24px;
  --     background-color: white;
  --     box-shadow: 0px 1px 8px 0px #1C232B26;
  --     border-radius: 4px;
  -- }
  --
  -- .block-header {
  --     font-family: "Roboto", sans-serif;
  --     font-size: 32px;
  --     font-weight: 500;
  --     line-height: 38px;
  -- }
  --
  -- .content-container {
  --     margin-top: 24px;
  -- }
.
 - index.ts
```

Після цього структура повинна виглядати приблизно так

```
├── package-lock.json
├── package.json
└── src
    ├── components
    │   ├── Avatar
    │   │   ├── Avatar.css
    │   │   ├── Avatar.tsx
    │   │   └── index.ts
    │   ├── Block
    │   │   ├── Block.css
    │   │   ├── Block.tsx
    │   │   └── index.ts
    │   ├── Button
    │   │   ├── Button.css
    │   │   ├── Button.tsx
    │   │   └── index.ts
    │   ├── Input
    │   │   ├── Input.css
    │   │   ├── Input.tsx
    │   │   └── index.ts
    │   ├── Table
    │   │   ├── Table.css
    │   │   ├── Table.tsx
    │   │   └── index.ts
    │   ├── TextareaBlock
    │   │   ├── TextareaBlock.css
    │   │   ├── TextareaBlock.tsx
    │   │   └── index.ts
    │   └── index.ts
```

```
    ├── index.css
    └── index.ts
```

Додаємо tsconfig.json за допомогою команди npx tsc –init і додаємо значення

```
{
  "compilerOptions": {
    // за замовченням
    "target": "es5",
    "esModuleInterop": true,
    "forceConsistentCasingInFileNames": true,
    "strict": true,
    "skipLibCheck": true,

    // додаємо
    "jsx": "react",
    "module": "ESNext",
    "declaration": true,
    "declarationDir": "types",
    "sourceMap": true,
    "outDir": "dist",
    "moduleResolution": "node",
    "allowSyntheticDefaultImports": true,
    "emitDeclarationOnly": true,
  }
}
```

Для збірки нашого застосунку додаємо rolup командою npm install rollup @rollup/plugin-node-resolve @rollup/plugin-typescript @rollup/plugin-commonjs rollup-plugin-dts --save-dev

і доодамо запуск в json "scripts": {

```
    "rollup": "rollup -c"
```

},

Далі необхідно створити реепозиторій на github після чого в проекті виконуємо команду git init. Переписуємо package.json додаючи в нього:

"name": "<Github user namr> /<repo name>",

"publishConfig": {

"registry": "https://npm.pkg.github.com/GenryhK"

},

Для публікації бібліотеки нам знадобиться створити файл .npmrc з таким змістом:

registry=https://registry.npmjs.org/

@USER_NAME:registry=https://npm.pkg.github.com/

//npm.pkg.github.com/:_authToken=TOKEN_FROM_GIT

Після цього натискаєм npm publish. Тепер наша версія бібліотеки з компонентами опублікована на гіт хаб, і ми можемо встановлювати її в наступні застосунки.

3.6 Перехід на мікро-фронтенд за використанням iframe технології

Для екранізації переходу на мікро-фронтенд архітектуру перенесемо в окремий проект companyPage. В якості прикладу другою бібліотекою використаємо react за допомогою команди npx create-react-app створюємо новий react проект після чого встановимо створену стайл бібліотеку за допомогою команди npm i npm install @REPO_NAME/style-lib.. І за допомогою неї перепишемо нашу сторінку компанії. Після того як сторінка готова встановлюємо в основному додатку на сторінці companyPage тег iframe з посиланням на новий проект (див.рис.3.10).

Якщо зараз ми запустимо застосунок то ми не зможемо отримати дані, оскільки companyPage отримувала дані з сервера відправляючи для їх отримання companyId який ця сторінка отримувала з url гет параметра, але ми не маємо можливості передавати цей параметр через url в iframe.

Рисунок 3.10 – Тег iframe з посиланням на новий проект

Для того, що б передати параметр в середину iframe нам доведеться використати postMessage з основного додатку і передати його в реакт додаток. Для цього нам необхідно видозмінити метод ngOnInit таким чином:

```
ngOnInit(): void {
  this.route.queryParams.subscribe((params) => {

    this.iframe = document.getElementById('react-company-page') as
HTMLIFrameElement;

    setTimeout(() => {
      this.iframe.contentWindow.postMessage({companyId:
params.companyId}, '*');
    }, 1000)

  })
 }
```

А в реакт частині застосунку ми підписуємось на лісенер що б знати коли буде передано код:

```
const getData = (companyId: string) => {
  fetch(API_URL + 'company/companyData?' + new URLSearchParams({
companyId } ))
    .then((res) => res.json())
    .then((data) => data && setCompanyData(data))
}

useEffect(() => {
  window.addEventListener('message', (event) => {
    const { companyId } = event.data

    companyId && !companyData.companyName &&
getData(companyId)
  });
}, []);
```

Варто звернути увагу, що в основному ангуляр застосунку ми використовуємо setTimeout, що є дуже поганою практикою з точки зору веб розробки. Оскільки ми вириваємо деяку частину коду з контексту виконання і передається в EventLoop звідки вона буде викликана не раніше ніж буде вказано в таймері, при цьому вона може бути викликана пізніше. Використання тайм-ауту зумовлено тим, що нам необхідно відправити id не раніше ніж завантажиться сторінка мікро-фронтенду. Іншим і більш правильним способом реалізувати необхідну поведінку буде підписатись на action onload у айфрейма:

```
ngOnInit(): void {
  this.route.queryParams.subscribe((params) => {

    this.iframe = document.getElementById('react-company-page') as
HTMLIFrameElement;

    this.iframe.onload = () => {
```

```
            this.iframe.contentWindow.postMessage({companyId:
params.companyId}, '*');
          }
        })
      }
```

Також потрібно зауважити, що операції з чистим js, такі як document.getElementById, є доволі витратними в фреймворках, оскільки при такому зверненні пошук буде проходити з самого початку DOM дерева поки елемент не буде знайдений.

Ми можемо побачити що імплементація через iframe підхід доволі швидка і не дуже затратна з точки зору ресурсів. Разом з тим, як ми бачимо при використанні iframe застосунки не пов’язані між собою і, за великим рахунком, являють собою повністю окремі додатки які не повністю інтегровані один в одний. Через це ми можемо побачити такі проблеми як відсутність контролю між застосунками, ми не маємо контролю за url і за рендором одного застосунку в середині іншого. В подальшому такі питання можуть спричинити багато проблем оскільки застосунки не можуть бути синхронізовані між собою. Як висновок можна сказати, що в тому випадку коли необхідний швидкий результат технологія iframe може бути застосована для переходу на мікро-фронтенд, але в разі якщо є час і ресурси то раціональніше імплементувати мікро-фронтенд іншим способом.

3.7 Перехід на мікро-фронтенд за допомогою бібліотеки simgle-spa

Для імплементації мікрофронтенд за допомогою single spa встановимо його глобально за допомогою команди npm install --global create-single-spa. Після цього нам потрібно зробити рут проект для single spa в який будуть підключатись інші частини мікрофронтенду для цього нам потрібно перейти в директорію і виконати команду npx create-single-spa --moduleType root-config

Після виконання цієї команди нам необхідно вибрати ім’я проекту,

пакетний менеджер котрий буде використовуватись, чи буде використовуватись тайпскрипт і так далі.

Тепер починаємо переносити проект. Але якщо подивитись на основну документацію, то можно побачити що версія Angular 9 не підтримується для переходу. Нам необхідно виконати команду ng update @Angular/core@10 @Angular/cli@10 --force для переходу на більшу версію. Далі в директорії яка з'явилась, створюємо директорію для проекту і переміщуємо додаток Angular в неї. Після цього необхідно виконати команду ng add single-spa-Angular в цій директорії, вона автоматично виконає такі дії:

1) встановить `single-spa-Angular`;

2) згенерує `main.single-spa.ts` у проекті в `src/`;

3) згенерує `single-spa-props.ts` в `src/single-spa/`;

4) згенерує `asset-url.ts` в `src/single-spa/`;

5) згенерує `EmptyRouteComponent` в `src/app/empty-route/`, щоб використовувати його в `app-routing.module.ts`;

6) додає npm скрипт `npm run build:single-spa`;

7) додає npm скрипт `npm run serve:single-spa`.

Для встановлення залежностей:

1) виконуємо команду npm;

2) заходимо в файл Angularjs і змінюємо @Angular-devkit/build-Angular на @Angular-builders/custom-webpackК;

3) додаємо дві залежності

npm i @Angular-builders/custom-webpack

npm i @types/node@14.0.4 --saveDev

4) заходимо в app.module.ts імпортуємо

import {APP_BASE_HREF} from '@Angular/common';

і додаємо в масив providers

{provide: APP_BASE_HREF, useValue: '/'}

Тепер в кореневій директорії проекту нам потрібно додати зміни до файлу microfrontend-layout.html, а саме, додати наш застосунок (див.рис.3.11).

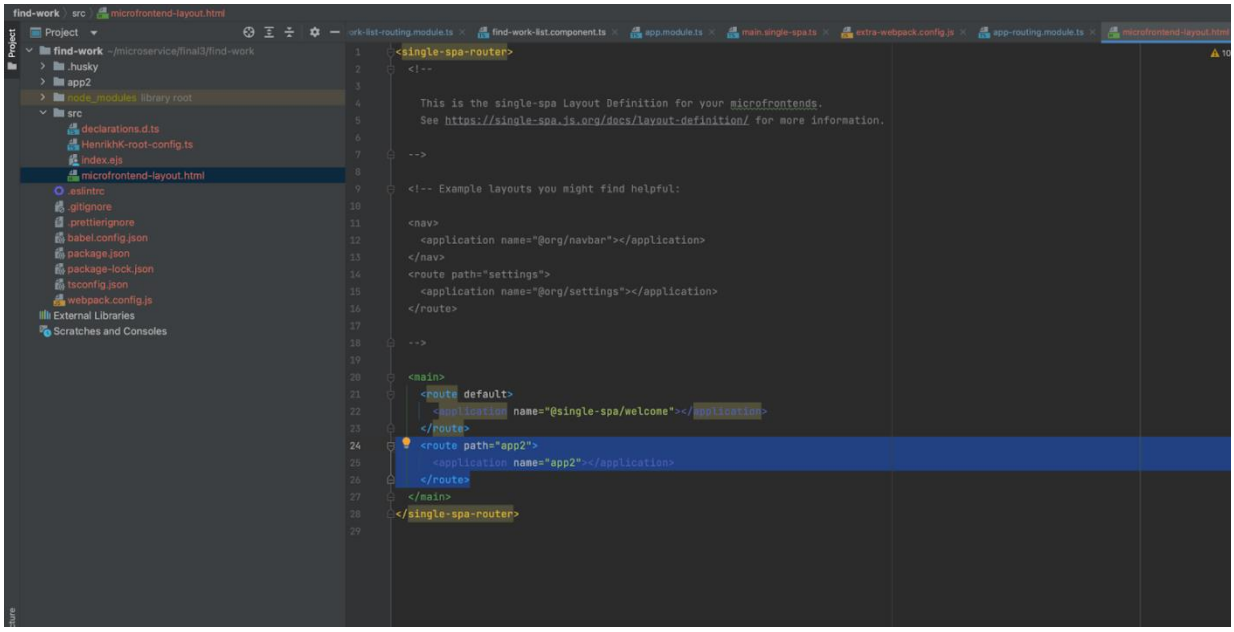

Рисунок 3.11 – Зміни до файлу microfrontend-layout.html

В файлі index.ejs додаємо реєстрацію мікро-фронтенду (див.рис.3.12).

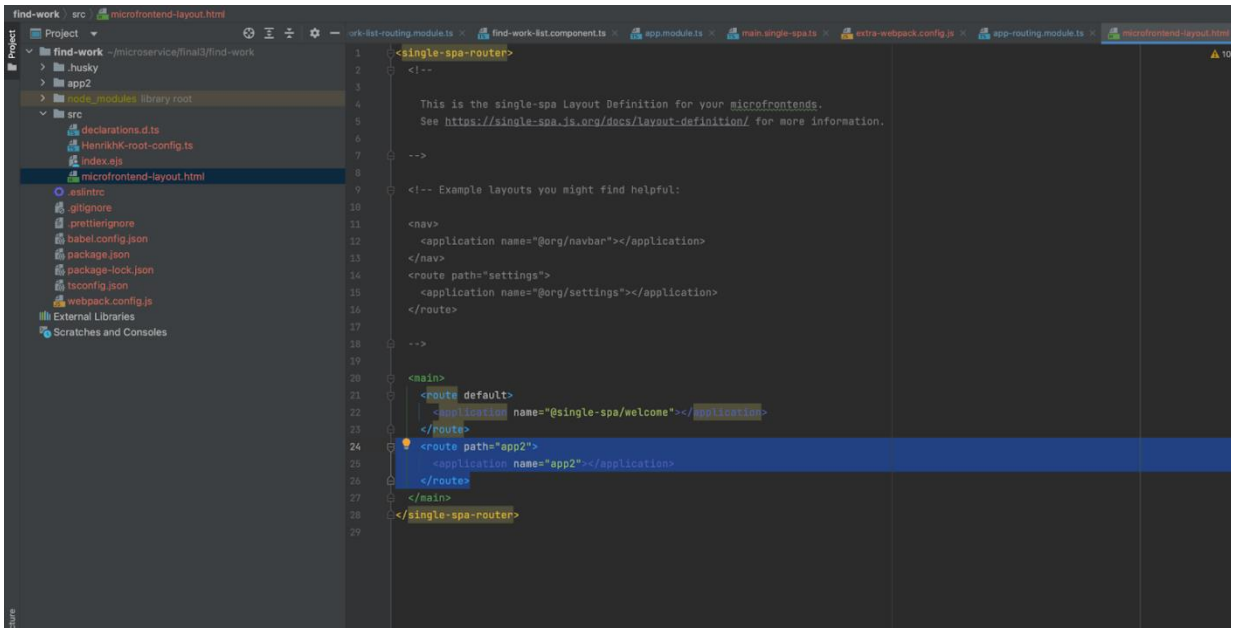

Рисунок 3.12 – Реєстрація мікро-фронтенду

Після цього ми отримуємо підключений перший проект (див.рис.3.13).

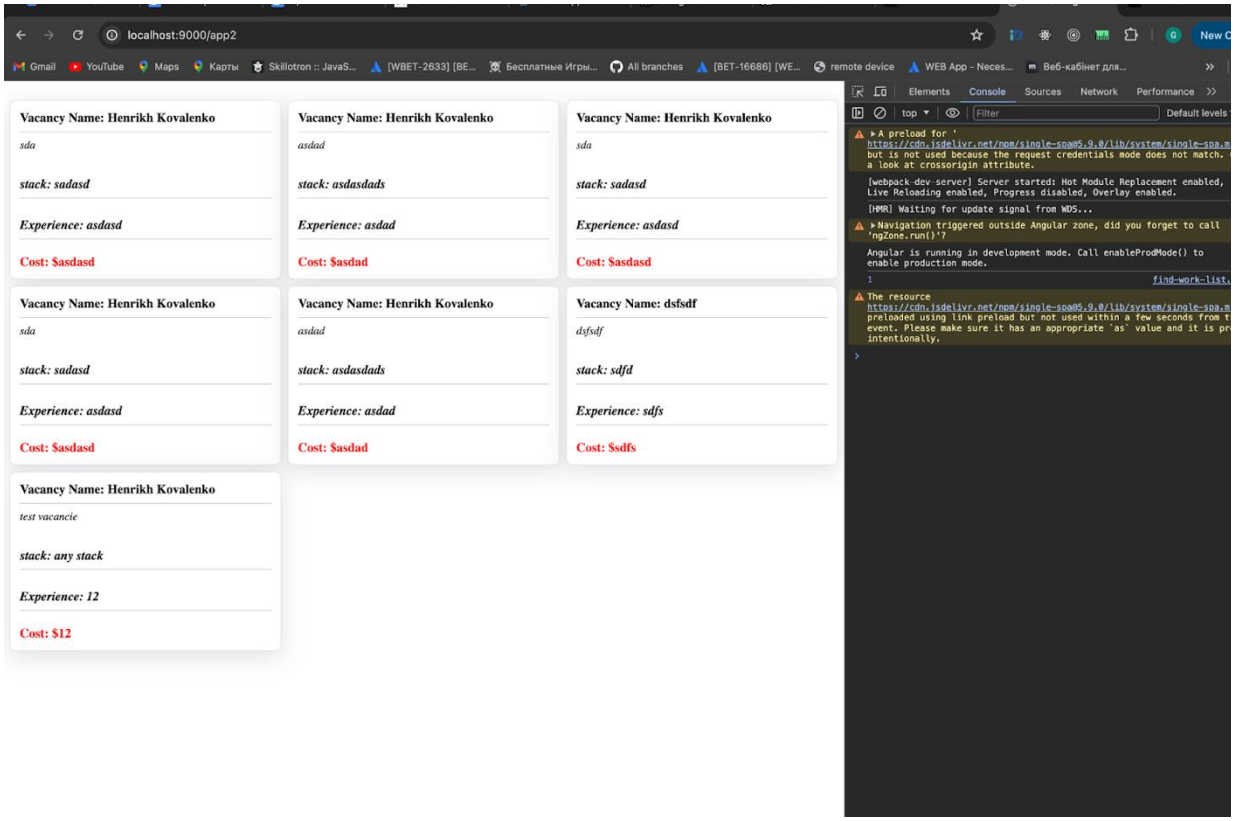

Рисунок 3.13 – Підключений основний проект до single-spa

Далі в файлі microfrontend-layout.html видаляємо

```
<route >
  <application name="@single-spa/welcome"></application>
</route>
```

і в тегу роуту нашого замінюємо path на default. Після цього додаток Angular став головним і ми можемо переходити по роутах.

Тепер потрібно винести companyPage в окремий проект, по аналогії як це робилось в попередньому розділі, створюємо додаток реакт за допомогою команди create-single-spa --framework react і встановлюємо в нього стайл бібліотеку яка була створена. В додатку Angular блокуємо роут companyPage

Встановлюємо залежності командою npm i і запускаємо командою npm start -- --port 8500

в файл microfrontend-layout.html додаємо

```
<route path="company-page">
  <application name="company-page"></application>
```

</route>

також в файл index.ejs додаємо

"react": "https://cdn.jsdelivr.net/npm/react@16.13.1/umd/react.production.min.js",

"react-dom": "https://cdn.jsdelivr.net/npm/react-dom@16.13.1/umd/react-dom.production.min.js",

"company-page": "//localhost:8500/HenrikhK-company-page.js"

в imports

Після цього сторінка буде доступна і за допомогою бібліотеки компонентів перенесемо сторінку з ангуляр додатку. Сформуємо повне посилання з getPharams в урлі (див.рис.3.14).

```
const getData = () => {
    const urlParams = new URLSearchParams(window.location.search);
    const companyId = urlParams.get('companyId');
    fetch( input: API_URL + 'company/companyData?' + new URLSearchParams({ companyId } )) Promise<Response>
        .then((res : Response ) => res.json()) Promise<any>
        .then((data) => data && setCompanyData(data))
}

useEffect(() => {
    getData()
}, []);
```

Рисунок 3.14 – Посилання з getPharams в урлі

Тепер видозмінимо основну сторінку реакт застосунку для того що б прийняти getPharams

import { Block, Input, Button, VacanciesList } from '@GenryhK/style-lib';

import './CompanyPage.css'

import { useEffect, useState } from 'react';

export const CompanyPage = () => {

const API_URL: string = 'http://localhost:5000/api/v1/'

```
        const [companyData, setCompanyData] = useState({
            companyLogo:'',
            companyName: '',
            companyPhone: '',
            hrName: '',
            vacancies: []
        })

        const onFieldChangeHandler = () => {}

        const onUpdateData = () => {}

        const getData = () => {
            const urlParams = new URLSearchParams(window.location.search);
            const companyId = urlParams.get('companyId');
            fetch(API_URL + 'company/companyData?' + new URLSearchParams({
companyId } ))
                .then((res) => res.json())
                .then((data) => data && setCompanyData(data))
        }

        useEffect(() => {
            getData()
        }, []);
        return (
            <Block header="Company">
                <div className="information">
                    <img className="logo" src={companyData.companyLogo}
alt="img" />
                    <div className="company-data">
```

```
                    <div className="item-block">
                        <Input header="Company name"
value={companyData.companyName} onChangeCallback={onFieldChangeHandler}
/>
                    </div>
                    <div className="item-block">
                        <Input header="Company phone"
value={companyData.companyPhone} onChangeCallback={onFieldChangeHandler}
/>
                    </div>
                    <div className="item-block">
                        <Input header="HR name" value={companyData.hrName}
onChangeCallback={onFieldChangeHandler} />
                    </div>
                    <div className="item-block">
                        <Input header="Company logo"
value={companyData.companyLogo} onChangeCallback={onFieldChangeHandler}
/>
                    </div>
                    <div className="item-block">
                        <Button name="Update data" onClickCallback={onUpdateData
as any} />
                    </div>
                </div>
            </div>
            <VacanciesList vacancies={companyData.vacancies} />
        </Block>
    )
}
```

Після цього перехід можна вважати закінченим і ми можемо

використовувати застосунок який повністю інтегрований і по суті являється тепер одним застосунком, може контролювати рендер в середині частин мікро-фронтендів і передавати дані між мікро-фронтендами.

4 ОЦІНКА ТРАНСФОРМАЦІЇ ДОДАТКУ

4.1 Оцінка складності трансформації від моноліту до мікро-фронтенду

Після трансформації додатку від моноліту в мікро-фрнтенд можна зробити висновки стосовно складності переходу з однієї структури в іншу. А також оцінити технологічні рішення які наводились в цій роботі.

Оцінку складності трансформації можна проводити в людино годинах помножених на частину складності застосунку.

Проаналізуємо створення ui бібліотеки, яка необхідна для використання:

1) ініціювання додатку і створення загальної структури – 1 година;

2) інсталяція залежностей – 0.5 годин;

3) перенесення компоненту – 0.5 годин/компонент (в середньому);

4) підготовка до публікації і публікація – 1 година.

З цього виходить 1 + 1 + 0,5 + (0,5 * 7) = 6 годин (в середньому).

Проаналізуємо створення мікро-фронтенд сторінки:

1) Ініціювання додатку і створення загальної структури – 3 година

2) Інсталяція залежностей – 1.5 годин

3) Перенесення компоненту – 1.5 годин / компонент в середньому

4) Підключення компоненту до API – 1 година

З цього виходить 3 + 1.5 + (1.5 * 1) = 7 годин (в середньому).

Підключення мікро-фронтенду до загального застосунку.

Враховуючи два наведені результати, проведемо аналіз для використання технології iframe та бібліотеки:

а) технологія Iframe:

1) підключення застосунку – 0.5 годин;

б) бібліотека single spa:

1) створення загального застосунку – 3 години;

2) перенесення існуючого застосунку включно з можливими проблемами пов’язаними з несумісністю залежностей – 3 години;

3) підключення мікро-фронтенду – 1 година.

Тепер введемо константу складності застосунку від 1 до 5, де 1 це «не складний застосунок», а 5 – «надскладний застосунок». Ця константа є оціночним числом, яке буде залежати від рівня команди, а також складності залежностей в самому застосунку і кількості коду в ньому.

З наведеного вище виходить що перехід до мікро-фронтенду, за допомогою технології iframe, зайняв (6 + 7 + 0.5) * 1 = 13,5 людино/годин.

Перехід же за допомогою single spa зайняв (6 + 7 + 3 + 3 + 1) * 1 = 20 людино/годин.

4.2 Оцінка отриманих додатків

Після отриманих підрахунків ми можемо побачити, що перехід з використанням технології iframe набагато швидший за перехід з використанням бібліотеки single spa. Але поряд з цим ми можемо назвати головні мінуси використання цього підходу:

1) сумісність з браузерами: Не всі браузери підтримують iframe, особливо старі версії, що може спричинити проблеми у роботі веб-додатків;

2) ускладнення навігації: Використання iframe може викликати проблеми з навігацією та історією браузера, що впливає на користувацький досвід;

3) проблеми з SEO: Зміст, що введено через iframe, може бути не індексованим пошуковими системами, що може негативно вплинути на SEO;

4) зниження продуктивності: Iframe може сповільнити завантаження сторінки, особливо при великих кількостях даних;

5) відсутність контролю стилів: Iframe має власний DOM і контекст виконання, через що не дозволяє контролювати стилі з головної сторінки;

6) безпека: Використання iframe може збільшувати ризик атак на безпеку, таких як "clickjacking";

7) проблеми з адаптивним дизайном: Iframe може викликати проблеми при розробці адаптивних веб-сайтів через їх фіксовану ширину та висоту;

8) обмежені можливості комунікації: Iframe можуть взаємодіяти з основною сторінкою лише за допомогою postMessage API, що дещо обмежує можливості комунікації між ними;

І навпаки ми можемо назвати певні плюси додатку single-spa:

1) покращена продуктивність: У порівнянні з iframe, Single Spa забезпечує більш ефективне завантаження компонентів, що покращує продуктивність;

2) Routing: Single Spa має вбудований маршрутизатор, який краще працює з мікрофронтендами, ніж стандартна навігація iframe;

3) Single Spa дозволяє пошуковим системам краще індексувати вміст мікрофронтендів, ніж це можливо з iframe;

4) адаптивний дизайн: Single Spa спрощує створення адаптивних мікро-фронтендів, а не має проблем з фіксованою шириною та висотою, як у iframe 5) Більше динамічності: є можливість динамічно завантажувати, монтувати і ремонтувати додатки на одній сторінці без перезавантаження, що надає більше гнучкості порівняно з iframe.

Отже вибір між використанням технології iframe або бібліотеки Single Spa при переході від монолітної архітектури до мікро-фронтенду варто робити, базуючись на конкретних вимогах і подальшому очікуванні розвитку веб-додатку. Якщо швидкість переходу є визначаючим фактором, і додаток не чекають складні технічні вимоги, iframe може бути підходящим вибором. Варто враховувати мінуси, такі як можливі проблеми з сумісністю, SEO, продуктивністю і безпекою, але в ряді випадків це може бути прийнятним компромісом. З іншого боку, якщо пріоритетом є гнучкість, продуктивність, адаптивний дизайн і добре структурований код, Single Spa може запропонувати значні переваги. Це можуть бути мікрофронтенди, розроблені з використанням різних технологій, вбудована маршрутизація і краща SEO-оптимізація. Проте, для цього потрібно більше часу та ресурсів для переходу. Отже, обидва підходи мають свої переваги та недоліки, і остаточний вибір має бути зроблений на основі індивідуальних потреб та цілей вашого веб-додатку.

ВИСНОВКИ

Як результат проходження науково-дослідної практики було проведено аналіз предмету і об’єкту дослідження. Були розглянуті і проаналізовані популярні фронтенд фреймворки і бібліотеки, а також архітектури, які пропонуються для написання веб застосунків. Також, окремо було розглянуто плюси і мінуси монолітної і мікросервісної архітектури, побудовано їх порівняння. Розглянуті найчастіші причини переходу від моноліту до мікро-фронтенду, а також методи і технології переходу від одної архітектури до іншої.

За результатами аналізу трансформації архітектурного підходу можна зробити висновки, що в зміні архітектури є як плюси так і мінуси. Створення експериментального додатку та трансформація його в мікро-фронтенд показує, що перехід від моноліту до мікро-фронтенду це доволі важкий процес, в тому випадку якщо потрібно зберегти повну структуру застосунку. Важкість переходу буде залежати від обраного способу а також від об’єму коду, складності зав’язків, структури монолітного застосунку.

Як було розглянуто у цій роботі перехід від моноліту до мікро-фронтенду має свої плюси і мінуси. Як в свій час писав Мартін Фаулер який був одним з засновників мікро-фронтенду «не треба починати новий проект з мікросервісів навіть якщо ви повністю впевненні, що майбутній застосунок буде достатньо великим, щоб виправдати такий підхід», тому можна сказати, що перехід від моноліту до мікро-фронтенду це скоріш вимушений крок для збереження функціональної цілісності і розвитку старого великого застосунку.

Тому, трансформувати чи не трансформувати моноліт в мікро-фронтенд потрібно вирішувати в кожному випадку окремо, зважуючи всі за і проти. Після прийняття позитивного рішення, знову ж таки, потрібно зважено підійти до вибору архітектури мікро-фронтендів і загального об’єднання мікро-фронтендів.

Загальна тема трансформації додатків і підходів до неї, потребує додаткового вивчення і розвитку, адже web застосунки дуже глибоко входять в теперішнє життя людини і постійно розростаються у розмірах, при цьому веб технології які використовуються для їх написання дуже динамічні і застаріння технологій це вже не справа десятиліть, а справа шести місяців.

ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАННЯ

1. monolithic architecture URL: https://www.techtarget.com/whatis/definition/monolithic-architecture/ (дата звернення: 22.01.2024).

2. Порівнюємо способи генерації сторінок: CSR, SSR, SSG, ISR. Гайд на основі стеку React. URL: https://dou.ua/forums/topic/41585/ (дата звернення: 22.01.2024).

3. Регресійне тестування. URL: https://qalight.ua/baza-znaniy/regresijne-testuvannya/ (дата звернення: 07.02.2024).

4. CoffeeScript. URL: https://uk.wikipedia.org/wiki/CoffeeScript (дата звернення: 07.02.2024).

5. Backbone.js: https://uk.wikipedia.org/wiki/Backbone.js (дата звернення: 07.02.2024).

6. Microservices vs. monolithic architecture URL: https://www.atlassian.com/microservices/microservices-architecture/microservices-vs-monolith (дата звернення: 11.02.2024).

7. Understanding the World of Micro-Frontend Development URL: https://blog.stackademic.com/understanding-the-world-of-micro-frontend-development-11cd4b5b3bab (дата звернення: 15.02.2024).

8. Що таке Angular? URL: https://Angular.tutorial.in.ua/what-is-Angular/ (дата звернення: 18.02.2024).

9. Singleton URL: https://refactoring.guru/design-patterns/singleton (дата звернення: 21.02.2024).

10. Що таке React JS? Як почати вивчати Реакт? Навички для react developer URL: https://cases.media/en/article/sho-take-react-js-yak-pochati-vivchati-reakt-navichki-dlya-react-developer (дата звернення: 01.03.2024).

11. React Hooks – огляд можливостей нового API. URL: https://dou.ua/lenta/articles/react-hooks-guide/ (дата звернення: 11.03.2024).

12. Lazy-loading feature modules. URL: https://Angular.io/guide/lazy-loading-ngmodules (дата звернення: 11.03.2024).

13. Функції високого порядку URL: https://krypton.com.ua/rozdil-3-funkczionalne-programuvannya/funkcziyi-vysokogo-poryadku/ (дата звернення: 11.03.2024).

14. <Suspense> URL: https://react.dev/reference/react/Suspense (дата звернення: 11.03.2024).

15. lazy URL: https://react.dev/reference/react/lazy (дата звернення: 11.03.2024).

16. Що таке iFrame і чому його небажано використовувати для пошукової оптимізації. URL: https://serpstat.com/uk/blog/sho-take-iframe/ (дата звернення: 17.03.2024).

17. window. URL: http://xn--80adth0aefm3i.xn--j1amh/window (дата звернення: 20.03.2024).

18. Window: postMessage() method. URL: https://developer.mozilla.org/en-US/docs/Web/API/Window/postMessage (дата звернення: 27.03.2024).

19. tailor. URL: https://github.com/zalando/tailor (дата звернення: 01.04.2024).

20. Рендеринг: різновиди, застосування, порівняльна характеристика URL: https://medium.com/@ralabs.ua (дата звернення: 08.04.2024).

21. Let’s build a Webshop out of Micro Frontends URL: https://senacor.blog/microfrontends/ (дата звернення: 10.04.2024).

22. Getting Started with single-spa URL: https://single-spa.js.org/docs/getting-started-overview/ (дата звернення: 15.04.2024).

23. Smart MonoreposFast CI URL: https://nx.dev/ (дата звернення: 15.04.2024).

24. How to Use Flux to Manage State in ReactJS - Explained with an Example URL: https://www.freecodecamp.org/news/how-to-use-flux-in-react-example/ (дата звернення: 21.04.2024).

25. WebSocket URL: https://uk.wikipedia.org/wiki/WebSocket (дата звернення: 21.04.2024).

26. Що таке Docker і навіщо він? URL: https://qagroup.com.ua/publications/shcho-take-docker-i-navishcho-vin/ (дата звернення: 21.04.2024).

27. Що таке Jenkins? URL: https://qagroup.com.ua/publications/shcho-take-jenkins/ (дата звернення: 21.04.2024).

28. Коваленко Г.С., Ребезюк Л.М. Дослідження технологій переходу від моноліту до мікро-фронтенду // Інформаційні технології в соціокультурній сфері, освіті та економіці: матеріали VIII Міжнародної науково-практичної конференції студентів і молодих учених. / М-во освіти і науки України; Київ. нац. ун-т культури і мистецтв. Київ : Видавничий центр КНУКіМ, 2024. – С.36-38.