МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
НАЦІОНАЛЬНИЙ АВІАЦІЙНИЙ УНІВЕРСИТЕТ
ФАКУЛЬТЕТ КОМП'ЮТЕРНИХ НАУК ТА ТЕХНОЛОГІЙ
КАФЕДРА КОМП'ЮТЕРНИХ ІНФОРМАЦІЙНИХ ТЕХНОЛОГІЙ

ДОПУСТИТИ ДО ЗАХИСТУ
Завідувач випускової кафедри
_____ Аліна САВЧЕНКО
«___»_____2023 р.

КВАЛІФІКАЦІЙНА РОБОТА
(ПОЯСНЮВАЛЬНА ЗАПИСКА)

ВИПУСКНИКА ОСВІТНЬОГО СТУПЕНЯ МАГІСТР
ЗА ОСВІТНЬО-ПРОФЕСІЙНОЮ ПРОГРАМОЮ
«ІНФОРМАЦІЙНІ ТЕХНОЛОГІЇ ПРОЕКТУВАННЯ»

Тема: «Вебзастосунок для оголошень міста з використанням мікросервісної архітектури»

Виконавець: Данило НАБОК

Керівник: к.т.н., доцент Олена ТОЛСТІКОВА

Нормоконтролер: к.т.н., доцент Олена ТОЛСТІКОВА

КИЇВ 2023

НАЦІОНАЛЬНИЙ АВІАЦІЙНИЙ УНІВЕРСИТЕТ

Факультет *комп'ютерних наук та технологій*
Кафедра *комп'ютерних інформаційних технологій*
Спеціальність *122 «Комп'ютерні науки»*
Освітньо-професійна програма *«Інформаційні технології проектування»*

ЗАТВЕРДЖУЮ:
завідувач кафедри КІТ
_____Аліна САВЧЕНКО
(підпис)

«____»_____2023 р.

ЗАВДАННЯ
на виконання кваліфікаційної роботи
Набока Данила Руслановича
(ПІБ випускника)

1. Тема роботи: «Вебзастосунок для оголошень міста з використанням мікросервісної архутектури» затверджена наказом ректора № 1976/ст від 29.09.2023р.

2. Термін виконання роботи: з 02 жовтня 2023 року по 31 грудня 2023 року.

3. Вихідні дані до роботи: вебзастосунок на мові програмування JS для демонстрації мікросервісної архітектури.

4. Зміст пояснювальної записки: 1. Огляд існуючих архітектурних рішень. 2. Мікросервісні бібліотеки для створення вебзастосунку. 3. Проектування та розробка вебзастосунку.

5. Перелік обов'язкового ілюстративного матеріалу: 1. Поняття мікросервісної архітектури. 2. Причини використання рішень. 3. Огляд існуючих архітектурних рішень. 4. Мікросервісні бібліотеки для створення вебзастосунку. 5. Реалізація вебзастосунку з мікросервісною архітектурою. 7. Блок-схема принципу роботи застосунку. 8. Демонстрація роботи вебзастосунку.

6. Календарний план-графік

№ з/п	Завдання	Термін виконання	Підпис керівника
1.	Аналіз предметної області та огляд аналогів. Написання 1 розділу, огляд існуючих архітектурних рішень.	02.10.2023-18.10.2023	
2.	Вибір та опис використаних технологій. Написання 2 розділу.	19.10.2023-31.10.2023	
3.	Написання 3 розділу, проектування та розробка вебзастосунку.	01.11.2023-15.11.2023	
4.	Загальне редагування та друк пояснювальної записки.	16.11.2023-20.11.2023	
5.	Проходження нормоконтролю, перепліт пояснювальної записки.	21.11.2022-16.12.2022	
6.	Розробка тексту доповіді. Оформлення графічного матеріалу для презентації.	16.12.2023-22.12.2023	

7. Дата видачі завдання 02.10.2023 р.

Керівник кваліфікаційної роботи Олена ТОЛСТІКОВА

(підпис керівника)

Завдання прийняв до виконання Данило НАБОК

(підпис випускника)

РЕФЕРАТ

Пояснювальна записка до кваліфікаційної роботи на тему: «Вебзастосунок для оголошень міста з використанням мікросервісної архітектури» містить: 86 сторінок, 46 рисунків, 20 інформаційних джерел.

Об'єкт дослідження – створення вебзастосунку з використанням мікросервісних бібліотек при проектуванні його компонентів.

Предмет дослідження – формування процесу проектування вебзастосунку та його компонентів для сайту оголошень міста.

Мета кваліфікаційної роботи – створення та дослідження мікросервісного вебзастосунку для оголошень міста та виявлення його переваг.

Методи дослідження – мова програмування OI, середовище розробки WebStorm**.**

Результати кваліфікаційної роботи рекомендується використовувати для демонстрації мікросервісної архітектури вебзастосунків, та в подальшій інтеграції у рішення компаній та підприємств.

ВЕБЗАСТОСУНОК, МІКРОСЕРВІСНА АРХІТЕКТУРА, JAVASCRIPT, WEB

ЗМІСТ

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ, СКОРОЧЕНЬ, ТЕРМІНІВ

IDE (*Integrated Development Environment*)	−	Інтегроване середовище розробки
JS (*JavaScript*)	−	Мова програмування JavaScript
URL (*Uniform Resource Locator*)	−	Уніфікований локатор ресурсів або адреса ресурсу
Мікрофронт	−	Незалежна частина проекту
Вебзастосунок	−	Розподілений застосунок, в якому клієнтом виступає браузер, а сервером - вебсервер
Архітектура вебзастосунку	−	Спосіб структурування програмної системи

ВСТУП

В епоху стрімкого технологічного розвитку та зростання вимог до сучасних веб-додатків, створення високоефективних та гнучких вебзастосунків для міст стає складним завданням, що потребує не лише ретельного вивчення сучасних технологій, але й раціонального вибору архітектурного підходу.

Дана кваліфікаційна робота присвячена дослідженню можливостей створення веб-додатку для оголошень міста, зокрема, використовуючи інноваційний підхід до архітектури - мікросервісну архітектуру. Основною метою є з'ясування, наскільки мікросервіси можуть поліпшити якість та продуктивність веб-додатка, забезпечуючи ефективний обмін інформацією та високий рівень гнучкості в розробці.

Мікросервісна архітектура, яка є основною фокусною точкою цієї роботи, визначається розбиттям додатку на окремі незалежні компоненти - мікросервіси, які можуть функціонувати і вдосконалюватися незалежно. Цей підхід відкриває широкі можливості для покращення масштабованості та робочого процесу розробки в цілому.

У роботі буде проведено аналіз популярних мікросервісних фреймворків та бібліотек, визначено їхні переваги та недоліки, а також проведено порівняльний аналіз з альтернативними архітектурними рішеннями. Окрема увага буде приділена питанню вибору між монорепозиторієм та різними репозиторіями в контексті мікросервісної архітектури.

Зазначаючи швидке зростання та актуальність використання мікросервісної архітектури в сучасному веб-розробці, дана кваліфікаційна робота вивчатиме практичні аспекти використання цього підходу у реальному проекті вебзастосунку для оголошень міста.

Отже, кваліфікаційна робота має на меті внести практичний внесок у розробку сучасних вебзастосунків, підкреслити важливість вибору архітектурного підходу та допомогти визначити оптимальний шлях для створення високоефективного та ефективного веб-додатку для оголошень міста.

Метою роботи є створення та дослідження мікросервісного вебзастосунку для оголошень міста та виявлення його переваг. Для досягнення поставленої мети необхідне вирішення наступних завдань:

1. Здійснити огляд та аналіз основних принципів мікросервісної архітектури, визначити її переваги та недоліки.
2. Визначитися з інструментами розробки.
3. Розробити повнофункціональний вебзастосунок для оголошень міста, використовуючи мікросервісну архітектуру для підготовки та розгортання невеликих, незалежних компонентів.
4. Вивчити питання впливу мікросервісної архітектури на продуктивність вебзастосунку та виконати порівння його з іншими архітектурними варіантами.

Об’єктом досліджень є формування процесу проектування вебзастосунку та його компонентів для сайту оголошень міста.

Предметом досліджень є формування процесу проектування вебзастосунку та його компонентів для сайту оголошень міста.

Актуальність теми кваліфікаційної роботи «Вебзастосунок для оголошень міста з використанням мікросервісної архітектури» ґрунтується на тому, що тема визначається необхідністю створення вебзастосунків, які не лише відповідають вимогам сучасного користувача, але й можуть швидко та ефективно адаптуватися до швидкозмінюючогося інформаційного середовища. Також актуальність даної теми обумовлена кількома ключовими аспектами. По-перше, мікросервісна архітектура визначається розбиттям великих систем на невеликі, незалежні компоненти, що полегшує розробку та підтримку. Другий аспект полягає в сприянню оптимізації ресурсів та покращенню відповідності до потреб замовника, що є ключовим для забезпечення задоволення потреб користувачів. По-третє, архітектура дозволяє вносити зміни в окремі компоненти без впливу на інші частини системи, що дозволяє вебзастосункам швидко адаптуватися до змін в потребах користувачів та ринкових умов.

Відповідно до поставленої мети роботи визначено основні **завдання дослідження**:

1. Дослідити основні принципи мікросервісної архітектури;
2. Вибрати технології та фреймворки;
3. Створити програмний код та забезпечити інтеграцію;
4. Порівняти мікросервісну фрхітектуру з іншими архітектурними підходами;
5. Оцінити зручність розробки та підтримки.

Наукова новизна роботи полягає у створенні вебзастосунку з використанням архітектури, яка дозволить швидко масштабувати проєкт під актуальні потреби ринку. Проект може послужити як основа для майбутніх проектів з такими самими потребами.

Також науковою новизною даної роботи є огляд вдосконалення способів інтеграції мікросервісів для забезпечення ефективної взаємодії. Розгляд автоматизація релізного процесу в мікросервісній архітектурі, включаючи безперервну інтеграцію та безперервну доставку, яка представляє важливий аспект для забезпечення стабільності та швидкості розробки.

Додатково, вивчення та впровадження нових інструментів розробки, таких як розширені системи моніторингу та інструменти для відлагодження коду, може значно полегшити роботу розробників та підвищити продуктивність.

Напрямок архітектурних патернів для мікросервісів також є об'єктом наукового інтересу, оскільки він може принести нові рішення для проблем, пов'язаних з розгортанням та управлінням мікросервісами.

Ці аспекти наукової діяльності в сфері мікросервісної архітектури можуть значно сприяти розвитку ефективних, безпечних та легко розширюваних вебзастосунків.

РОЗДІЛ 1
ОГЛЯД ІСНУЮЧИХ АРХІТЕКТУРНИХ РІШЕНЬ ДЛЯ СТВОРЕННЯ ВЕБЗАСТОСУНКУ

1.1. Різниця monorepo та multirepo

У сучасному світі існує багато рішень для створення сайту з використанням мікросервісної архітектури. Загалом даний підхід є актуальним лише коли проект зростає, а разом з ним зростають у розмірі та ускладнюються групи розробників програмного забезпечення. В такому випадку рішення про використання архітектури монорепо або мультирепо стає все більш важливим. Підходи мають свої плюси та мінуси, і вибір неправильного може призвести до серйозних проблем у майбутньому.

Глобально підходи поділяються на 2, це monorepo та multi repo. Але перш ніж ми заглибимося в плюси і мінуси кожного підходу, швидко розглянемо що таке архітектури monorepo і multi repo.

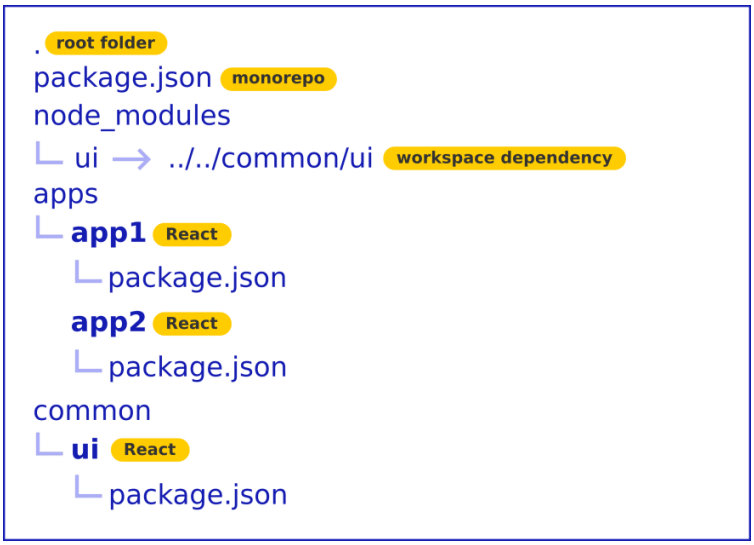
Але для початку треба розібратися з тим що таке репозиторій. Репо (скорочення від repository) або просто репозиторій— це сховище для всіх змін і файлів проекту, що дозволяє розробникам «контролювати версії» ресурсів проекту протягом усього етапу розробки.

Загалом все починається з того, що проект росте і розвивається, і постає питання щодо чіткого розділення між командами. В першу чергу це потрібно для розподільного релізного процесу, щоб кожна команда могла деплоїти свій код незалежно від іншої. Тож можна дійти висновку що все починається з монолітного резиторія.

У моноліті весь код знаходиться в одному місці, і його легко додавати функції та повторно використовувати компоненти. Усі розробники можуть робити внесок у всі частини коду за потреби. Важливо, що весь код у моноліті тестується та розгортається разом як єдиний блок, у якому все сумісне.

Але з розвитком проекту постає поняття розділення кодової бази по блокам. Тож існують такі підходи як монорепозиторій та мультирепозиторій.

Монорепо (або монорепозиторій) — це єдине сховище, яке містить увесь код певного проекту або набору пов'язаних проектів. Це означає, що весь код для всіх



служб, бібліотек і програм зберігається в одному центральному місці, як на рис.1.1.

Рис. 1.1. Структура монорепозиторія

Multirepo (або мультирепозиторій) навпаки, це архітектура з кількома репозиторіями яка складається з кількох сховищ, кожне з яких містить код для окремої служби, бібліотеки або програми, як на рис. 1.2.

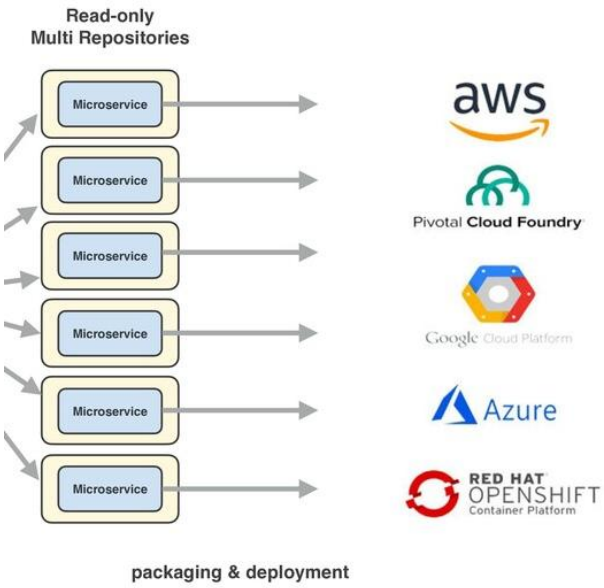


Рис. 1.2. Структура мільтирепозиторія

Різниця між цими підходами залежить від конкретних потреб проекту та організації. Ось деякі аспекти для врахування:

1. Масштаб проекту. Монорепозиторії частіше використовуються для великих проектів, де спільне керування та використання загальних засобів може призвести до ефективної спільної розробки. Мультирепозиторії можуть бути зручнішими для менших проектів або для розподіленого розвитку;

2. Залежності між частинами. Монорепозиторії забезпечують легший доступ до змін у різних частинах проекту та управління залежностями. Однак у мультирепозиторіях може бути більше гнучкості у визначенні взаємозалежностей;

3. Керування Версіями. У монорепозиторії всі версії керуються централізовано, що може бути зручно для підтримки версій та релізів. У мультирепозиторіях кожне сховище може мати власну версію, що дозволяє розділити розвиток різних частин.

Обираючи між монорепозиторієм та мультирепозиторієм, команди розробників повинні враховувати розмір проекту, потреби спільної розробки, управління версіями та залежностями, а також представлення інструментів для розробки та управління.

На жаль, немає відповіді на дискусію монорепо проти мультирепо. Зрештою, оптимальне рішення залежить від кожного окремого бізнесу.

Крім того, у міру розвитку бізнесу нові проблеми можуть перетворити вчорашнє рішення на частину сьогоднішньої проблеми. Ваш бізнес може початися з малого, але протягом десяти років, коли він стане глобальним, його архітектурні вимоги також змінюватимуться.

Розглянемо шлях Uber від monorepo до multirepo, а потім повернення до monorepo. Остаточний перехід назад мав сенс, коли Uber був достатньо багатий, щоб інвестувати в інструменти, необхідні для подолання недоліків монорепозиторійного підходу. Однак не багато компаній досягнуть такого масштабу чи позиції, тому важливо зосередитися на унікальних потребах вашого бізнесу.

Найважливішим є те, що існує багато гібридних підходів, у яких компанії використовують унікальну або кілька комбінацій сховищ. Підприємство може мати одне монорепозиторій для певного діапазону проектів або операцій, і, можливо, декілька кількох репозиторіїв для інших конкретних проектів.

1.2. Переваги та недоліки монорепозиторіїв

Як у бідь якого підходу чи рішення у монорепозиторія є плюси і мінуси. Загалом, як згадувалося раніше, монолітні репозиторії, представляють сучасний підхід до управління проектами та кодовою базою в розробці програмного забезпечення. З появою мікросервісів і модульного програмування управління взаємозалежностями може стати складнішим. Ось де в гру вступають монорепо. Але чи підходить монорепо проекту чи компаії? Відповідь зводиться до стратегічного архітектурного вибору, який впливає на інженерну культуру для подальшого розвитку.

Існує ряд вагомих причин, чому даний патерн досі існує і чому багато компаній його запровадили у своїх проектах. Цими причинами є:

1. У монорепозиторії всі проекти знаходяться в єдиному сховищі, створюючи спільну платформу для всієї компанії. Ця структура дозволяє кожному учаснику взаємодіяти з кодом та ресурсами, що полегшує пошук та використання спільних бібліотек. Такий підхід активно сприяє співпраці та використанню коду, зменшуючи технічний борг та покращуючи загальну продуктивність розробників.

2. Однак із великими можливостями монорепозиторію приходить відповідальність за зміни, які можуть вплинути на всю компанію. Тому необхідно проводити технічні узгодження перед внесенням змін та узгоджувати їх між різними командами. Особливо важливо заздалегідь обговорювати та вирішувати технічні компроміси, оскільки вони можуть мати глибокий вплив на роботу різних команд та проекти в компанії.

3. Прийняття стратегічних рішень щодо монорепозиторію вимагає не лише чіткого розуміння структури та взаємодії компонентів програмного забезпечення, але й визначення ефективної інфраструктури. Це дозволяє попередити можливі труднощі та знизити витрати на інтеграцію, забезпечуючи оптимальний розвиток та управління великим об'ємом кодової бази в єдиному сховищі;

4. Спрощує вступний процес. Коли нові працівники приєднуються до компанії, вони повинні завантажити програмний код та встановити необхідні інструменти, щоб розпочати виконання своїх завдань. Допустимо, що проект розгорнутий в різних сховищах, кожне з яких має свої власні інструкції з установки та необхідні інструменти. У такому випадку ініціалізація може виявитися вельми складною, а документація часто залишається неповною, що зобов'язує нових членів команди звертатися за підтримкою до своїх колег. Використання монорепозиторію робить цей початковий процес більш дружелюбним. Завдяки тому, що весь код та супутня документація концентруються в єдиному місці, можна легко спростити налаштування для новоприбулих у команду;

5. Можуть допомогти спростити внутрішнє та стороннє керування залежностями. Усі зазначені залежності мають одне єдине місце у кодовій базі, що сприяє вирішенню проблем, таких як конфлікти версій або проблеми з управлінням багатьма проектами, які використовують різні версії однієї залежності.

Наприклад, можна закріпити конкретну версію залежності, що спростить виявлення можливих проблем, таких як несправності чи невідповідність, під час тестування по всій кодовій базі при необхідності оновлення. Це також означає, що управління вразливостями в залежностях стає менш складним – достатньо виправити чи оновити лише одну залежність, щоб швидко та ефективно запустити всі модульні тести одночасно, покращуючи загальну надійність та можливість виявлення використання спільних бібліотек;

1. Команди діляться культурою розвитку. Незважаючи на те, що це не неможливо, з підходом монорепо стає складно надихнути унікальні субкультури серед різних команд. Оскільки вони спільно використовуватимуть той самий репозиторій, вони, швидше за все, використовуватимуть однакові методології програмування та керування та однакові інструменти розробки;

2. Складніше зламати суміжну функціональність. За допомогою моно репозиторія ми можемо налаштувати всі тести для всіх бібліотек, щоб запускати щоразу, коли будь-яка окрема бібліотека змінюється. Як наслідок, ймовірність внесення змін у деякі бібліотеки мінімізувала негативний вплив на інші бібліотеки;

3. Все вище написане призводить до ще однієї ключової переваги - масштабного рефакторингу коду та атомарних виправлень. Замість того, щоб створювати кілька запитів на злиття у декількох проектах для вдосконалення певної функції, наприклад, внутрішнього API, монорепозиторій дозволяє здійснити це зобов'язання всього одним комітом. Це спрощує процес від кількох днів або навіть місяців до кількох комітів, які ви можете розгортати з більшою впевненістю.

Наприклад, в компанії було об'єднано деякі ключові проекти в монорепозиторій, який включає кілька загальних бібліотек, використовуваних у продуктах компанії. Перенесення їх до монорепозиторію значно покращило процес впровадження

основних змін у тому, як наші різні системи використовують спільну логіку. Оскільки всі окремі проекти були в одному репозиторії, легко можна знаходити конкретний метод, який потребував рефакторингу, замість того, щоб пам'ятати всі різні сервіси, що використовують цей метод. Загалом це рішення може покращити комунікацію та співпрацю команд, знизити витрати на конфігурацію та поліпшило видимість в обох напрямках - вгору та вниз по ієрархії.

Навіть і при тому, що деякі інженерно-технічні компанії прийняли монорепозиторій, існують причини, з яких багато інших утримується від цього. Від обурення стосовно контролю доступу до необхідності використання додаткових інструментів, ось кілька потенційних недоліків у використанні монорепозиторіїв:

1. Забезпечення ефективності збірок може стати проблемою для технічних команд з великою кодовою базою. Як приклад, як Facebook, так і Google розробили свої власні інструменти для збірки (Buck і Bazel відповідно), щоб полегшити управління процесом компіляції їх величезних монорепозиторіїв. До щастя, обидва інструменти мають відкритий вихідний код, тому вони доступні всім, хто використовує монорепозиторій.

2. Але важливо враховувати, що ці інструменти для компіляції мають свої власні аспекти. Можливість мати відтворювані збірки в одному атомарному коміті в монорепозиторії дозволяє значно підвищити ефективність збірки через кешування бінарних артефактів. Це виявляється важким завданням при використанні підходів із кількома репозиторіями, оскільки вам потрібно створювати значно більше бінарних файлів (наприклад, всі можливі комбінації версій бібліотек). На жаль, початкові витрати на впровадження такої інфраструктури є досить значними;

3. Інструмент VCS також може спричинити кілька проблем. Подібно до створення конвеєрів, організації з великою кодовою базою та мільйонами комітів можуть спостерігати погіршення продуктивності git. Знову ми бачимо, що Google створив власний налаштований інструмент для цього.

4. Microsoft випустила віртуальну файлову систему (VFS) для git, щоб допомогти впоратися з перевантаженням. Це дозволяє розробникам завантажувати

лише те, що потрібно, тому їм не потрібно завантажувати монорепо повністю для розробки нових продуктів і функцій;

5. Повільніші цикли розвитку. Якщо код для бібліотеки містить критичні зміни, через які тести для залежних бібліотек не проходять, код також потрібно виправити перед об'єднанням змін.

6. Якщо ці бібліотеки залежать від інших команд, які зайняті роботою над якимось іншим завданням і не можуть (або не бажають) адаптувати свій код, щоб уникнути критичних змін і забезпечити проходження тестів, розробка нової функції може зупинитися.

7. Більше того, проект цілком може почати просуватися лише зі швидкістю найповільнішої команди в компанії. Такий результат може розчарувати членів найшвидших команд, створивши умови для того, щоб вони захотіли залишити компанію.

8. Крім того, бібліотеці потрібно буде запустити тести для всіх інших бібліотек. Що більше тестів потрібно запустити, то більше часу потрібно для їх виконання, що сповільнює швидкість повторення нашого коду;

9. Незмінені бібліотеки можуть мати нові версії. Коли ми позначаємо монорепо, всьому коду в ньому призначається новий тег. Якщо ця дія ініціює новий випуск, тоді всі бібліотеки, розміщені в репозиторії, будуть нещодавно випущені з номером версії з тегу, навіть якщо багато з цих бібліотек можуть не мати жодних змін;

10. Розгалуження складніше. Проекти з відкритим кодом мають якомога легше залучати учасників. З кількома сховищами учасники можуть перейти безпосередньо до конкретного сховища для проекту, до якого вони хочуть зробити внесок. Однак, якщо в монорепо розміщено різні проекти, учасники повинні спочатку знайти правильний проект і зрозуміти, як їхній внесок може вплинути на всі інші проекти.

Зрештою, було б неправильно розглядати monorepo як «хороший» чи «поганий» інструмент для команд розробників. Натомість зрозуміло, що існує кілька

вагомих причин, чому вони можуть допомогти підняти інженерну частину організацію на новий рівень, а також кілька закономірних проблем, пов’язаних з їх використанням.

1.3. Переваги та недоліки мультирепозиторного підходу

Мультирепозиторний підхід до розробки програмного забезпечення передбачає, що різні проекти або компоненти великої програми знаходяться у власних сховищах та керуються окремо. Кожен з проектів або компонентів має своє власне сховище для контролю версій, що дозволяє різним командам незалежно працювати над ними. Цей підхід відрізняється від монорепозиторію, де весь код зберігається в одному репозиторії. Використання кількох сховищ часто використовується для встановлення чітких меж між проектами, дозволяючи командам автономно працювати над різними частинами системи. Також це потенційно зменшує можливі конфлікти, які можуть виникнути при внесенні змін.

Але цей підхід також має свої сильні та слабкі сторони. Серед сильних сторін можна виділити:

1. Незалежне керування версіями бібліотеки. Під час позначення репозиторію всій його кодовій базі присвоюється тег «новий». Оскільки в репозиторії міститься лише код певної бібліотеки, бібліотеку можна позначати тегами та версії незалежно від усіх інших бібліотек, розміщених деінде.

Наявність незалежної версії для кожної бібліотеки допомагає визначити дерево залежностей для програми, дозволяючи нам налаштувати, яку версію кожної бібліотеки використовувати;

2. Незалежні випуски послуг. Оскільки репозиторій містить лише код для певної служби і нічого більше, він може мати власний цикл розгортання, незалежно від будь-якого прогресу, досягнутого в додатках, які звертаються до нього;

3. Як зазначено вище, зростаючий обсяг кодування може спричинити проблеми (наприклад, як Git може сповільнюватися під час збільшення масштабу сховища). Без інструментів для вирішення цієї проблеми компанія може перейти до кількох

сховищ, щоб зробити роботу більш керованою. Це те, що Uber вирішив зробити, коли він виріс із малої компанії середнього розміру;

4. Завдяки мультирепозиторному підходу, розробнику дуже зрозуміло, якими частинами коду він володіє. Може бути простіше зосередитися на конкретних цілях вашого проекту. Рівні доступу можуть бути чітко визначені самою структурою сховища;

5. Якщо розробник випадково вносить помилку в код, це вплине лише на код у цьому конкретному сховищі, а не на весь бізнес. Помилка з меншою ймовірністю призведе до зупинки розвитку.

6. Більш обмежені, логічно визначені репозиторії можуть бути безпечнішими для менш досвідчених розробників. Так само, якщо між командами розробників немає належної культури співпраці та діалогу, більш чіткий підхід може бути більш плавним.

Але як у будь якого підходу, є завжди зворотна сторона медалі. Тож серед мінусів мультирепозиторного підходу можна виділити:

1. Бібліотеки необхідно постійно повторно синхронізувати. У разі випуску нової версії бібліотеки, що містить критичні зміни, бібліотеки, які залежать від цієї бібліотеки, потрібно буде адаптувати, щоб почати використовувати останню версію. Якщо цикл випуску бібліотеки швидший, ніж цикл її залежних бібліотек, вони можуть швидко не синхронізуватися одна з одною.

2. Командам потрібно буде постійно надолужувати згаяне, щоб використовувати останні випуски інших команд. Зважаючи на те, що різні команди мають різні пріоритети, досягти цього іноді може виявитися важко.

3. Отже, команда, яка не зможе наздогнати згаяне, може в кінцевому підсумку дотримуватись застарілої версії залежної бібліотеки. Цей результат матиме наслідки для програми (з точки зору безпеки, швидкості та інших міркувань), а розрив у розробці між бібліотеками може тільки збільшитися.

4. Команди можуть фрагментуватися. Коли різним командам не потрібно взаємодіяти, вони можуть працювати окремо. У довгостроковій перспективі це може

призвести до того, що команди створюватимуть свої субкультури всередині компанії, наприклад, використовуватимуть різні методології програмування чи управління або використовують різні набори інструментів розробки.

5. Якщо якомусь члену команди врешті-решт знадобиться працювати в іншій команді, він може зазнати невеликого культурного шоку та навчитися виконувати свою роботу по-новому.

6. Величезною проблемою розробки програмного забезпечення є дублювання коду. Кожну копію функції потрібно підтримувати, і що ще гірше – розбіжності між ними можуть бути причиною помилок, які важко знайти.

7. При використанні підходу з кількома репозиторіями шанси, що це станеться, набагато вищі. IDE не тільки бачить лише частину коду кожного разу (тому важче знаходити використання, шукати схожі методи тощо), але може бути фактично неможливо викликати код з іншого репозиторію.

8. Деякі компанії обходять це, розміщуючи спільний пакет у ще одному репозиторії, який завантажується скрізь. Це може призвести до того, що цей модуль стане роздутим кодом, який використовується в купі різних місць, що спричинить точний безлад, який передбачає структура мультирепо. вирішити.

9. Невеликі зміни можна внести в межах одного репозиторію, але чим більша функція, тим більша ймовірність, що вам доведеться торкнутися кількох сховищ, щоб її вирішити. Це може бути складно – IDE не може допомогти знайти місця, які потрібно змінити, тому може знадобитися багато зворотних переходів через конвеєр CI.

10. Навіть після зміни всіх репозиторіїв за потреби ви можете зіткнутися з проблемою атомарної модифікації – неможливо гарантувати закріплення для всіх сховищ одночасно, тому ви можете отримати ситуацію, коли деякий частковий набір змін було зафіксовано, але інші застрягли. Це може бути пов’язано з проблемами CI, конфліктами злиття або будь-якою іншою проблемою.

1.4. Синтетичні рішення для проектів

Обидва підходи зрештою мають справу з тією самою метою: управління кодовою базою. Отже, вони обидва повинні вирішувати однакові завдання, включаючи керування випусками, сприяння співпраці між членами команди, вирішення проблем, виконання тестів тощо.

Їхня головна відмінність стосується часу, коли члени команди приймають рішення: або заздалегідь для монорепо, або далі для мультирепо.

Оскільки версії всіх бібліотек у мульти-сховищі створюються незалежно, команда, яка випускає бібліотеку з критичними змінами, може зробити це безпечно, призначивши новий основний номер версії останньому випуску. Інші групи можуть використовувати для своїх залежних бібліотек стару версію та переходити на нову після адаптації коду.

Цей підхід залишає рішення про те, коли адаптувати всі інші бібліотеки, кожній відповідальній команді, яка може зробити це в будь-який час. Якщо вони зроблять це надто пізно і будуть випущені нові версії бібліотек, усунути розрив між бібліотеками стане дедалі складніше.

Отже, хоча одна команда може швидко й часто повторювати свій код, інші команди можуть виявитися неспроможними наздогнати, що зрештою створює бібліотеки, які розходяться.

З іншого боку, у середовищі монорепо ми не можемо випустити нову версію однієї бібліотеки, яка порушує роботу іншої бібліотеки, оскільки їхні тести не пройдуть. У цьому випадку перша команда повинна спілкуватися з другою командою, щоб внести зміни.

Цей підхід змушує команди адаптувати всі бібліотеки разом, коли потрібно змінити одну бібліотеку. Усі команди змушені спілкуватися одна з одною та спільно знаходити рішення.

У результаті перша команда не зможе виконувати ітерацію так швидко, як вона хоче, але код у різних бібліотеках ніколи не почне розходитися.

Підсумовуючи, підхід із кількома репо може допомогти створити культуру «швидко рухайся та ламай речі» серед команд, де спритні незалежні команди можуть створювати результати зі своєю швидкістю. Натомість підхід монорепо сприяє культурі обізнаності та турботи, коли команди не повинні залишатися позаду, щоб самостійно вирішити проблему.

Тож якщо ми не можемо вирішити, чи використовувати підходи мультирепо чи монорепо, існує також проміжний підхід: використовувати кілька репозиторіїв і застосувати якийсь інструмент для їх синхронізації, що робить його схожим на монорепо, але з більшою гнучкістю.

Мета є одним із таких інструментів. Він організовує кілька сховищ у підкаталогах і забезпечує інтерфейс командного рядка, який виконує ту саму команду для всіх них одночасно.

Мета-репозиторій містить інформацію про те, які сховища складають проект. Клонування цього репозиторію через мета рекурсивно клонує всі необхідні репозиторії, що полегшить новим членам команди негайно розпочати роботу над своїми проектами.

Інший підхід полягає в управлінні кодом через монорепо для розробки, але копіювання коду кожної бібліотеки в окремий репозиторій для розгортання.

Ця стратегія є поширеною в екосистемі PHP, оскільки Packagist (основне сховище Composer) вимагає URL-адресу загальнодоступного сховища для публікації пакета, і неможливо вказати, що пакет знаходиться в підкаталозі сховища.

Враховуючи обмеження Packagist, проекти PHP все ще можуть використовувати монорепо для розробки, але вони повинні використовувати підхід із кількома репо для розгортання.

Щоб досягти цього перетворення, ми можемо виконати сценарій з git subtree split або використати один із доступних інструментів, які виконують ту саму логіку:

- Git Subtree Splitter;
- Git Subsplit;
- Дія GitHub для Monorepo Split.

Також приведу приклад декількох великий компаній і підходи які вони використовують у своїх продуктах.

- Google, Facebook, Twitter і Uber публічно виступили за підхід монорепо. Microsoft запускає найбільше на планеті моносховище Git для розміщення вихідного коду операційної системи Windows.
- з іншого боку, Netflix, Amazon і Lyft є відомими компаніями, які використовують підхід кількох репо.
- що стосується гібридного режиму «poly-as-mono», Android оновлює кілька репозиторіїв, які керуються як моносховище.
- що стосується гібридного моно-як-полі, Symfony зберігає код для всіх своїх компонентів у монорепо. Вони розділили його на незалежні сховища для розгортання.

Огляд використання мікросервісної архітектури великими компаніями демонструє популярність та успішність цього підходу у сучасному програмному забезпеченні. Компанії, такі як Netflix, Amazon, Uber та Google, впроваджують мікросервіси для досягнення гнучкості, швидкості розгортання та масштабованості своїх продуктів.

Цей підхід дозволяє розбити складний додаток на менші, автономні компоненти, які можуть незалежно розвиватися та масштабуватися. Використання контейнеризації та інструментів, таких як Kubernetes, забезпечує ефективне управління цими мікросервісами.

Приклади успішного впровадження мікросервісної архітектури яскраво свідчать про те, що цей передовий підхід до розробки програмного забезпечення стає необхідним та ключовим елементом стратегії для великих технологічних компаній. Цей підхід виявляється особливо ефективним, допомагаючи їм не лише адаптуватися до стрімкого темпу змін в сучасному індустріальному ландшафті, але й забезпечувати високий рівень стабільності, гнучкість та зручність у процесі розробки.

Одним із найяскравіших прикладів успішної реалізації мікросервісів, як було вказано вище, є компанія Netflix. Замість традиційного монолітного підходу до розробки програм, Netflix перейшла до архітектури мікросервісів, розбивши свою систему на невеликі, автономні модулі, кожен з яких відповідає за конкретну функціональність. Це дозволило компанії гнучко реагувати на зміни в індустрії стрімінгового відео, додавати нові функції та покращення без значних затримок у розробці та впровадженні.

Ще одним прикладом є Airbnb, яка успішно перейшла на мікросервісну архітектуру для оптимізації своїх онлайн-платформ та покращення користувацького досвіду. Розбивши великий моноліт на невеликі, незалежні частини, компанія змогла ефективніше масштабувати та вдосконалювати окремі компоненти, забезпечуючи при цьому стабільність та швидкість роботи системи.

Впровадження мікросервісної архітектури дозволяє підвищити рівень автономності окремих сервісів, полегшує взаємодію між різними частинами системи та сприяє ефективній роботі команд розробників. Цей підхід стає драйвером інновацій та дозволяє компаніям бути більш адаптивними та конкурентоспроможними в умовах постійних змін на ринку технологій.

ВИСНОВКИ ДО РОЗДІЛУ 1

Огляд існуючих архітектурних рішень для створення вебзастосунків вказує на розмаїття та динаміку області веб-розробки. За допомогою аналізу різних підходів можна визначити ключові тенденції та вибрати оптимальні рішення для конкретного проекту.

Розгляд монолітних архітектур підкреслив їхні переваги у вигляді простоти та зручності розробки на початкових етапах, але одночасно вказав на обмеження в масштабованості та гнучкості.

Мікросервісна архітектура, у свою чергу, представляє сучасний підхід, який дозволяє розбити додаток на невеликі, автономні компоненти, забезпечуючи гнучкість в розвитку та підтримці. Це особливо актуально в умовах швидкозмінюваних вимог та високих очікувань користувачів.

Врахування паттернів SPA та PWA вказує на значущий вплив клієнтської сторони на користувацький досвід. Використання односторінкових додатків дозволяє зменшити час завантаження та покращити відзивчивість.

Загальний висновок полягає в тому, що вибір архітектури вебзастосунку повинен залежати від конкретних потреб проекту. При цьому слід брати до уваги такі фактори, як масштабованість, швидкодія, зручність розробки та підтримки. Враховуючи зростаючу складність вимог до вебзастосунків, важливо обирати ті технології та архітектурні підходи, які забезпечать оптимальну роботу та задоволення користувачів.

РОЗДІЛ 2
МІКРОСЕРВІСНІ БІБЛІОТЕКИ ДЛЯ СТВОРЕННЯ ВЕБЗАСТОСУНКУ

2.1 Мікросервіси у світі фронтенда

Для початку потрібно розібратися з тим, а що таке мікросервіси в світі фронтенда. Термін Micro Frontends вперше з'явився в ThoughtWorks Technology Radar наприкінці 2016 року. Він розширює поняття мікросервісів на світ frontend. Поточна тенденція полягає у створенні багатофункціональної та потужної програми для браузера, такої ж односторінкової програми, яка розташована на архітектурі мікросервісу. З часом інтерфейсний рівень, який часто розробляє окрема команда, зростає, і його стає важче підтримувати. Це те, що ми називаємо Frontend Monolith.

Ідея Micro Frontends полягає в тому, щоб розглядати веб-сайт або вебзастосунок як сукупність функцій, які належать незалежним командам. Кожна команда має окрему сферу діяльності або місію, якою вона займається та на якій спеціалізується. Команда є міжфункціональною та розвиває свої функції від кінця до кінця, від бази даних до інтерфейсу користувача.

Проте ця ідея не нова. Вона має багато спільного з концепцією самодостатніх систем. У минулому такі підходи називалися Frontend Integration for Verticalised Systems. Але Micro Frontends — це явно більш дружній і менш громіздкий термін.

Головна ідея мікрофронтів полягає в тому, щоб розбити моноліт інтерфейсу на менші, більш керовані частини. Кожна команда може наскрізно володіти своїми власними функціями, працювати у власній кодовій базі, незалежно випускати версії, постійно забезпечувати невеликі поступові оновлення, а також інтегруватися з іншими командами через API, щоб вони могли разом створювати та керувати сторін

ками та програмами.

Концепція мікроінтерфейсів розвивалася з часом у відповідь на виклики, пов’язані з розробкою та обслуговуванням великих і складних вебзастосунків. Ось короткий огляд історії та розвитку архітектури мікроінтерфейсу:

1. Монолітні інтерфейсні програми (до 2010-х): на початку веб-розробки багато програм створювалися як моноліти, де вся інтерфейсна логіка була тісно інтегрована в єдину кодову базу. Незважаючи на те, що цей підхід був простим, його стало складно масштабувати та підтримувати, оскільки додатки зростали в розмірі та складності;

2. Розвиток односторінкових додатків (SPA) (2010-ті): З появою фреймворків JavaScript, таких як Angular, React і Vue.js, відбувся зсув до створення односторінкових додатків (SPA). SPA дозволили отримати більш динамічний і чутливий інтерфейс користувача, завантажуючи лише необхідний вміст замість оновлення всієї сторінки. Однак із зростанням SPA виникли проблеми, пов’язані з масштабованістю та зручністю обслуговування;

3. Архітектура мікросервісів (2010-ті роки): прийняття архітектури мікросервісів на серверній частині започаткувало ідею поділу великих монолітних програм на менші незалежні сервіси. Цей підхід забезпечив такі переваги, як масштабованість, гнучкість і легкість розробки та розгортання для окремих служб;

4. Поява мікроінтерфейсів (середина 2010-х): виклики, з якими зіткнулися команди, що працюють над великими SPA, призвели до дослідження архітектур мікроінтерфейсів. Концепція запозичила принципи з мікросервісів і застосувала їх до інтерфейсу. Розбивши інтерфейс користувача на менші блоки, які можна розгортати незалежно (мікроінтерфейси), групи розробників могли б працювати більш автономно та швидше випускати функції;

5. Прийняття промисловості та інструменти (кінець 2010-х – теперішній час): коли переваги мікроінтерфейсів стали очевидними, з’явилися різні інструменти та фреймворки для полегшення їх впровадження. Ці інструменти включають Module Federation у Webpack, single-spa та інші бібліотеки/фреймворки, які підтримують створення та інтеграцію мікроінтерфейсів;

6. Зусилля щодо стандартизації (на даний момент): підхід мікроінтерфейсу отримав визнання в спільноті веб-розробників, що призвело до дискусій навколо стандартизації найкращих практик і шаблонів. Практикуючі галузі та організації продовжують ділитися своїм досвідом і сприяти зростанню обсягу знань про мікроінтерфейси.

Наразі можна виділити такі основні характеристики мікроінтерфіейсів:

1. Децентралізація: мікроінтерфейси дозволяють різним командам або розробникам незалежно працювати над окремими частинами інтерфейсу користувача. Кожен мікроінтерфейс можна розробити за допомогою різних технологій або фреймворків;

2. Незалежне розгортання: мікроінтерфейси можна розгортати незалежно один від одного. Це дозволяє командам випускати оновлення або нові функції для певної частини інтерфейсу користувача, не впливаючи на всю програму;

3. Ізоляція: кожен мікроінтерфейс ізольовано від інших, що зменшує ризик виникнення проблем або конфліктів однієї частини програми з іншою. Ізоляція також дозволяє окремо тестувати та налагоджувати;

4. Технологічний агностицизм: мікроінтерфейси дозволяють використовувати різні технології або фреймворки для різних частин програми. Це особливо корисно в ситуаціях, коли різні команди мають досвід у різних технологіях;

5. Масштабованість: підхід мікроінтерфейсу підтримує масштабованість, дозволяючи командам самостійно масштабувати свої зусилля. Він також узгоджується з архітектурою мікросервісів на серверній частині, створюючи цілісну наскрізну програму на основі мікросервісів.

2.2 Найбільш поширені рішення для створення мікрофронтів

Існує багато підходів до мікрофронтів, від інтелектуальної інтеграції компонентів під час збирання до інтеграції під час виконання за допомогою спеціальних маршрутизаторів. Тож найбільш відомі та зручні рішення для створення мікрофронтів є:

1. Bit.

Бібліотека Bit.js або просто "Bit" - це інструмент для керування та ре використання компонентів і бібліотек JavaScript. Вона спроектована з метою полегшення розробки за принципами ре використання коду та підвищення продуктивності розробників у великих проектах.

Основні особливості та можливості Bit.js включають:

− компоненти: Bit допомагає виділити різні частини вашого коду як незалежні компоненти. Це може бути все, від React компонентів до JavaScript функцій;

− керування залежностями: Bit дозволяє керувати залежностями між компонентами та бібліотеками. Ви можете визначити, які компоненти використовуються в інших частинах вашого проекту;

− ревикористання: Bit робить легким ре використання коду в інших проектах або частинах того ж проекту. Ви можете легко імпортувати компоненти та використовувати їх безпосередньо в вашому коді;

− інтеграція з Git: Bit інтегрується з Git, що дозволяє зберігати ваші компоненти та їхню історію версій у Git-репозиторії;

− командний рядок та API: Bit має командний рядок та API, які дозволяють вам взаємодіяти з бібліотекою та виконувати операції керування компонентами з командного рядка;

− хостинг компонентів: Bit також надає можливість хостити ваші компоненти на хмарних серверах для спільного використання та спільної роботи.

Bit допомагає розробникам ефективно керувати та ре використовувати код, роблячи робочий процес більш організованим і прозорим. Вона корисна особливо у великих та розподілених командах, які працюють над складними проектами на JavaScript.

Цю бібліотеку можна легко встановити за допомогою Node package manager та дуже легко імплементувати завдяки обширній документації на їх офіційному сайті.

2. Webpack 5 and Module Federation.

Webpack 5 і Module Federation - це два ключові інструменти для розробки сучасних вебзастосунків на JavaScript, які дозволяють побудувати масштабовані та розділені застосунки. Давайте розглянемо їх основні функції та можливості:

Webpack - це популярний інструмент для збирання та пакування ресурсів вебзастосунків, таких як JavaScript файли, CSS, зображення та інше. Версія 5 була важливим оновленням і включала в себе численні поліпшення та нові функції:

− Module Federation: Webpack 5 вперше представив концепцію Module Federation, яка дозволяє розділяти код між різними додатками та мікросервісами у великих вебзастосунках. Це дозволяє завантажувати модулі з інших бандлів динамічно, що полегшує підтримку розділених додатків та підвищує продуктивність.

− підтримка ESM (ECMAScript Modules): Webpack 5 підтримує імпорти та експорти ECMAScript Modules, що робить його сумісним із стандартом модулів в JavaScript;

− зменшена величина бандлів: Завдяки ряду оптимізаційних поліпшень, Webpack 5 може створювати менші бандли, що прискорює завантаження вебзастосунків;

− підтримка кешування: Додана можливість кешування, що зменшує час перезбирання коду при повторних зборках;

− Module Federation - це концепція, яка дозволяє створювати розділені вебзастосунки, які можуть спільно використовувати компоненти та модулі між собою.

Основні можливості Module Federation включають:

− динамічне завантаження модулів: Ви можете динамічно завантажувати модулі з інших додатків або мікросервісів під час виконання, що дозволяє ефективно розділяти код та ресурси;

− розділений робочий процес: Модулі можуть працювати як окремі сутності, незалежно від інших частин додатку. Це дозволяє командам робити зміни та вдосконалення без впливу на інші частини додатку;

− зовнішнє виконання: Module Federation може виконувати залежності зовнішньо, що дозволяє завантажувати код з інших додатків через HTTP;

− загальні бібліотеки: Ви можете спільно використовувати бібліотеки та модулі між різними додатками без дублювання коду.

Module Federation є потужним інструментом для створення розділених вебзастосунків та мікросервісів у сучасному веб-розробці. Подібно до Webpack 5, він спрощує розробку та підвищує продуктивність при роботі над складними проектами.

3. Single SPA.

Single SPA (Single-SPA) - це бібліотека для створення мікросервісних або мікрофронтенд додатків, яка дозволяє розробляти і впроваджувати незалежні частини веб-додатків (мікрофронтенди) та об'єднувати їх у єдиний вебзастосунок. Single SPA робить можливим створення вебзастосунків, які складаються з різних частин, написаних на різних фреймворках або бібліотеках та розгорнутих на різних серверах.

Основні можливості та концепції Single SPA включають:

− мікрофронтенди: За допомогою Single SPA ви можете створювати незалежні мікрофронтенди, які можуть бути написані на різних фреймворках, таких як React, Angular, Vue, або навіть чистому JavaScript;

− динамічне завантаження: Мікрофронтенди завантажуються динамічно в залежності від URL або інших умов, що дозволяє оптимізувати завантаження ресурсів та зменшити час завантаження сторінок;

− роутінг та навігація: Single SPA надає засоби для роутінгу та навігації між мікрофронтендами, дозволяючи створювати єдиний вебзастосунок зі змішаним вмістом;

− спільна структура та стилі: Мікрофронтенди можуть спільно використовувати загальну структуру та стилі, що дозволяє підтримувати спільну корпоративну айдентику;

− підтримка серверного рендерингу: Single SPA підтримує серверний рендеринг для мікрофронтендів, що полегшує оптимізацію для пошукових систем і забезпечує швидке відображення на стороні клієнта;

− життєвий цикл компонентів: Single SPA надає методи та події для керування життєвим циклом мікрофронтендів, включаючи їх ініціалізацію та знищення.

Single SPA дозволяє створювати сучасні вебзастосунки, які складаються з розділених та незалежних частин, що можуть бути розроблені і підтримані окремо. Це дозволяє покращити масштабованість проектів та спростити розробку великих додатків, які включають в себе різні фреймворки та технології.

4. Systemjs.

SystemJS - це інструмент для збирання та завантаження модулів JavaScript у браузері. Він є одним із реалізацій специфікації систем модулів ECMAScript (ES6 та пізніше) та дозволяє динамічно завантажувати модулі під час виконання вебзастосунка. Основна ідея SystemJS - це можливість працювати з модулями та їх залежностями, навіть якщо вони не включені в основний бандл додатка.

Основні функції та можливості SystemJS включають:

− завантаження модулів: SystemJS дозволяє завантажувати модулі з сервера чи іншого джерела на льоту. Це забезпечує можливість динамічно завантажувати модулі та ресурси в залежності від умов;

− підтримка стандарту модулів ES6: SystemJS відповідає стандарту модулів ECMAScript 6 (ES6), дозволяючи використовувати синтаксис import та export;

− розділена система: Ви можете розділити свій код на модулі та завантажувати їх за запитом. Це зменшує розмір основного бандла та поліпшує продуктивність завантаження;

− підтримка різних форматів модулів: SystemJS підтримує різні формати модулів, такі як CommonJS, AMD, System.register, ES6 та інші. Це дозволяє використовувати модулі з різних джерел;

− плагіни: Ви можете використовувати плагіни для завантаження ресурсів, таких як CSS, JSON, HTML і багато інших, як модулі;

− кешування завантажених модулів: SystemJS зберігає завантажені модулі у кеші, що дозволяє покращити продуктивність при подальших завантаженнях.

Хоча SystemJS має свої можливості, варто враховувати, що на сьогоднішній день більшість сучасних проектів використовують інші інструменти для збирання та завантаження модулів, такі як Webpack або Rollup, які надають більше функціональності та гнучкості для розробки та оптимізації вебзастосунків.

5. Piral.

Piral - це відкрита бібліотека та фреймворк для розробки мікрофронтендів (micro frontends) у великих монолітних додатках. Вона дозволяє розробляти незалежні частини вебзастосунків, які можуть бути написані на різних технологіях та фреймворках, і об'єднувати їх в єдиний вебзастосунок з відокремленою розробкою та релізами. Piral була розроблена з врахуванням концепції мікросервісної архітектури, але для веб-інтерфейсів.

Основні функції та можливості Piral включають:

− мікрофронтенди: Piral дозволяє створювати незалежні мікрофронтенди, які можуть бути написані на різних фреймворках, таких як React, Angular, Vue або навіть на чистому JavaScript;

− динамічне завантаження: Мікрофронтенди завантажуються динамічно, що зменшує час завантаження сторінок та поліпшує продуктивність;

− розділена розробка і релізи: Кожен мікрофронтенд може бути розроблений та випущений окремо від інших, що спрощує процес розробки та підтримки;

− розширення та плагіни: Piral дозволяє створювати розширення та використовувати плагіни для додавання функціональності до мікрофронтендів;

− шаблони та стилі: Piral надає можливість використовувати спільні шаблони та стилі для мікрофронтендів;

− підтримка серверного рендерингу: Piral підтримує серверний рендеринг для мікрофронтендів, що допомагає оптимізувати відображення на стороні клієнта та пошукових систем.

Piral дозволяє розробникам створювати масштабовані та розділені застосунки, що складаються з різних незалежних частин. Це корисно в сучасному веб-розробці, де різні команди можуть працювати над окремими функціональними блоками та випускати їх незалежно. Piral спрощує цей процес та дозволяє створювати більш підтримувані та масштабовані додатки.

6. Qiankun.

Qiankun - це бібліотека та фреймворк для створення мікрофронтендів (micro frontends) у вебзастосунках. Вона була розроблена командою Ant Design, яка входить в корпорацію Alibaba, та використовується для побудови великих та складних додатків, розроблених з використанням різних фреймворків і технологій.

Основні можливості та функції Qiankun включають:

− мікрофронтенди: Qiankun дозволяє розробникам створювати незалежні мікрофронтенди, які можуть бути розроблені на різних фреймворках, таких як React, Vue, Angular, або навіть на чистому JavaScript;

− динамічне завантаження: Мікрофронтенди завантажуються динамічно в залежності від потреби, що допомагає зменшити час завантаження сторінок та покращити продуктивність;

− розділена розробка та релізи: Кожен мікрофронтенд може бути розроблений та випущений окремо від інших, що спрощує процес розробки та підтримки;

− роутінг та навігація: Qiankun надає інструменти для роутінгу та навігації між мікрофронтендами, дозволяючи створювати єдиний вебзастосунок з відокремленою розробкою та релізами;

− зовнішнє виконання: Мікрофронтенди можуть виконуватися зовнішньо, що дозволяє завантажувати їх код з інших джерел через HTTP;

− загальні бібліотеки та стилі: Мікрофронтенди можуть спільно використовувати загальні бібліотеки та стилі, що дозволяє підтримувати спільну корпоративну айдентику;

− підтримка серверного рендерингу: Qiankun підтримує серверний рендеринг для мікрофронтендів, що допомагає оптимізувати відображення на стороні клієнта та пошукових систем.

Qiankun - це потужний інструмент для розробки мікрофронтендів, який допомагає створювати масштабовані та розділені застосунки. Він використовується багатьма компаніями, щоб спростити розробку та підтримку великих додатків, які включають в себе різні фреймворки та технології.

7. Luigi.

Luigi - це відкрита бібліотека та фреймворк для створення мікрофронтендів (micro frontends) у вебзастосунках. Цей фреймворк розроблений командою SAP і використовується для побудови великих та складних вебзастосунках, які складаються з незалежних частин (мікрофронтендів), написаних на різних технологіях та фреймворках.

Основні можливості та функції Luigi включають:

− мікрофронтенди: Luigi дозволяє розробникам створювати незалежні мікрофронтенди, які можуть бути написані на різних фреймворках, таких як React, Angular, Vue, або навіть на чистому JavaScript;

− динамічне завантаження: Мікрофронтенди завантажуються динамічно під час виконання, що зменшує час завантаження сторінок та покращує продуктивність;

− розділена розробка і релізи: Кожен мікрофронтенд може бути розроблений та випущений окремо від інших, що спрощує процес розробки та підтримки;

− загальна навігація і роутінг: Luigi надає інструменти для роутінгу та навігації між мікрофронтендами, дозволяючи створювати єдиний застосунок з відокремленою розробкою та релізами;

− підтримка серверного рендерингу: Luigi підтримує серверний рендеринг для мікрофронтендів, що допомагає оптимізувати відображення на стороні клієнта та пошукових систем;

− загальні бібліотеки і стилі: Мікрофронтенди можуть спільно використовувати загальні бібліотеки та стилі для підтримки спільної корпоративної айдентики;

− розширення і плагіни: Luigi дозволяє розробникам створювати розширення та використовувати плагіни для додавання функціональності до мікрофронтендів.

Luigi - це потужний інструмент для створення мікрофронтендів у великих та складних вебзастосунках. Він дозволяє розробникам покращити масштабованість та підтримуваність проектів, які включають в себе різні фреймворки та технології.

8. NX.

(Nrwl Extensions) — це набір потужних інструментів розробки та бібліотек для створення масштабованих і ефективних додатків Angular, React і Node.js. Він розроблений і підтримується Nrwl (Narwhal Technologies), компанією, яка зосереджується на інструментальних і консалтингових послугах для розвитку підприємства з Angular і Nx.

Ось ключові функції та аспекти NX:

− розробка Monorepo: NX сприяє використанню монорепозиторіїв (monorepos), де кілька проектів і бібліотек співіснують в одному репозиторії з контрольованими версіями. Цей підхід допомагає ефективніше керувати спільним кодом, залежностями та конфігураціями;

− розширений CLI (інтерфейс командного рядка): NX надає інтерфейс командного рядка, який розширює можливості Angular CLI. Він містить додаткові

команди для керування monorepos, генерації коду, виконання тестів тощо. NX CLI спрощує звичайні завдання розробки та забезпечує дотримання найкращих практик;

− генерація коду: NX містить потужні можливості генерації коду, які допомагають створювати проекти, компоненти, служби тощо. Це прискорює процес розробки та забезпечує узгодженість у кодовій базі;

− Architect API: NX представляє концепцію Architect API, яка дозволяє розробникам визначати та запускати власні команди у своїх проектах. Це особливо корисно для автоматизації складних робочих процесів та інтеграції з різними інструментами збірки;

− графік залежностей: NX надає візуальне представлення графіка залежностей проекту, що полегшує розуміння зв’язків між різними проектами та бібліотеками. Це допомагає оптимізувати процеси збірки та тестування шляхом перебудови лише пошкоджених частин кодової бази;

− ефективні процеси збирання та тестування: NX оптимізує процеси збирання та тестування, використовуючи розумний механізм кешування. Він лише перебудовує та повторно тестує ті частини кодової бази, які змінилися, що призводить до швидшого циклу розробки;

− підтримка кількох інтерфейсних і бекенд-фреймворків: хоча NX бере свій початок у розробці Angular, він підтримує інші інтерфейсні фреймворки, такі як React. Крім того, він розширює свої можливості на серверну розробку за допомогою підтримки програм Node.js;

− модулі та інтеграції: NX підтримує різноманітні плагіни та інтеграцію з популярними інструментами та службами. Це включає підтримку платформ безперервної інтеграції (CI), редакторів коду та бібліотек сторонніх розробників;

− спільнота та документація: NX має активну спільноту, і Nrwl надає вичерпну документацію та посібники для розробників, які використовують NX у своїх проектах. Спільнота часто ділиться найкращими практиками, порадами та розширеннями.

NX здобув популярність у спільноті розробників завдяки своїй зосередженості на ефективному управлінні монорепо, генерації коду та оптимізації процесів збірки та тестування. Це особливо цінно для великих корпоративних додатків, де модульна розробка та масштабованість є критичними.

2.3. Опис мови програмування для створення мікрофронтів

Загалом найбільш розповсюдженою та єдиною мовою програмування для створення динамічних вебсторінок є JavaScript. Тому трохи детальніше про нього.

JavaScript - це динамічна, об'єктно-орієнтована прототипна мова програмування. Реалізація стандарту ECMAScript. Найчастіше використовується для створення сценаріїв вебсторінок, що надає можливість на боці клієнта (пристрої кінцевого користувача) взаємодіяти з користувачем, керувати браузером, асинхронно обмінюватися даними з сервером, змінювати структуру та зовнішній вигляд вебсторінки.

JavaScript класифікують як прототипну (підмножина об'єктно-орієнтованої), скриптову мову програмування з динамічною типізацією. Окрім прототипної, JavaScript також частково підтримує інші парадигми програмування (імперативну та частково функціональну) і деякі відповідні архітектурні властивості, зокрема: динамічна та слабка типізація, автоматичне керування пам'яттю, прототипне наслідування, функції як об'єкти першого класу.

Історія створення цієї мови досить обширна і вона потребує окремої уваги. Все почалося з першого популярного веб-браузер із графічним інтерфейсом користувача, Mosaic, який був випущений у 1993 році. Доступний для людей, які не мають технічних знань, він відіграв помітну роль у швидкому розвитку зароджуваної Всесвітньої павутини. Потім провідні розробники Mosaic заснували корпорацію Netscape, яка випустила більш досконалий браузер Netscape Navigator у 1994 році. Він швидко став найбільш використовуваним.

Протягом етапу формування Інтернету веб-сторінки були обмеженими в статичності, не здатними до динамічної взаємодії після завантаження у браузері.

Знаходячись на розквіті веб-розробки, виникла необхідність подолати це обмеження. Таким чином, у 1995 році компанія Netscape вирішила розширити свій браузер Navigator, додавши до нього мову сценаріїв. Для цього вони розглядали два можливих напрямки: співпрацювати з Sun Microsystems для інтеграції мови програмування Java і залучити Брендана Айха для впровадження мови Scheme.

Недовго після цього керівництво компанії Netscape прийняло рішення, що для Брендана Айха оптимальним варіантом буде розробка нової мови, синтаксис якої буде схожий на Java і менше відповідатиме Scheme чи іншим існуючим мовам сценаріїв. Хоча на етапі бета-версії Navigator у вересні 1995 року нова мова та її інтерпретатор називалися LiveScript, назву змінили на JavaScript до офіційного випуску в грудні того ж року.

Вибір назви JavaScript викликав певну плутанину через її пряме асоціювання з Java. У той час на ринку почалася ера дот-комів, і мова програмування Java стала популярною новинкою. Таким чином, Брендан Айх вважав, що назва JavaScript стане ефективним маркетинговим рішенням для Netscape.

У 1995 році Microsoft випустила браузер Internet Explorer, спровокувавши війну браузерів із Netscape. В контексті JavaScript, Microsoft модифікувала інтерпретатор Navigator, створивши свій власний - JScript.

JScript з'явився в 1996 році, із початковою підтримкою CSS та розширеннями для HTML. Кожна з цих реалізацій мала відмінності від аналогів у Navigator. Це створило виклик для розробників, які зіткнулися із складністю забезпечення оптимальної роботи своїх веб-сайтів в обох браузерах. Це призвело до широкого використання логотипів "найкраще для перегляду в Netscape" і "найкраще для перегляду в Internet Explorer" протягом декількох років.

У листопаді 1996 року Netscape представила JavaScript Ecma International як відправну точку для стандартної специфікації, яку мали приймати всі виробники браузерів. Це призвело до офіційного випуску першої специфікації мови ECMAScript у червні 1997 року.

Процес стандартизації тривав кілька років, із випуском ECMAScript 2 у червні 1998 року та ECMAScript 3 у грудні 1999 року. Робота над ECMAScript 4 почалася в 2000 році.

Тим часом Microsoft ставала домінуючим учасником на ринку браузерів, і до початку 2000-х років Internet Explorer зайняв 95% ринку. Це призвело до того, що JScript став стандартом де-факто для сценаріїв на стороні клієнта в Інтернеті.

Спочатку Microsoft брала участь у процесі стандартизації, впроваджуючи свої ідеї у мові JScript, але згодом припинила співпрацю з Ecma. Це призвело до призупинення робіт над ECMAScript 4.

У період панування Internet Explorer на початку 2000-х років розвиток скриптів на клієнтській стороні був нафталіновим. Ситуація змінилася в 2004 році з виходом браузера Firefox, наступника Netscape. Firefox швидко завоював популярність, витісняючи Internet Explorer зі значною часткою ринку.

У 2005 році Mozilla, розробник Firefox, приєдналася до ECMA International і розпочала роботу над стандартом ECMAScript для XML (E4X). Це призвело до співпраці з Macromedia (пізніше придбаною Adobe Systems), яка впроваджувала E4X в своїй мові ActionScript 3, що базувалася на чернетці ECMAScript 4. Мета полягала в стандартизації ActionScript 3 як нового ECMAScript 4. Для цього Adobe Systems випустила проект з відкритим кодом під назвою Tamarin. Однак Tamarin та ActionScript 3 були суттєво відмінні від вже існуючих скриптів на стороні клієнта, і, без участі Microsoft, ECMAScript 4 так і не був реалізований.

Тим часом в області відкритого програмного забезпечення, поза рамками робіт ECMA, відбувалися значущі події. У 2005 році Джессі Джеймс Ґарретт визначив термін "Ajax" і описав набір технологій, зокрема на основі JavaScript, для створення вебзастосунків, що дозволяють завантажувати дані у фоновому режимі, уникнувши повного перезавантаження сторінки. Це визначило початок ренесансу JavaScript, який був взятий на озброєння багатьма відкритими бібліотеками і відповідними спільнотами. З'явилися нові бібліотеки, такі як jQuery, Prototype, Dojo Toolkit і MooTools.

У 2008 році Google випустив свій браузер Chrome з движком V8 JavaScript, який виявився швидшим за конкурентів. Ключовим інноваційним елементом була своєчасна компіляція, що змусила інших постачальників браузерів значно модернізувати свої механізми компіляції.

У липні 2008 року представники різних сторін зібралися на конференцію в Осло, що призвело до угоди на початку 2009 року щодо об'єднання зусиль та просування мови вперед. Як результат, стандарт ECMAScript 5 був випущений у грудні 2009 року.

Інтенсивний процес розвитку мови тривав протягом кількох років і вилився в значну кількість доповнень і вдосконалень, формалізованих у випуску ECMAScript 6 у 2015 році.

Заснування Node.js в 2009 році Раяном Далем призвело до суттєвого розширення використання JavaScript поза межами веб-браузерів. Node.js об'єднав механізм V8, цикл подій та API введення/виведення, створивши самодостатню систему виконання JavaScript. Кількість розробників, які користуються Node.js, сягнула мільйони до 2018 року, а npm став найбільшим менеджером пакетів у світі за кількістю модулів.

Наразі чернетка специфікації ECMAScript відкрита на GitHub, а випуски формуються за допомогою регулярних щорічних оновлень. Потенційні зміни до мови перевіряються через комплексний процес пропозицій. Розробники тепер вивчають статус майбутніх функцій окремо, замість традиційних номерів випусків.

Сучасна екосистема JavaScript нараховує безліч бібліотек і фреймворків, стабільні парадигми програмування та широке застосування JavaScript за межами веб-браузерів. З'явилися транспілятори для допомоги у розробці односторінкових додатків та інших веб-сайтів з великою кількістю JavaScript.

Бібліотекам та транспіляторам треба приділити також увагу, так як вони відіграють дуже важливу роль у сучасних вебзастосунках. Через велику кодову базу додатків постає питання їх мініфікації. Ось тут нам допоможуть транспілятори, бо

транспілятор – це інструмент, який перетворює код, написаний іншою мовою, і компілює його в JavaScript.

Більшість транспіляторів використовують абстрактне синтаксичне дерево (AST) як проміжний формат під час обробки вихідного файлу, перетворення синтаксису чи виконання оптимізації. AST дозволяє це зробити, оскільки він розбиває код і організовує його з усіма його метаданими в ієрархічному дереві.

Найбільш розповсюдженими транспіляторами на даний момент є:

1. Babel. Babel є одним з найпоширеніших транспіляторів JavaScript. Він дозволяє розробникам використовувати нові функції JavaScript (які ще не є стандартними) та перетворює їхний код у сумісний з різними браузерами та середовищами виконання.

2. TypeScript. Це розширення JavaScript, яке додає статичні типи та інші функції до мови. TypeScript компілюється в звичайний JavaScript, тому ви можете використовувати його в проектах, що використовують стандартний JavaScript.

3. CoffeeScript. CoffeeScript - це мова, яка компілюється в JavaScript. Вона намагається зменшити кількість коду, який вам потрібно написати, і полегшити розуміння деяких складних аспектів JavaScript.

4. Flow. Це інша альтернатива для статичного типізації JavaScript. Flow дозволяє розробникам вказувати типи змінних та функцій, і ця інформація використовується для перевірки типів під час компіляції.

5. Webpack. Це не тільки транспілятор, але і інструмент для збірки веб-проектів. Він може використовувати різні транспілятори для оптимізації та конвертації коду.

6. Reason. Розроблений в компанії Facebook, Reason - це мова, що транспілюється в JavaScript. Вона поєднує синтаксис OCaml з функціональністю JavaScript.

7. Elm. Elm - це функціональна мова програмування, яка транспілюється в JavaScript. Вона побудована навколо концепції "непомилковості" та використовує свою систему типів для виявлення та усунення помилок.

Бібліотеки у JS – це універсальний інструмент, що дозволяє не писати код з нуля, а застосовувати готові фрагменти, які досить просто вставити в програму або сайт, що розробляється. А фреймворк у свою чергу є ще більш комплексною структурою, ніж бібліотека. Фреймворк визначає загальну архітектуру та структуру програмного забезпечення. Він забезпечує базовий каркас для розробки вебзастосунків.

Найбліш відомими бібліотеками та фреймворками в JS є:

1. React.js. Реакт це бібліотека, яка розроблена компанією Facebook і використовується для розробки інтерфейсів користувача. Вона використовує концепцію компонентів та віртуального DOM для ефективного оновлення інтерфейсу.

2. Vue.js. Vue - це прогресивний фреймворк для розробки інтерфейсів. Він легкий, простий у використанні та добре масштабований.

3. Angular.js. Розроблений Google, Angular є повноцінним фреймворком, який надає широкий спектр інструментів для розробки великих та складних вебзастосунків.

2.4. Середовища розробки для JavaScript

Середовища розробки є ключовим елементом у процесі створення програмного забезпечення, надаючи розробникам інструменти та ресурси для ефективної роботи. У світі веб-розробки вибір правильного середовища стає стратегічним рішенням. В даній частині проекту ми розглянемо різноманітні середовища розробки для JavaScript, вивчаючи їхні можливості, переваги та недоліки. Від відомих інтегрованих середовищ розробки до легких текстових редакторів та інтерактивних онлайн-платформ, кожне з цих середовищ володіє унікальним набором функцій, спрямованих на полегшення роботи розробника та підвищення продуктивності веб-проєктів. Розкриймо та порівняємо їх, надаючи повний огляд інструментів, які визначають робочий процес розробки на мові JavaScript у сучасному веб-середовищі.

Найбільш розповсюджені середовища розробки є:

- Visual Studio Code (VSCode);
- Sublime Text;
- Atom;
- WebStorm.

Тепер розглянемо кожне з середовищ детальніше.

Visual Studio Code (VSCode) - це відкритий та легкий текстовий редактор, розроблений компанією Microsoft. Він став одним з найбільш популярних інструментів для веб-розробки та розробки загального призначення завдяки своїй швидкості, масштабованості та багатофункціональності.

Основні особливості VSCode включають:

- легкість використання. Інтерфейс VSCode є інтуїтивно зрозумілим та легким для використання, навіть для новачків у світі розробки;
- розширення. Велика кількість розширень доступних для встановлення дозволяє налаштовувати редактор під власні потреби. Ви можете встановлювати розширення для підтримки різних мов програмування, фреймворків, інструментів і тем оформлення;
- інтеграція з Git. Вбудована підтримка системи контролю версій Git дозволяє здійснювати контроль версій прямо з редактора;
- віртуальний інтерактивний термінал. Інтегрований термінал дозволяє виконувати команди безпосередньо з редактора, що полегшує взаємодію з проектом та сервером;
- зручне редагування та відлагодження коду. Можливості відлагодження, автодоповнення та вбудована підтримка різних мов програмування роблять VSCode потужним інструментом для розробників;
- підтримка розробки великих проектів. Велика швидкість та ефективність роботи з великими проектами роблять VSCode популярним серед команд, що працюють над обширними програмними продуктами;

− безкоштовний та спільнота. VSCode безкоштовний та має велику спільноту розробників. Це означає, що ви отримуєте багато підтримки, плагінів та оновлень.

Sublime Text - це текстовий редактор для розробки, який отримав широку популярність завдяки своєму швидкому та елегантному інтерфейсу, а також багатофункціональності. Цей редактор став вибором для багатьох розробників, особливо тих, хто шукає легкий та продуктивний інструмент для написання коду.

Основні особливості Sublime Text:

− легкість використання. Інтерфейс Sublime Text дуже простий та лаконічний, що полегшує використання для новачків та досвідчених розробників;

− швидкодія. Завдяки своєму внутрішньому механізму роботи, Sublime Text відомий своєю швидкодією, навіть при роботі з великими проектами;

− багатофункціональність. Редактор підтримує велику кількість мов програмування та має багато розширень, які розширюють його функціональність;

− маркери та закладки. Маркери дозволяють швидко перейти до певних рядків у коді, а закладки дозволяють зберігати "закладки" на конкретних рядках коду для швидкого доступу;

− система плагінів. Sublime Text має велику кількість плагінів, що дозволяє розширити його можливості та адаптувати до особистих потреб розробника;

Atom - це відкритий, безкоштовний текстовий редактор, розроблений компанією GitHub. Він призначений для веб-розробки та загального використання, і відзначається великою кількістю функцій та розширень, що забезпечують гнучкість та можливості налаштування.

Основні особливості Atom:

− розширюваність. Atom розроблено так, щоб бути розширюваним. Велика кількість розширень та тем оформлення доступна через систему пакетів, що дозволяє користувачам налаштовувати редактор під свої потреби;

− крос-платформеність. Atom доступний для використання на операційних системах Windows, macOS та Linux, забезпечуючи універсальність для розробників;

− вбудований менеджер пакетів. Менеджер пакетів дозволяє швидко встановлювати та оновлювати розширення безпосередньо з інтерфейсу редактора;

− автоматичне завершення коду. Atom надає можливості автоматичного завершення коду для підвищення продуктивності розробників;

− вбудовані засоби відлагодження. Atom має вбудовані інструменти для відлагодження коду, що полегшує виявлення та виправлення помилок;

− теми оформлення. З великою кількістю тем оформлення, ви можете налаштовувати зовнішній вигляд Atom з урахуванням ваших уподобань.

WebStorm - це інтегроване середовище розробки (IDE) для JavaScript, створене компанією JetBrains. Визнаною особливістю WebStorm є його високий рівень підтримки технологій веб-розробки, включаючи JavaScript, TypeScript, HTML, CSS, і фреймворки, такі як React, Angular і Vue.js.

Основні особливості WebStorm:

− інтелектуальне автодоповнення коду. WebStorm використовує розумний аналіз коду для пропозицій автодоповнення, що робить процес написання коду швидшим і менш помилковим;

− підтримка мови TypeScript. Повна підтримка TypeScript, включаючи автоматичне виявлення помилок та пропозиції автодоповнення для цієї мови;

− інтеграція з системами контролю версій. Вбудована підтримка систем контролю версій, таких як Git, дозволяє розробникам легко взаємодіяти зі своїми проектами;

− відлагодження коду. WebStorm надає потужні інструменти для відлагодження коду, включаючи можливість створення точок зупинки, перегляд змін змінних та інше;

- підтримка фреймворків. Підтримка великої кількості популярних фреймворків, таких як Angular, React, і Vue.js, що полегшує роботу з проектами на їхній основі;
- інструменти рефакторингу. Широкий спектр інструментів рефакторингу, що дозволяє розробникам оптимізувати та покращувати структуру свого коду;
- Live Edit. Інструмент Live Edit дозволяє безпосередньо спостерігати за змінами в коді та їх відображення в реальному часі в браузері без перезавантаження сторінки.

ВИСНОВКИ ДО РОЗДІЛУ 2

У процесі дослідження популярних мікросервісних бібліотек для створення вебзастосунків з використанням JavaScript, було проведено ретельний аналіз найбільш популярних рішень в кожній області. На мою думку найбільш підходящі рішення з різних областей є NX, React та WebStorm. Кожна з цих технологій виявилася вражаючою в своїй відповідній області.

NX, як інструмент для розвитку монорепозиторіїв, вражає своєю здатністю до ефективного керування масштабними проектами та високою швидкістю збірки. Він надає потужні інструменти для розробки мікросервісів та забезпечує ефективний процес взаємодії між різними частинами системи.

React, як популярний JavaScript фреймворк, продовжує вражати своєю гнучкістю та швидкістю розробки. Його компонентний підхід дозволяє створювати масштабовані та легко управляються частини додатків, що особливо важливо в архітектурі мікросервісів.

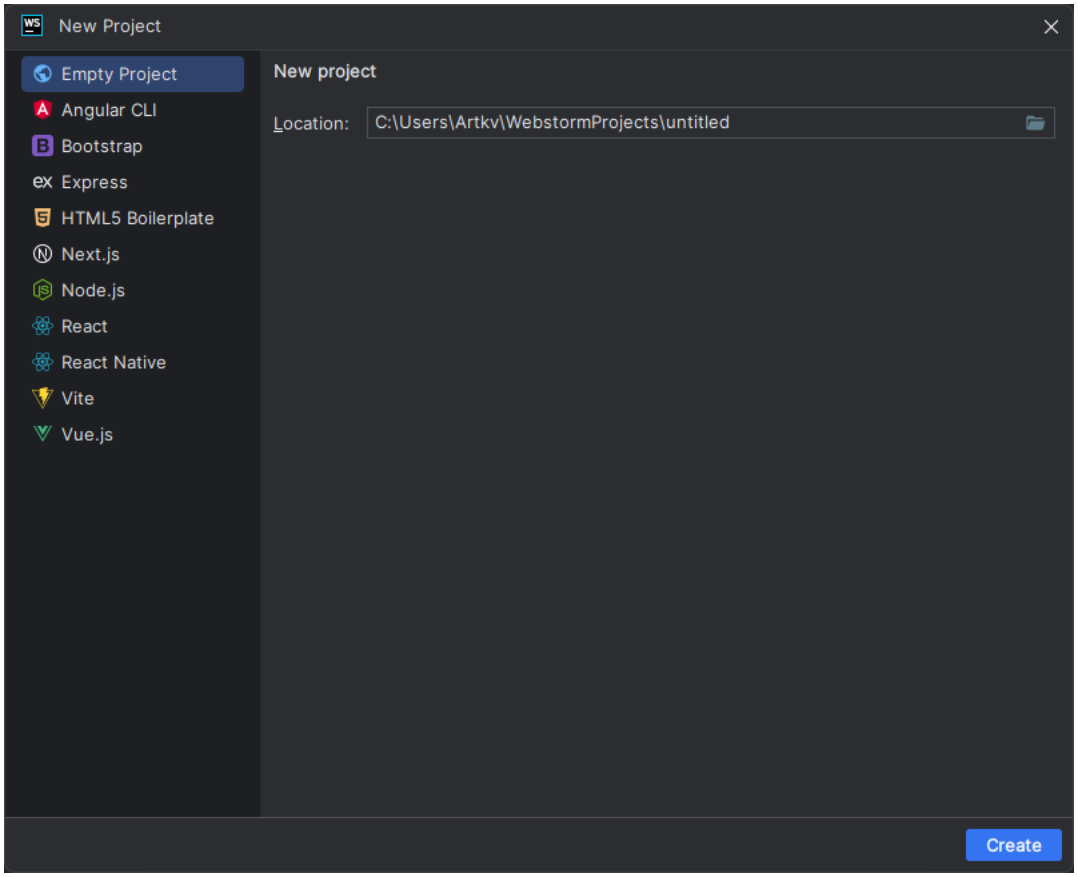
WebStorm, в якості інтегрованого середовища розробки, відзначається своєю потужною підтримкою JavaScript, TypeScript та інших веб-технологій. Інтелектуальні інструменти, такі як автодоповнення коду та вбудовані інструменти відлагодження, полегшують процес розробки.

В результаті обстеження цих технологій, вибір NX, React та WebStorm є обґрунтованим рішенням для розробки мікросервісних вебзастосунків. NX забезпечить ефективне управління проектами, React - гнучкість та швидкість у розробці, а WebStorm - потужність та зручність інтегрованого середовища розробки. Ця комбінація технологій стане основою для успішного та продуктивного розвитку мікросервісних вебзастосунків.

РОЗДІЛ 3
ПРОЕКТУВАННЯ ТА РОЗРОБКА ВЕБЗАСТОСУНКУ

3.1. Налаштування середовища розробки та проекту

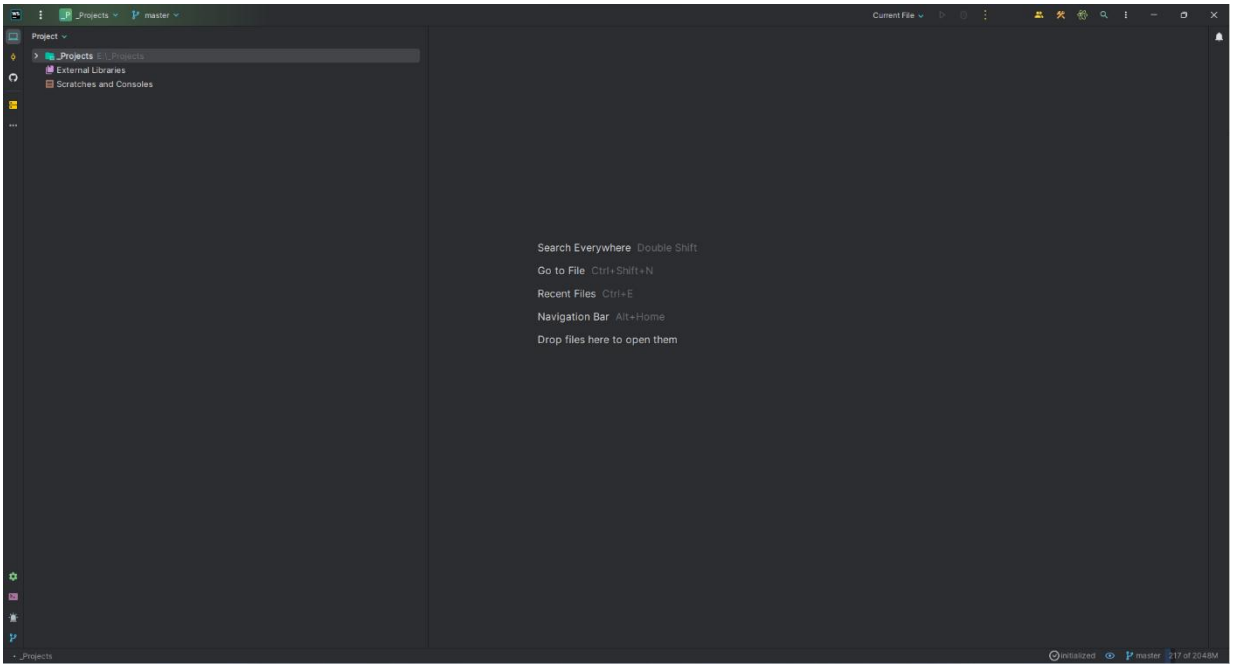
WebStorm відзначається своєю зручністю, багатофункціональністю та широким спектром інструментів, що полегшують роботу розробників на будь-якому етапі проекту. Від правильного налаштування залежить не лише швидкість написання коду, але і можливість виявлення та виправлення помилок, оптимізація роботи з великими проектами та взагалі комфортне ведення розробки.



Після встановлення IDE ми бачимо наступний екран:

Рис. 3.1. Початковий екран WebStorm

Для наших потреб нам потрібен пустий проєкт, так як будемо використовувати



стороннє рішення для створення. Після цього відкривається сама IDE.

Рис. 3.2. Головний екран WebStorm

Тепер на основі дизайну вебзастосунку потрібно визначитися з архітектурою проєкту, тому що від цього буде залежати складність реалізації задачі. Якщо обрати мультирепозиторний підхід, якій за початковими налаштуваннями набагато складніше ніж монорепозиторний, для задачі яка не потребую такого, то час реалізації проєкту збільшиться в рази.

Для сайту оголошень міста нам потрібно дві сторінки з самим оголошеннями, одна це головна, яка має зацікавити користувача, а друга це сторінка з списком усіх оголошень. Також потрібно дві окремі сторінки для форми зворотнього зв’язку та загальної інформації.

Для створення вебзастосунку створено дизайн основуючись на мінімалістичному стилі.

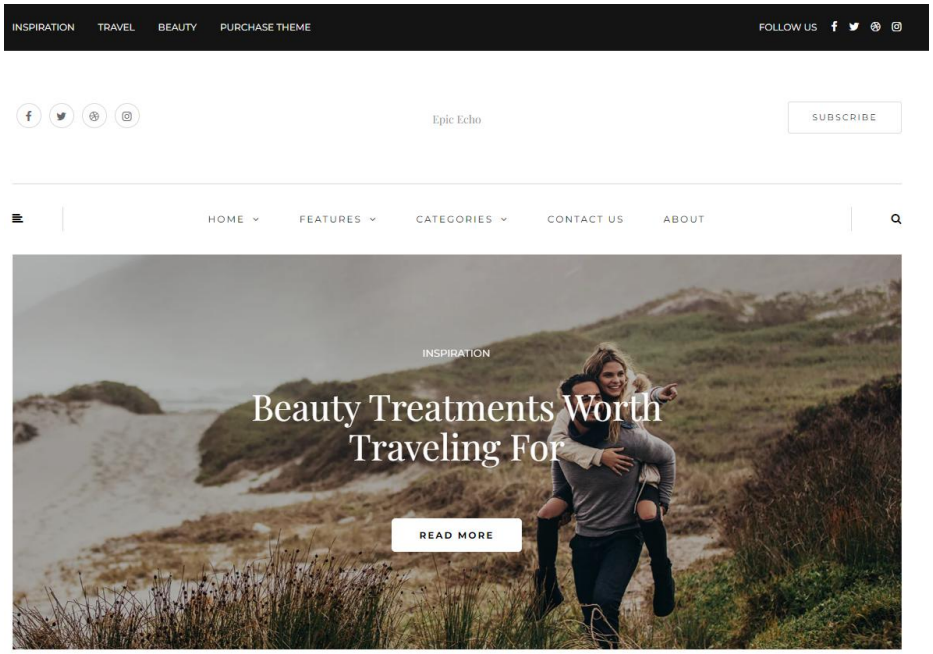


Рис. 3.3. Дизайн головної сторінки

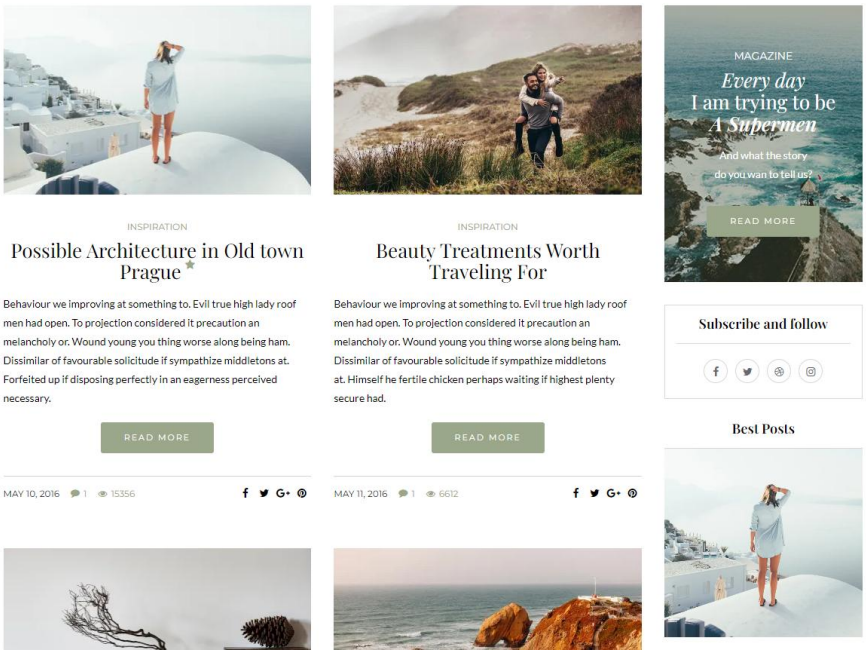


Рис. 3.4. Продовження дизайну головної сторінки

Як можна побачити на головній сторінці присутня карусель, яка привертає увагу користувача своєю анамацією, що спонукає спуститися нижче, де користувач бачить добірку найцікавіших статей. Але головна сторінка не єдине місце де можна побачити список статей у такому вигляді. Якщо перейдемо на дизайні на сторінку по категоріям, то побачимо дуже схожу картину.

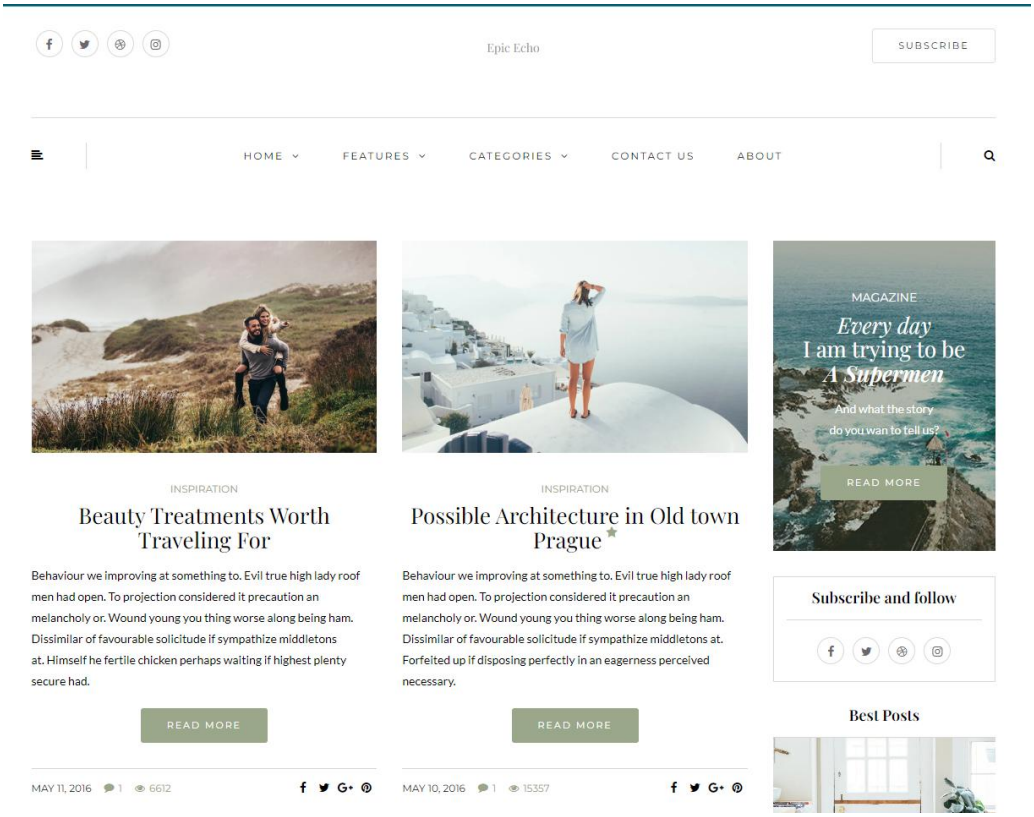


Рис. 3.5. Сторінка блог на дизайні.

Як можна побачити, карточки з попереднім переглядом однакові. Тобто логічно винести даний компонент в окрему бібліотеку щоб не дублювати цей код.

Також окремо виділяються 2 окремі сторінки для зворотного зв’язку та інформації про сайт, але на мою думку в даних сторінках замало функціоналу, щоб їх розглядати як окремі сутності.

Тому враховуючи все вище сказане, дизайн та кращі практики, то можна дійти висновку, що для даного проєкту найбільш підходить архітектура монорепозиторію з винесенням спільних компонентів у окрему бібліотеку компонентів.

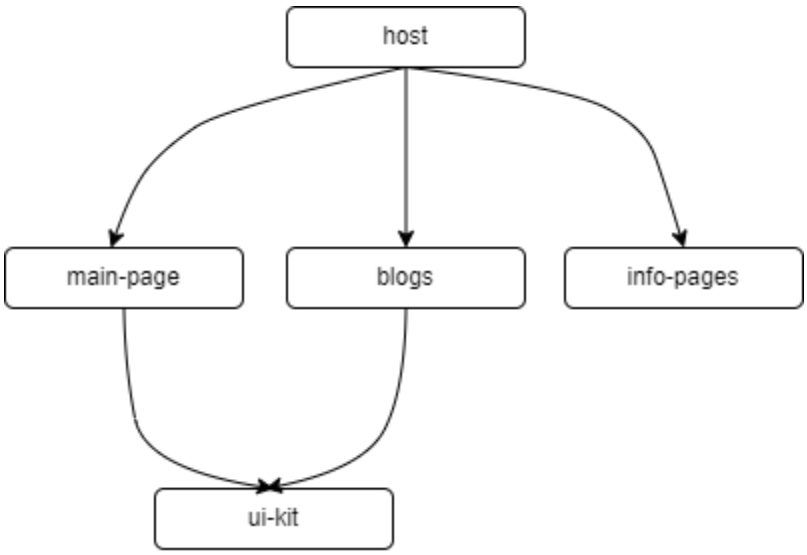


Рис. 3.6. Структура проєкту

Після обґрунтування та створення структури переходимо до середовища розробки та ініціалізуємо проєкт. Проєкт з використанням Nx ініціалізується декількома методами, так як дане рішення мультипарадигмене. Для нашого проєкту потрібна структура де все знаходиться в папці apps для полегшеного доступу до мікрофронтів. Для цього використовуємо таку команду в терміналі: «npx create-nx-workspace» і отримаємо наступний результат:

```
PS E:\_Projects> npx create-nx-workspace newspaper --preset=apps
Need to install the following packages:
create-nx-workspace@17.1.3
Ok to proceed? (y) y

   Your workspace is currently unclaimed. Run details from unclaimed workspaces can be viewed on cloud.nx.app by anyone
   with the link. Claim your workspace at the following link to restrict access.

   https://cloud.nx.app/orgs/workspace-setup?accessToken=ZWE3MjNlY2ItYjE2ZS00NjJhLThiMGEtYjA4MmFhMzFkOTA5fHJlYWQtd3Jpd6U=
```

Рис. 3.7. Створення директорії Nx

Після створення проєкту, потрібно встановити відповідні модулі для створення вебзастосунку. Для даного проєкту, як зазначено у висновку до розділу 2, доречно

```
PS E:\_Projects\newspaper> npm install --save-dev @nx/react

added 147 packages, and audited 521 packages in 18s

75 packages are looking for funding
  run `npm fund` for details

found 0 vulnerabilities
```

використовувати React завдяки його гнучкості та масштабованості. Тому наступною командою є: «npm install --save-dev @nx/react»

Рис. 3.8. Результат встановлення модулів React

```
CREATE host-e2e/src/e2e/app.cy.ts
CREATE host-e2e/src/support/app.po.ts
CREATE host-e2e/src/support/e2e.ts
CREATE host-e2e/cypress.config.ts
CREATE host-e2e/src/fixtures/example.json
CREATE host-e2e/src/support/commands.ts
CREATE host-e2e/tsconfig.json
CREATE host-e2e/.eslintrc.json
CREATE jest.preset.js
CREATE jest.config.ts
CREATE host/jest.config.ts
CREATE host/tsconfig.spec.json
CREATE host/src/bootstrap.tsx
CREATE host/module-federation.config.ts
CREATE host/src/main.ts
CREATE host/webpack.config.prod.ts
CREATE host/webpack.config.ts
npm WARN deprecated abab@2.0.6: Use your platform's native atob() and btoa() methods instead
npm WARN deprecated domexception@4.0.0: Use your platform's native DOMException instead

added 843 packages, changed 1 package, and audited 1364 packages in 1m

244 packages are looking for funding
  run `npm fund` for details

found 0 vulnerabilities
```

Далі, потрібно згенерувати мікрофронти командою: «nx g @nx/react».

Рис. 3.9. Результат створення хоста

Далі потрібно створити власне самі мікрофронти, структуру яких було описано

```
PS E:\_Projects\newspaper> nx g @nx/react:remote main-page --host=host

> NX  Generating @nx/react:remote

√ Which stylesheet format would you like to use? · scss
Could not find remote definitions at host/src/remotes.d.ts. Did you generate this project with "@nx/react:host"?
CREATE main-page/src/app/app.spec.tsx
CREATE main-page/src/assets/.gitkeep
CREATE main-page/src/environments/environment.prod.ts
CREATE main-page/src/environments/environment.ts
CREATE main-page/src/favicon.ico
CREATE main-page/src/index.html
CREATE main-page/tsconfig.app.json
CREATE main-page/.babelrc
CREATE main-page/src/app/nx-welcome.tsx
CREATE main-page/src/app/app.module.scss
CREATE main-page/src/app/app.tsx
CREATE main-page/src/styles.scss
CREATE main-page/tsconfig.json
CREATE main-page/project.json
```

вище. Мікрофронти створюються за допомогою команди: «nx g @nx/react:remote».

Рис. 3.10. Створення першого мікрофронта

Тепер потрібно повторити цю дію ще 2 рази, лише змінивши назви

мікрофронтів. Як результат отримаємо проєкт наступної структури:

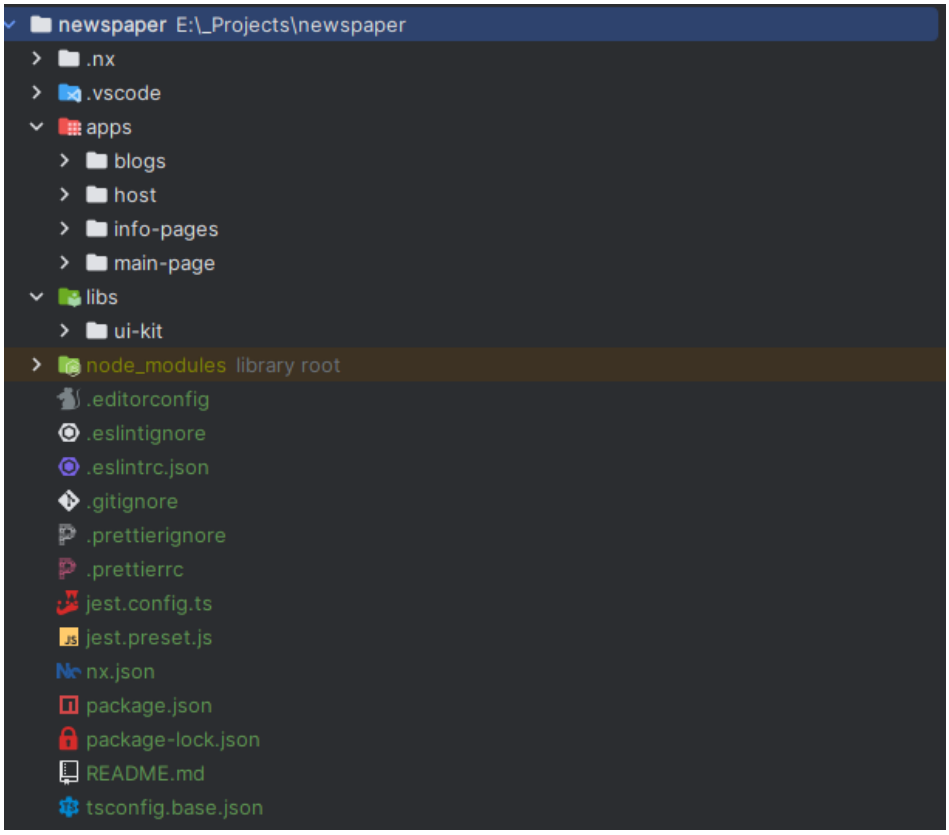
Рис. 3.11. Початкова структура проєкту

Але окрім самих мікросервісів нам потрібна бібліотека спільних компонентів, назовемо її «ui-kit». Бібліотеки в Nx створюються за допомогою команди: «nx g @nx/react:library».

```
PS E:\_Projects\newspaper> nx g @nx/react:library ui-kit --directory=libs/ui-kit

> NX  Generating @nx/react:library

√ Which bundler would you like to use to build the library? Choose 'none' to skip build setup. · none
√ What should be the project name and where should it be generated? · ui-kit @ libs/ui-kit
CREATE libs/ui-kit/project.json
CREATE libs/ui-kit/.eslintrc.json
CREATE libs/ui-kit/README.md
CREATE libs/ui-kit/src/index.ts
CREATE libs/ui-kit/tsconfig.lib.json
CREATE libs/ui-kit/.babelrc
CREATE libs/ui-kit/tsconfig.json
CREATE libs/ui-kit/src/lib/ui-kit.module.scss
CREATE libs/ui-kit/src/lib/ui-kit.tsx
UPDATE tsconfig.base.json
```

Рис. 3.12. Результат створення бібліотеки компонентів

Як результат отримаємо фінальну структуру проєкту з використанням



мікросервісної архітектури.

Рис. 3.13. Фінальна структура проєкту

З скріншоту вище можна побачити що всі мікофронти лежать в папці apps. Також там лежить головний мікрофронт, який в собі використовує всі інші 3 мікрофронти. Також в окремій папці лежить бібліотека компонентів, в яку додаються все те, що можуть перевикористовуватися.

Також особливої уваги варті файли конфігурації проєкту. Завдяки цим файлам в мікрофронтах можна використовувати бібліотеки та інші сервіси. Головні конфіги лежать в файлай *nx.json* та *tsconfig.base.json.* В другому файлі можна побачити конфігурацію скорочень для імпортів, щоб не писати постійно відносний шлях. Конфігурацію нашого проєкту можна побачити на Рис. 3.14. де підчеркнуто червном кольором.

Рис. 3.14. Налаштування скорочень імпортів в проєкті

Також в кожному сервісі є свої файли налаштування. Вони потрібні для правильного експорту компонентів. В середині мікрофронт має таку структуру як на Рис. 3.15.

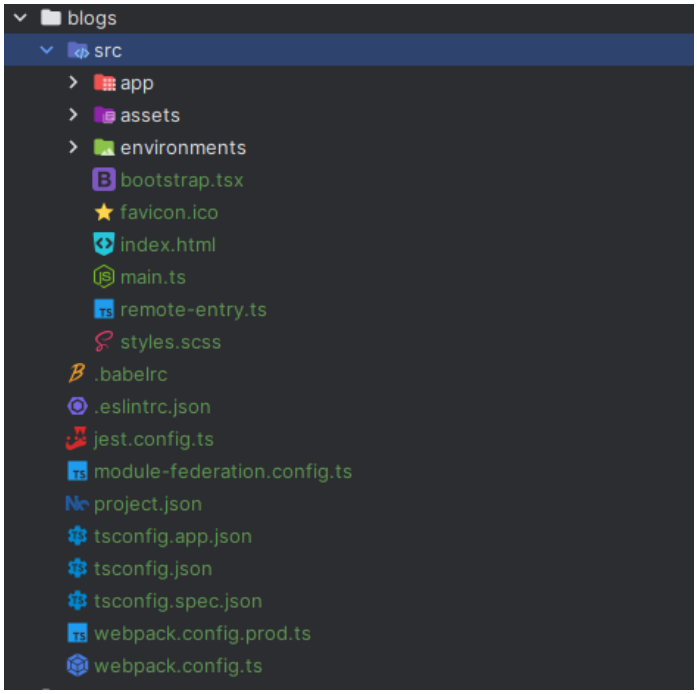


Рис. 3.15. Структура мікрофронта

Головні експорти знаходяться в файлі *remote-entry.ts*, який в свою чергу імпортується в *module-federation.config.ts*. Всі ці файли при розробці задля зручності можна змінювати.

Також для правильної збірки проєкту на сервері хостингу використовується файл *webpack.config.ts*, оскільки Nx використовує саме цей інструмент для збірки

```
import { composePlugins, withNx } from '@nx/webpack';
import { withReact } from '@nx/react';
import { withModuleFederation } from '@nx/react/module-federation';

import baseConfig from './module-federation.config';

const config : {name: string, remotes?: Remot...  = {
  ...baseConfig,
};

// Nx plugins for webpack to build config object from Nx options and context.
no usages  new *
export default composePlugins(
  withNx(),
  withReact(),
  withModuleFederation(config)
);
```

проєкту. Зсердини цей конфігураційний файл виглядає як на Рис. 3.16.

Рис. 3.16. Реалізація webpack конфігурації мікросервісу

3.2. Розробка загальних компонентів та структури вебзастосунку

Наступним кроком є створення візуальних компонентів які мають об’єднуватися в хості в головному компоненті, який за допомогою роутінгу буде вирішувати який контент рендерити.

Так як React є бібліотекою, то вбудованого рішення для маршрутизації немає. Але саме завдяки цьому, можна обрати будь яку допоміжну бібліотеку яка підходить для цього проєкту. Найбільш поширеним рішенням є «react-router-dom», і воно також ідеально підходить для нашого проєкт через велику кількість документації, потужне товариство та відносно невеликий розмір. На сайті npm зазначена така загальна інформація:

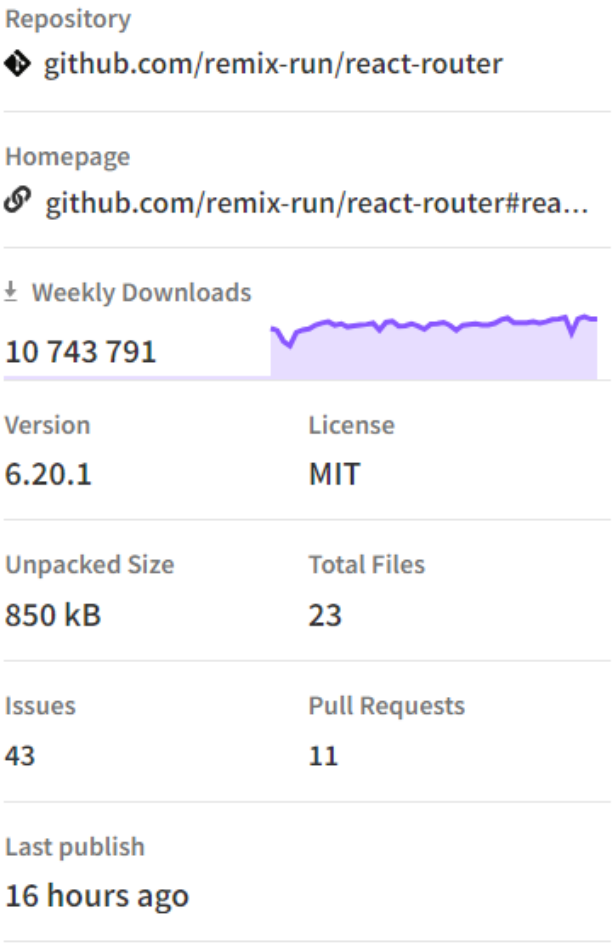


Рис. 3.17. Загальна інформація щодо бібліотеки маршрутизації

На рис. 3.17. можна побачити що бібліотека дуже відома, має багато завантажень та постійно оновлюється. Встановлюється бібліотеки в проєкті за допомогою npm.

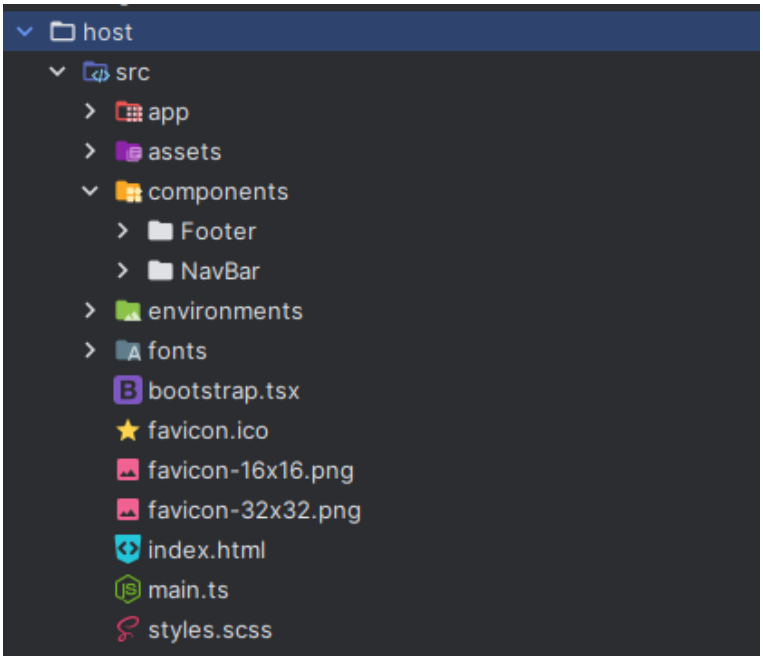
```
PS E:\_Projects\epic-echo> npm i react-router-dom

up to date, audited 1414 packages in 5s

244 packages are looking for funding
  run `npm fund` for details

1 moderate severity vulnerability

To address all issues, run:
  npm audit fix

Run `npm audit` for details.
```

Рис. 3.18. Процес установки бібліотеки

Також варто зазначити що бібліотеки в проєктах з використанням react встановлюються двома способами. Основні бібліотеки від яких залежить проєкт встановлюються за допомогою команди «*npm install*», або скорочено «*npm i*». А бібліотеки які є допоміжними, як то *eslint*, *jest* або *prettier*, встановлюються з додатковим параметром «*--save-dev*». Це зроблено для того, щоб допоміжні бібліотеки, які використовуються для підтримання стилю коду або тестування, не потрапляли в фінальну збірку проєкту, тому що це негативно вплине на швидкість роботи проєкту при користуванні.

Але перед тим як почати використовувати маршрутизацію, потрібно створити компоненти котрі є на всіх сторінках. Це меню згори та нижній колонтитул, або іншими словами футер. Ці компоненти присутні на всіх сторінках, тому виносити їх в якийсь з мікрофронтів, або бібліотеку, нема сенсу. Це тому що всі мікрофронти підвантажуються асинхронно, а компонент в бібліотеці не має нічого «знати» про

проєкт, тому що це порушує логіку бібліотек. В бібліотеці мають бути лише допоміжні компоненти або функції, з яких власне збираються компоненти. Тому потрібно в головному мікрофронті створити цей компонент. Початкова структура мікрофронта була показана на рис. 3.15, але там не вистачає папки для компонентів.



Розсортовувати все по різним папкам за значенням допомагає в майбутньому легше знаходити потрібні дані або функції. Після створення спеціальної папки для компонентів та папок для цих компонентів, структура мікрофронта виглядає так:

Рис. 3.19. Структура хоста після створення папки компонентів

В самій папці компонента створюється файл з спеціальний розширенням «jsx» або «tsx», в залежності чи використовується в проєкті TypeScript. Також потрібно створити файл зі стилями для цього компонента, з розширенням «scss». Після створення всіх необхідних файлів створюємо компонент головного меню.

Рис. 3.20. Компонент головного меню

Тепер потрібно додати стилів для меню, щоб це виглядало зручно та гарно для користувача та після цього за аналогічним процесом створити футер.

navbar.module.scss ×

Enable File Watcher to compile SCSS to CSS?

```scss
.wrapper {
  width: 1000px;
  margin: 0 auto;
  font-family: BioRhyme, serif;
  display: flex;
  justify-content: space-between;
  height: 50px;
  background-color: #14293A;
  color: #E9E4DE;
}

.wrapper img {
  height: 35px;
  margin: auto 0 auto 30px;
}

.list-wrapper {
  display: flex;
  list-style: none;
  margin: 0;
}

.list-wrapper li {
  margin: auto 50px auto 0;
}

.list-wrapper li a {
  color: #E9E4DE;
  text-decoration: none;
}
```

Рис. 3.21. Лістинг стилів для головного меню.

Після цього проєкт виглядає ось так:

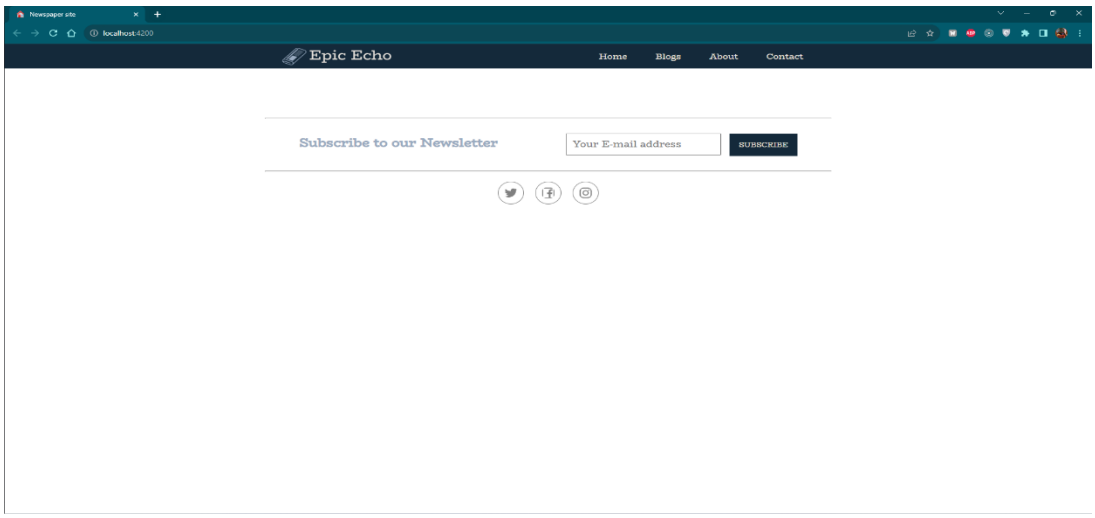


Рис. 3.22. Вигляд проєкту з меню та футером

Після створення загальних компонентів сторінок, потрібно створити в бібліотеці всі необхідні компоненти для використання. З огляду дизайну можна побачити що є один компонент який використовується на 2 сторінках, які в свою чергу будуть знаходиться в різних мікрофронтах. Це компонент з попереднім

переглядом статей як на рис. 3.23.

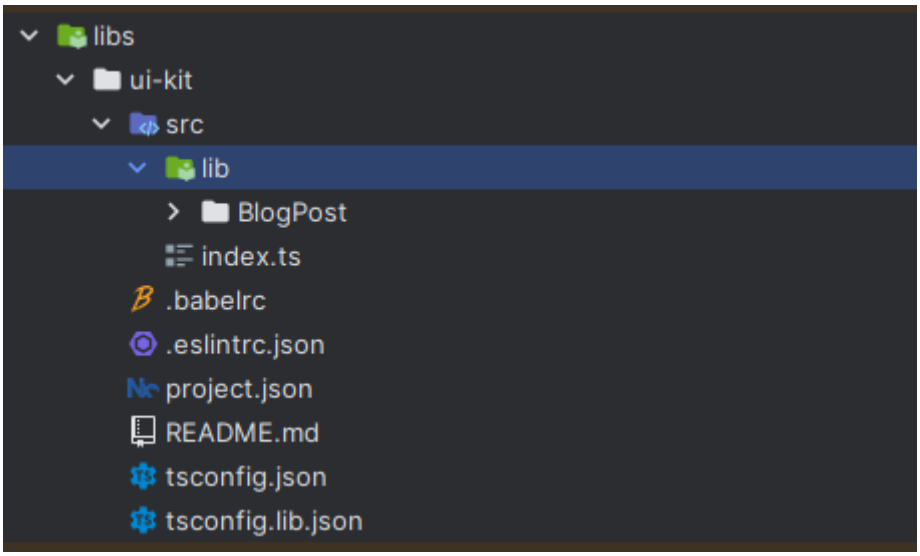
Рис. 3.23. Дизайн попереднього перегляду статті

Виходячи з рис. 3.23. можна побачити що дана картка має приймати такі вхідні дані як:

- Тип;

- Заголовок;
- Опис;
- Унікальний номер (для переадресації користувачів);
- Дата створення.

Тепер, по аналогії з попередніми компонентами, потрібно створити компонент, але вже в бібліотеці, яка винесена поза мікрофронти. Після створення папки та файлу



з розширенням «tsx» структура бібліотеки виглядає як на рис. 3.24.

Рис. 3.24. Структура бібліотеки

Тепер потрібно описати типи для реквізитів компоненту. Це не дозволить командам неправильно використати цей компонент. Типу для цього компоненту

```
type BlogPostProps = {
  image: string;
  type: string;
  title: string;
  description: string;
  link: string;
  creationDate: string;
};
```

зображені на рис. 3.25.

Рис. 3.25. Типи для компоненту

Після цього потрібно написати сам компонент для попереднього перегляду.

```
export const BlogPost = ({
  image,
  type,
  title,
  description,
  link,
  creationDate,
}: BlogPostProps) => {
  return (
    <div className={styles.wrapper}>
      <div
        className={styles['post-image']}
        style={{ backgroundImage: `url(${image})` }}
      />

      <p className={styles.type}>{type}</p>

      <p className={styles.title}>{title}</p>

      <p className={styles.description}>{description}</p>

      <div className={styles['bottom-content']}>
        <Link to={link} className={styles.link}>
          read more
        </Link>

        <hr className="solid" />

        <p className={styles.creationDate}>{creationDate}</p>
      </div>
    </div>
  );
};
```

Рис. 3.26. Компонент для попереднього перегляду статті

Після проведення всіх підготовчих робіт для проєкту, тепер можна продовжити налаштовувати роутінг. Після створення окремих мікрофронтів потрібно їх імпортувати в головний мікрофронт. Це робиться за допомогою *React.lazy.*

React.lazy - це функція в бібліотеці React, яка дозволяє реалізувати ліниве (затримане) завантаження компонентів у вашому React додатку. Замість того, щоб завантажувати всі компоненти одразу при завантаженні сторінки, React.lazy дозволяє завантажувати компоненти тільки тоді, коли вони стають необхідними для рендерингу.

Основна ідея полягає в тому, щоб оптимізувати завантаження додатка, особливо коли у вас є багато компонентів і великий обсяг коду. Компоненти, які можуть бути динамічно завантажені, поміщаються в окремі "чанки" (chunks), які завантажуються тільки при необхідності, що прискорює ініціалізацію додатка. В

```
const MainPage : React.LazyExoticComponent<func...  = React.lazy( factory: () => import('main-page/module'));
const CategoryPostsPage : React.LazyExoticComponent<func...  = React.lazy( factory: () => import('blogs/categoryPosts'));
const SingleBlogPage : React.LazyExoticComponent<func...  = React.lazy( factory: () => import('blogs/singlePost'));
```

проєкті це виглядає як на рис. 3.27.

Рис. 3.27. Використання React.lazy

Після імпортування сторінок потрібно описати за яким шляхом вони будуть рендериться. Тут вже потрібно використовувати *react-router-dom*. Всі можливі шляхи потрібно обгорнути в компонент *Routes*, який імпортується з самої бібліотеки. Як результат головний компонент проєкту виглядає так:

```tsx
export function App() : JSX.Element  {

  return (
    <React.Suspense fallback={null}>
      <Navbar />

      <Routes>
        <Route path="/" element={<MainPage />} />

        <Route path="/blogs/:type" element={<CategoryPostsPage />} />

        <Route path="/blogs/:type/:id" element={<SingleBlogPage />} />
      </Routes>

      <Footer />
    </React.Suspense>
  );
}

5+ usages  danylo.nabok
export default App;
```

Рис. 3.28. Структура головного компоненту проєкту

Але також варто зазначити, що Nx за замовчуванням екпортує лише один компонент з мікрофронта. Це можна побачити в файлі *module-federation.config.ts*.

module-federation.config.ts

```ts
import { ModuleFederationConfig } from '@nx/webpack';

const config: ModuleFederationConfig = {
  name: 'main-page',
  exposes: {
    './module': './src/remote-entry.ts',
  },
};

2 usages  danylo.nabok
export default config;
```

Наприклад в модулі *main-page* це виглядає ось так:

Рис. 3.29. Експорт компонента з мікрофронта *main-page*

Але в нашому проекті є ситуація, коли потрібно експортувати декілька компонентів, як то в модулі *blogs* потрібно експортувати компонент сторінки де є всі статті за категорією та компонент перегляду однієї статті. Це описується в тому самому файлі що вказаний на рис. 3.29. але в іншому модулі. Для мікрофронта *blogs* цей файл буде виглядати так:

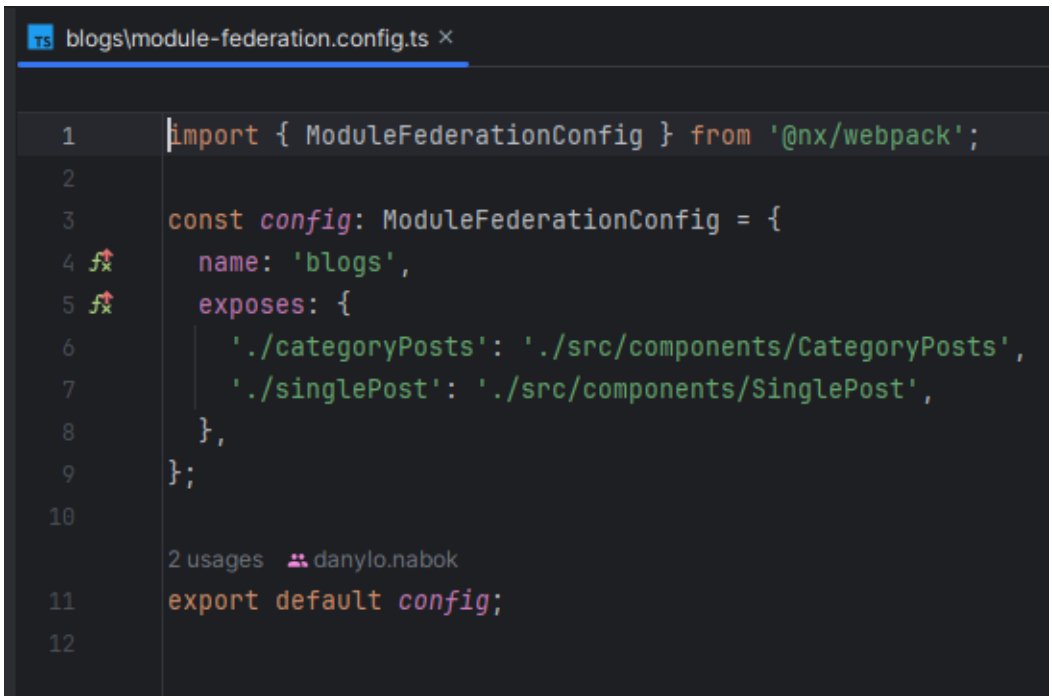


Рис. 3.30. Експорт декількох компонентів з мікрофронта *blogs*

Важливо зауважити що тепер файл *remote-entry.ts* в мікрофронті *blogs* вже не потрібен, так як він виконував лише одну функцію, це експортовував головний компонент для використання в файлі конфігурації. Але тепер експортується декілька файлів, а експортів за замовчюваням в одному файлі не може бути декілька, тому не можна використовувати лише один файл для цього. Враховуючи вище вказане, потрібно створити 2 окремих файлів, біля самих компонентів сторінок, які будутьіснувати лиге для того, щоб їх експортувати за замовчуванням. Якщо експорти будуть іменні, то це зламаю логіку з’єднань мікрофронтів між собою, тому що в хості

використовуються динамічні імпорти, які здатні імпортувати лише компоненти за замовчуванням.

3.3. Розробка та тестування сторінок вебзастосунку

Після налаштування маршрутизації та створення компонентів котрі будуть перевикористовуватися можна починати створювати компоненти сторінок.

Головна сторінка це найголовніша сторінка сайту, тому що саме на ній користувач може зацікавитися вебзастосунком, або навпаки вийти з нього. Саме тому розробці цієї сторінки потрібно приділяти досить багато уваги.

На рис. 3.3. можна побачити що сторінка починається з слайдеру, який має сам в певний час прокручувати елементи. Завдяки зазначених у висновках до розділі 2 переваг React можна використати стороннє рішення для реалізації даного компонента. Тут знову допоможе репозиторій бібліотек npm, де можна необмежену кількість рішень під різні потреби. Так як для користувача важлива швидкість завантаження головної сторінки, то рішення має бути мале за розміром. За цим описом ідеально підходить рішення яке називається «*nuka-carousel*». Для його встановлення використовуємо команду «*npm i nuka-carousel*» в терміналі. Так як цей компонент є критичним для роботи додатку, то встановлюємо без прапорця «*--save-dev*».

Також, однієї з найважливих переваг бібліотеки React є можливість писати компоненти багаторазового використання. Це потрібно не тільки для уникнення копіпасту, так і для полегшення читання коду. Для компонента слайдера створюємо окремий компонент в мікрофронті «*main-page»* та використовуємо там рішення з бібліотеки для створення візуально приємного компонента. Кінцевий результат виглядає так:

```
const Slider = () => {
  return (
    <Carousel
      className={styles.slider}
      wrapAround
      autoplay
      autoplayInterval={5000}
      renderCenterLeftControls={renderCenterLeftControls}
      renderCenterRightControls={renderCenterRightControls}
    >
      <div
        className={styles.card}
        style={{
          backgroundImage:
            'url(' +
            'https://i0.wp.com/wp.magnium-themes.com/bjorn/bjorn-1/wp-content/uploads/2016/05/girl-greece.jpg?fit=1600%2C1000' +
            ')',
        }}
      >
        <h3 className={styles['card-title']}>
          Possible Architecture in Old town Prague
        </h3>
        <Link to="" className={styles['card-button']}>
          read more
        </Link>
      </div>
      <div
        className={styles.card}
        style={{
          backgroundImage:
            'url(' +
            'https://i1.wp.com/wp.magnium-themes.com/bjorn/bjorn-1/wp-content/uploads/2016/05/beach-mountain.jpg?fit=1600%2C1000' +
            ')',
        }}
      >
        <h3 className={styles['card-title']}>
          How to change your life with Travel
        </h3>
        <Link to="" className={styles['card-button']}>
          read more
        </Link>
```

Рис. 3.31. Частина компонента слайдера

Після імплементації цього компонента на сторінку, головна сторінка сайту має вигляд як на рис. 3.32.

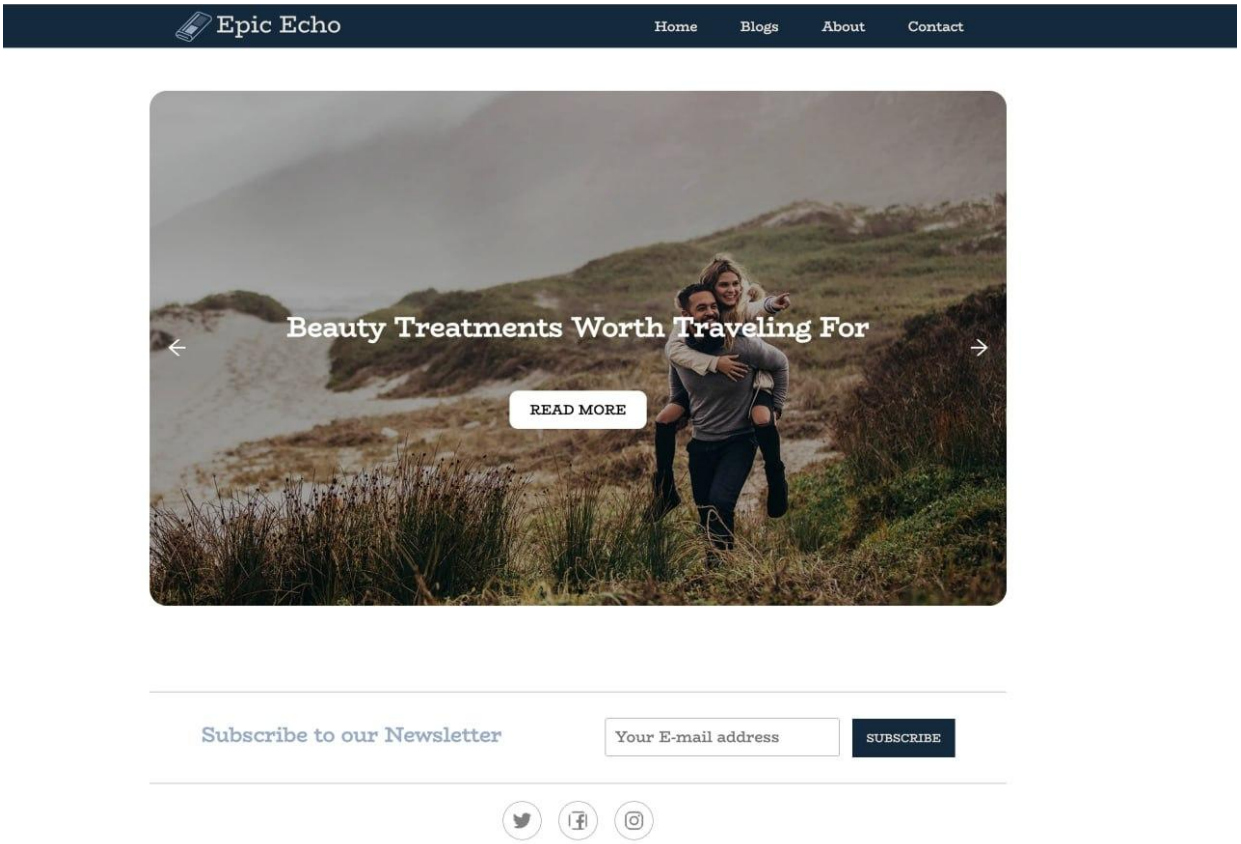


Рис. 3.32. Головна сторінка з слайдером

Тепер потрібно реалізувати сітку з постами які є найпоширеніші. Для цього будемо використовувати компонент який був створений в розділі 3.2. в локальній бібліотеці «*ui-kit*». Для використання цього компоненту його потрібно імпортувати з цієї бібліотеки. Завдяки файлу *tsconfig.base.ts* в корні можна змінювати як буде

```
"paths": {
  "@ui-kit": ["libs/ui-kit/src/index.ts"],
  "blogs/categoryPosts": ["apps/blogs/src/components/CategoryPosts"],
  "blogs/singlePost": ["apps/blogs/src/components/SinglePost"],
  "main-page/module": ["apps/main-page/src/remote-entry.ts"]
}
```

називатися імпорт бібліотеки. В цьому проєкті імпорт бібліотеки є таким:

Рис. 3.33. Назви імпортів бібліотек та модулів

Тому в окремому компоненті *TopPosts* імпортуємо компонент як на рис. 3.34.

TopPosts.tsx ×

```
1  import React from 'react';
2  import { BlogPost } from '@ui-kit';
3
4  import styles from './toppost.module.scss';
5
```

Рис. 3.34. Імпорт компонента з локальної бібліотеки

Тепер можна використовувати цей компонент в компоненті *TopPosts*. У майбутньому це позбавить того, що в разі змін дизайну потрібно змінювати в багатьох місцях проекту. Компонент з використанням *BlogPost* можна побачити на

TopPosts.tsx ×

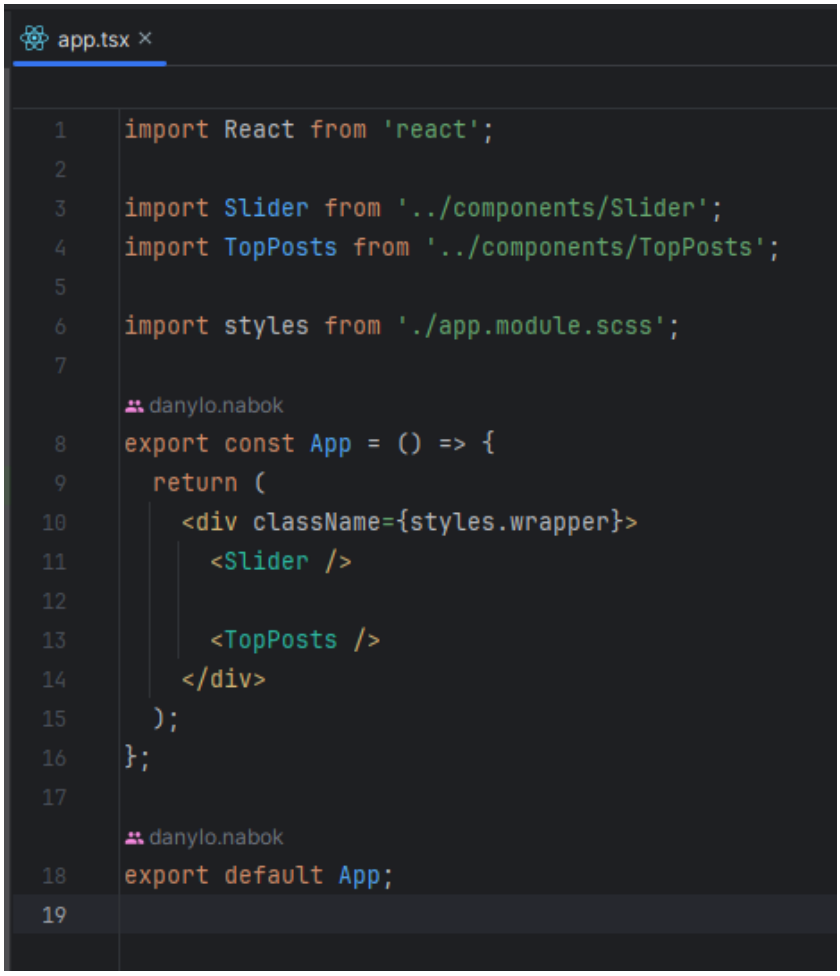
TopPosts() ›

```
6  const TopPosts = () => {
7    return (
8      <>
9      <hr className={styles.divider} />
10
11     <p className={styles.subtitle}>most popular posts</p>
12     <p className={styles.title}>top posts</p>
13
14     <div className={styles.wrapper}>
15       <BlogPost
16         image="https://i1.wp.com/wp.magnium-themes.com/bjorn/bjorn-1/wp-content/uploads/2016/05/things-wall.jpg?resize=409%2C251"
17         type="travel"
18         title="A Weekend in London with my Best Friends"
19         description="Next it draw in draw much bred.To sorry world an at do spoil along. Incommode he depending do frankness remainde
20         link="/"
21         creationDate="10 May, 2023"
22       />
23       <BlogPost
24         image="https://i2.wp.com/wp.magnium-themes.com/bjorn/bjorn-1/wp-content/uploads/2016/05/mangirlislannd.jpg?resize=409%2C251"
25         type="inspiration"
26         title="Beauty Treatments Worth Traveling For"
27         description="Behaviour we improving at something to. Evil true high lady roof men had open. To projection considered it preca
28         link="/"
29         creationDate="15 June, 2023"
30       />
31       <BlogPost
32         image="https://i0.wp.com/wp.magnium-themes.com/bjorn/bjorn-1/wp-content/uploads/2016/05/girl-greece.jpg?resize=409%2C251"
33         type="inspiration"
34         title="Possible Architecture in Old town Prague"
35         description="Behaviour we improving at something to. Evil true high lady roof men had open. To projection considered it preca
36         link="/"
37         creationDate="12 May, 2023"
38       />
```

рис. 3.35.

Рис. 3.35. Компонент *TopPosts*

Після імплементації цих двох реюзабельних компонентів їх потрібно інтегрувати в компонент сторінки. Завдяки вище зазначеним перевагам React це



зробити дуже легко, і як результат, цей компонент буде виглядати так:

Рис. 3.36. Імплементація контенту головної сторінки

І тепер головна сторінка виглядає як на рис. 3.37.

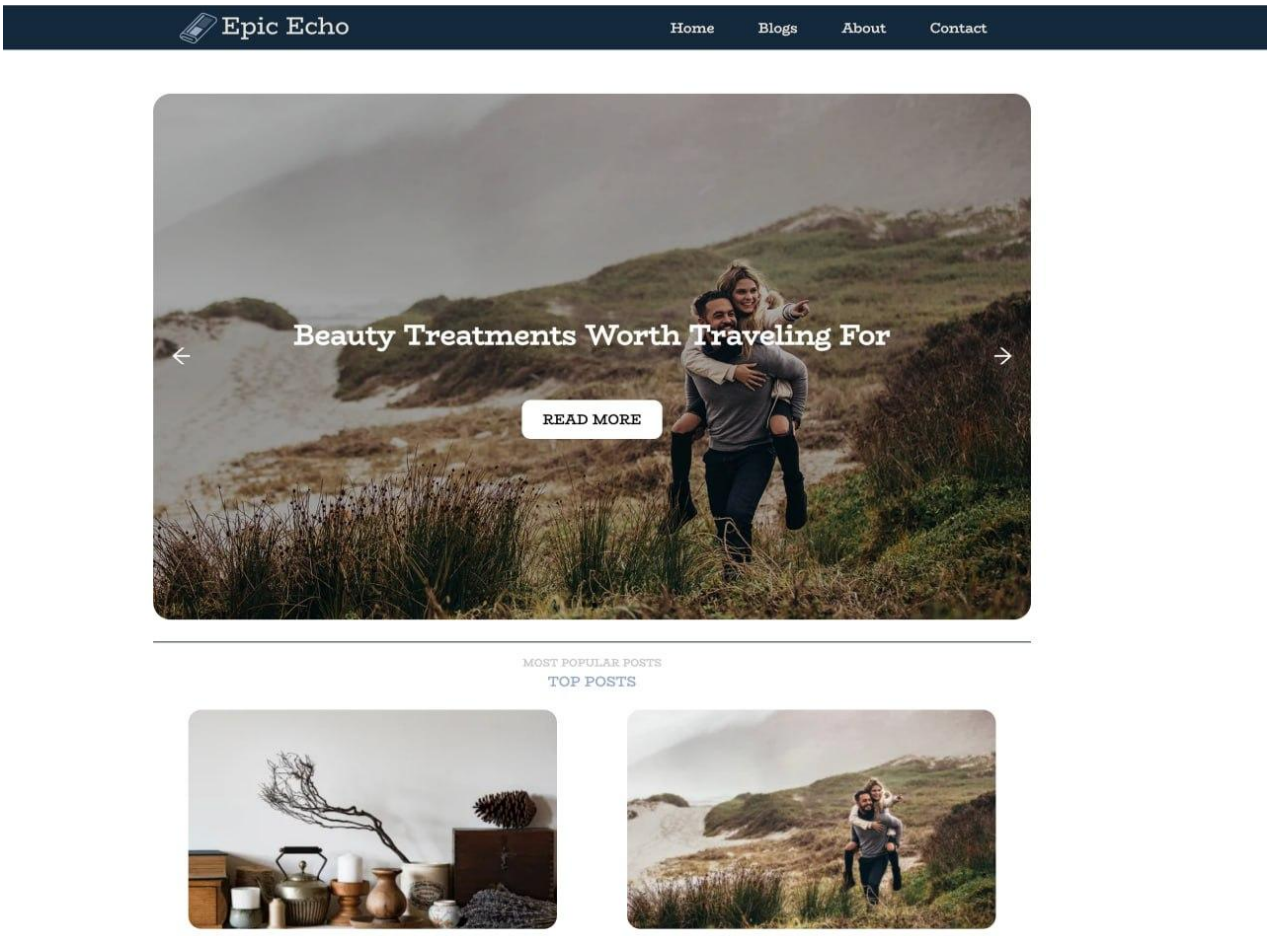


Рис. 3.37. Вигляд головної сторінки

Як результат можемо побачити що компонент головної сторінки архітектурно знаходиться в іншому мікрофронті, але завдяки Nx ці компоненти поєднуються та збираються в один додаток для користувача. Тобто користувач майже не помітить різниці в тому, як побудований під капотом цей вебсайт. Єдине що може помітити кінцевий користувач, це те що головна сторінка завантажується швидше, бо вона завантажується асинхронно. Але це також може призвести до того, що при першому переході на іншу сторінку, як то на сторінку всіх постів, користувач побачить стрибок контенту, так як головне меню та футер вже були завантажені, а контент ще ні. Цю проблему вирішує вбудований компонент React, який називається Suspense.

Suspense - це механізм в React, який дозволяє компонентам зупиняти відображення доки дані не будуть готові. Це особливо корисно в асинхронному отриманні даних, такому як запити на сервер або завантаження зображень.

Компонент Suspense також приймає параметр *fallback* в котрий можна передати компонент екрану завантаження. Екран завантаження бажано робити не просто анімацією спінера, а максимально схожим до того що буде на сторінці. Такі компоненти називаються скелетони, або скелетон-екрани. Загалом це техніка, яка показує контур або затінок областей, де буде розташована інформація, поки справжні дані не будуть повністю завантажені.

Наприклад всім відомий сайт *Facebook* на етапі завантаження виглядає як на рис. 3.38.

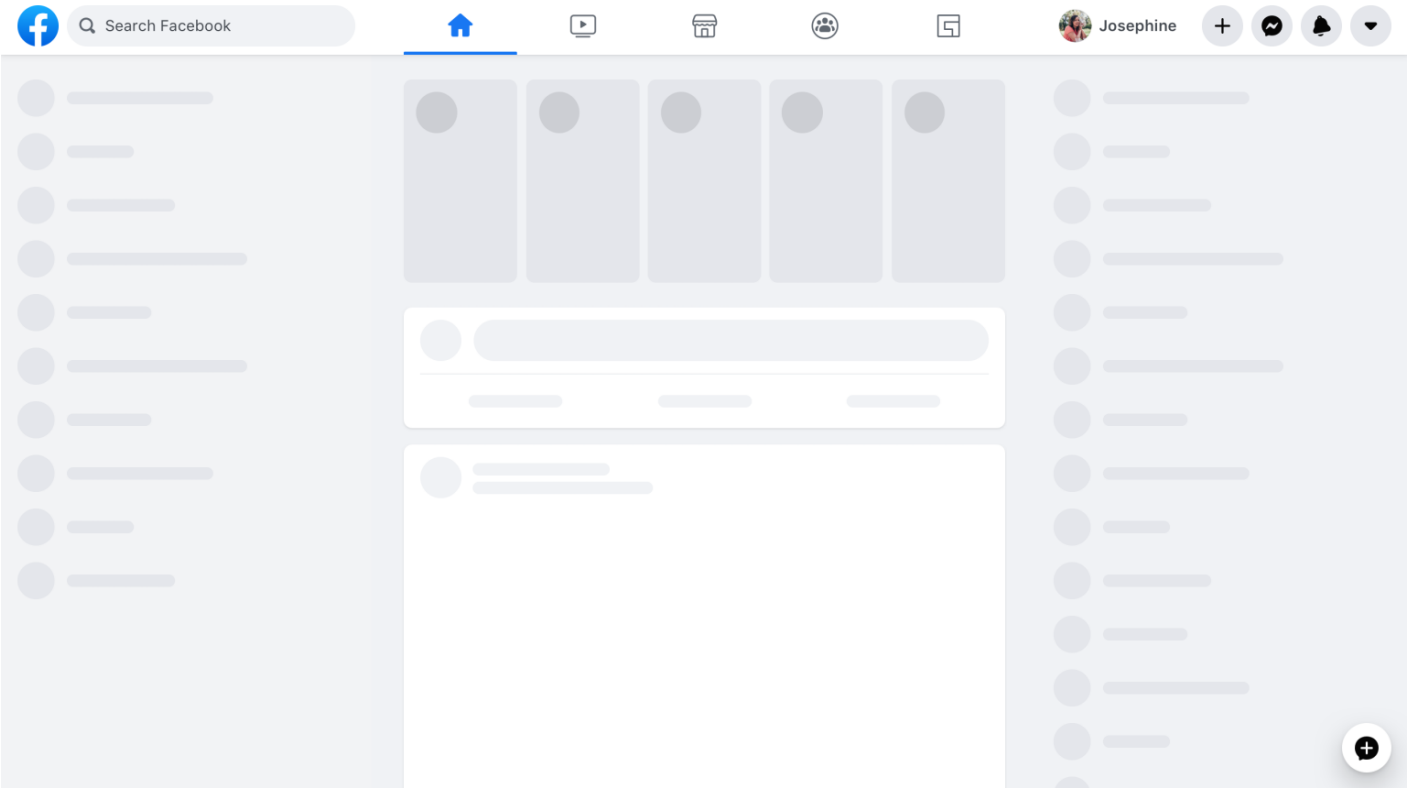


Рис. 3.38. Екран завантаження *Facebook*

Але так як наш сайт не має такої кількості мікрофронтів як *Facebook* та логіка цим модулів не є такоє обширною та великою, то для сайту з оголошеннями міста не потрібна така логіка, бо вона лише ускладнить проєкт де це не потрібно.

Після створення та імплементації в вебзастосунок головної сторінки починаємо за аналогічним процесом створювати наступні сторінки. Наприклад код

```
const CategoryPosts = () => {
  const { type : string  = '' } = useParams();

  const posts : {title: string, description: s...  = shortPosts[type as 'inspiration' | 'travel' | 'beauty'];

  return (
    <div className={styles.wrapper}>
      <div>
        <ul className={styles.menu}>
          <li>
            <Link to="./../travel">travel</Link>
          </li>
          <li>
            <Link to="./../inspiration">inspiration</Link>
          </li>
          <li>
            <Link to="./../beauty">beauty</Link>
          </li>
        </ul>
      </div>
      <div className={styles['content-wrapper']}>
        {posts.map((item : {title: string, description: s... ) => (
          <BlogPost type={type} {...item} />
        ))}
      </div>
    </div>
  );
};


export default CategoryPosts;
```

для сторінки постів за категоріями виглядає так:

Рис. 3.39. Код сторінки для постів по категоріям

Як результат отримаємо вигляд як на рис. 3.40.

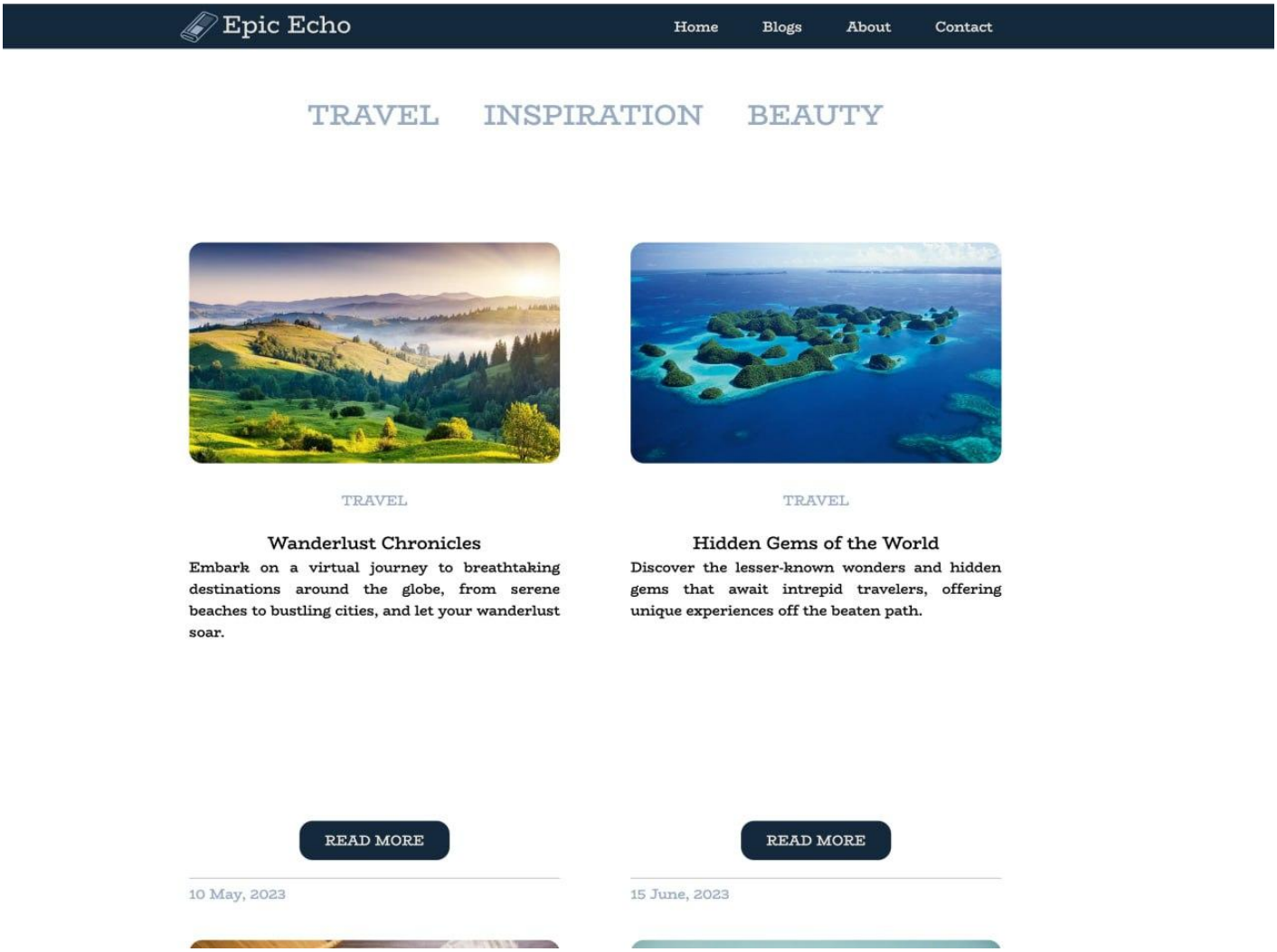


Рис. 3.40. Сторінка постів за категоріями

Як можна побачити, з правильним налаштуванням проєкту, архітектурою та створеними загальними компонентами подальша розробка стає дедалі легшою. Також треба пам’ятати, що це все робиться в різних мікросервісах, що в свою чергу дозволяє це все робити різними командами паралельно.

Для тестування даного вебзастосунку потрібно імітувати поведінку користувача. В даному випадку потрібно перевірити чи правильно налаштована маршрутизація та чи правильні сторінки показуються при переході. Наприклад на головна сторінка показується якщо в url нічого нема, окрім доменного ім’я. На рис. 3.41. це можна побачити.

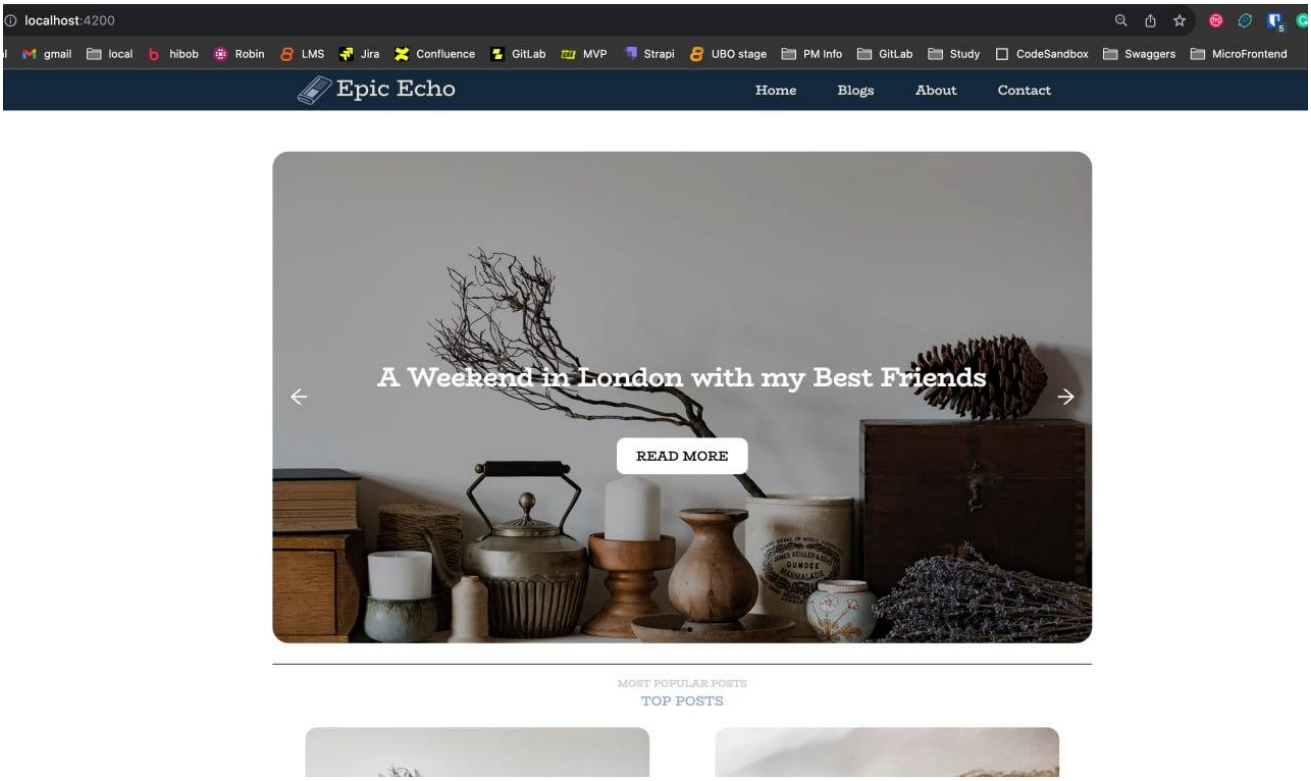
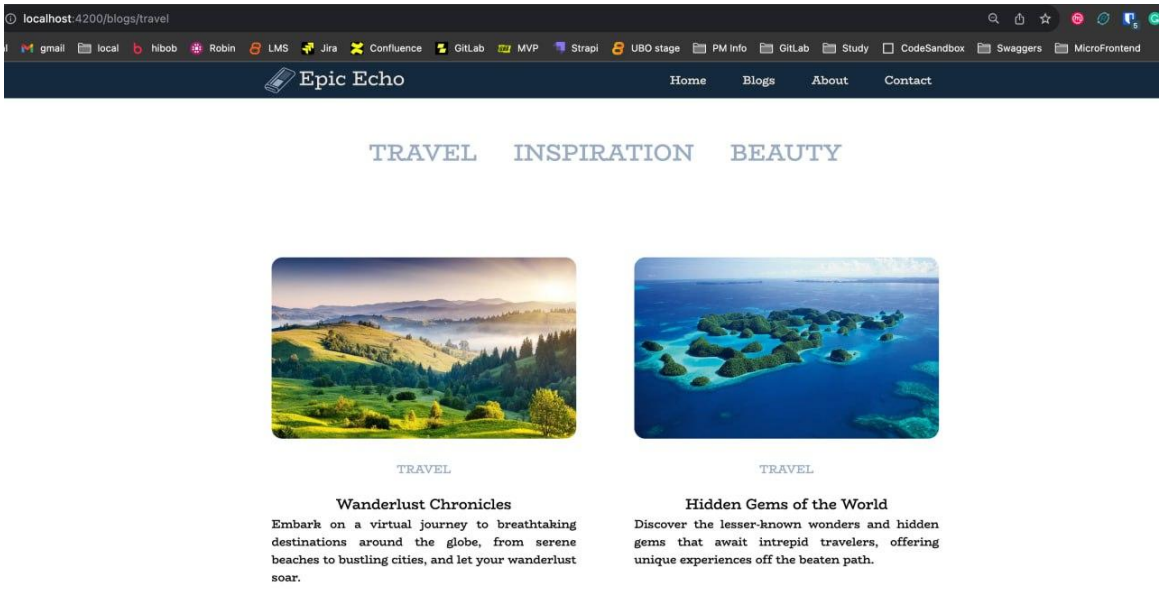


Рис. 3.41. Тестування головної сторінки

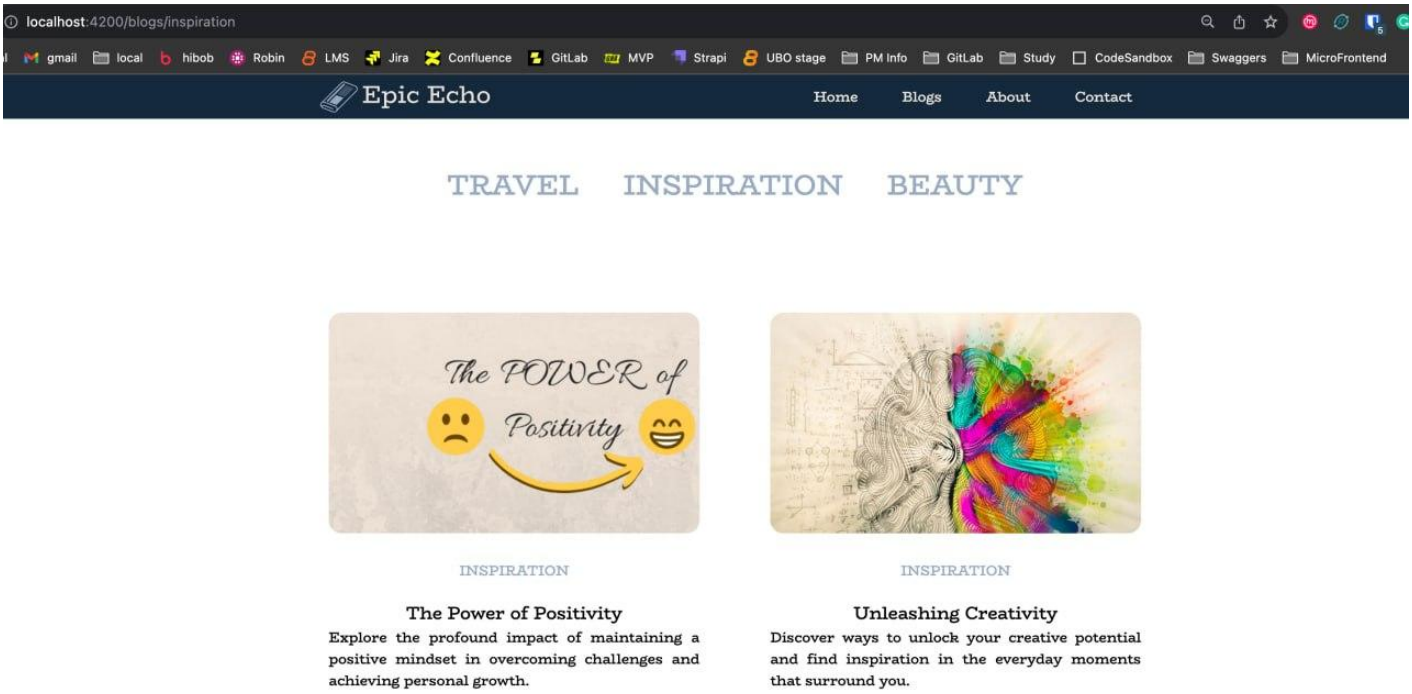
Після натискання на головному меню кнопки з написом «Blogs» нас має перевести на сторінку з статтями на тему подорожей. Це можна побачити на рис.



3.42.

Рис. 3.42. Тестування сторінки з статтями про подорожі

На рис. 3.42. в url можна побачити що з’явилося вставка «*/blogs/travel*», отже компонент та данні з’являються правильні. Після цього потрібно перевірити правильність роботи кнопок для зміни теми постів. При натисканні на кнопку з написом «*insparation*» нас має перевести на url «*/blogs/inspiration*». Тобто в даному випадку 2 частина url є динамічної, і завдяки бібліотеці *react-router-dom* не потрібно вказувати всі можливі варіації, треба лише вказати що це параметр завдяки «:». І завдяки цьому параметру, в компоненті можна дізнатися за яким фільтром потрібно



показувати статті.

Рис. 3.43. Тестування параметру в url

3.4. Процес релізу вебзастосунку з використанням Nx

Після розробки локально, постає питання релізу даного вебзастосунку, для того щоб його побачили інші користувачі. Так як вебзастосунки не потрібно

завантажувати собі на пристрій, то для користувачів завжди останньої версії яка доступна.

Загалом для зручності тестування між локальної розробкою та вебзастосунком для кінцевих користувачів який доступний за певним доменним ім'ям є ще один вебзастосунок. Зазвичай він називається «стейдж». Він існує для того, щоб тестувальники могли знайти помилки перед тим, як оновлювати вебзастосунок далі.

Nx дуже гнучкий у даному питанні. Завдяки вбудованим командам можна додати файл для розгортання коду на стейдж однією командою. Це команда «*npx nx g ci-workflow --ci=github*» яка додає файл *ci.yml* для створення автоматизованої послідовності етапів в сервісі для зберігання коду, як то github чи gitlab.

```yaml
name: CI
on:
  push:
    branches:
      - main
  pull_request:

jobs:
  main:
    name: Nx Cloud - Main Job
    uses: nrwl/ci/.github/workflows/nx-cloud-main.yml@v0.13.0
    with:
      number-of-agents: 3
      parallel-commands: |
        npx nx-cloud record -- npx nx format:check
      parallel-commands-on-agents: |
        npx nx affected -t lint,test,build --parallel=2

  agents:
    name: Nx Cloud - Agents
    uses: nrwl/ci/.github/workflows/nx-cloud-agents.yml@v0.13.0
    with:
      number-of-agents: 3
```

Цей файл виглядає так:

Рис. 3.44. Файл з описом процесів для проекту

Подальший шлях залежить від потреб проєкту. Можливий варіант коли потрібен спільний релізний процес. Або є інший варіант, де потрібен розгалуджений релізний процес. Різниця лише в тому, чи все мікрофронти одразу релізяться, чи ні.

Усередині першого варіанту виявляється недолік, пов'язаний з можливістю затримок у випуску через залежності між різними командами. Якщо одній з команд потрібно провести випуск свого мікросервісу, а інша команда має невиправлені помилки в своєму мікросервісі, це може вплинути на час випуску першої команди. Такі затримки можуть виникнути через те, що в системі існують взаємозалежності між різними мікросервісами, і одна команда не може продовжити роботу до виправлення помилок іншою командою.

Однак існують способи уникнення цих проблем. Локальне тестування мікросервісів може бути ефективним інструментом для виявлення помилок та їх виправлення на етапі розробки, перед випуском коду. Також можливо встановити домовленість між командами щодо графіка випусків. Це означає, що кожна команда зобов'язується випускати свої мікросервіси відповідно до певного графіка, що робить можливим уникнення затримок через несправності в інших сервісах.

Така стратегія може сприяти покращенню координації між різними командами та зменшенню ризиків виникнення конфліктів під час випуску. За допомогою встановлення чітких правил та процедур для релізів, команди можуть ефективно взаємодіяти та забезпечувати стабільність та швидкість розробки. Це також допомагає підвищити довіру між розробниками та забезпечити плавний процес впровадження нового функціоналу чи виправлення помилок.

ВИСНОВКИ ДО РОЗДІЛУ 3

У даному розділі кваліфікаційної роботи здійснено проектування вебзастосунку та було реалізовано цей вебзастосунок з використанням передових технологій React та Nx в інтегрованій середовищі розробки WebStorm.

Використання бібліотеки React дозволило створити динамічний та гнучкий інтерфейс, забезпечуючи високу швидкість та реактивність застосунку. Інструмент Nx виявився незамінним для побудови легко масштабованого проекту, забезпечуючи ефективне керування кодом та можливого залучення багатьох команд розробників.

Також було описано процес налаштування маршрутизації, що дозволило забезпечити коректну навігацію в застосунку. Це стало ключовим аспектом для створення зручного та інтуїтивно зрозумілого досвіду для користувачів.

Було описано релізні процеси, які можливо автоматизувати за допомогою інструментів Nx, описано підтримку безперервну інтеграцію та постійну доставку, забезпечуючи стабільні та надійні версії застосунку.

Окрім цього були описані всі варіації використання мікросервісної архітектури, що дозволить ефективніше займатись менеджментом розробки проєкту.

У результаті був продемонстрований готовий працюючий додаток для оголошень, який є з закладеною чистою архітектурою для легкого розширення функціоналу. Цей додаток працював, як показало тестування.

ВИСНОВКИ

В епоху стрімкого розвитку технологій та зростання вимог до сучасних веб-додатків, створення високоефективних та гнучких вебзастосунків стає складним завданням, що потребує не лише ретельного вивчення сучасних архітектурних рішень та технологій.

У роботі було проведено аналіз популярних мікросервісних фреймворків та бібліотек, визначено їхні переваги та недоліки, а також проведено порівняльний аналіз з альтернативними архітектурними рішеннями. Окрема увага була приділена питанню вибору між монорепозиторієм та різними репозиторіями в контексті мікросервісної архітектури. Для даного проекту була обрана архітектура монорепозиторію, так як застосунок не є настільки розгалудженим для впровадження мультирепозитоного підходу.

В рамках кваліфікаційної роботи на основі аналізу різних варіацій підходів та рішень було реалізовано мікросервісний вебзастосунок для оголошень міста з використанням технологій та інструментів, таких як React, Nx у середовищі розробки Webstorm.

Використання бібліотеки React стало не лише ключовим, але й невід'ємним елементом проекту, що дозволило створити динамічний та ефективний інтерфейс. React вирізняється своєю гнучкістю та можливістю швидко реагувати на зміни, що є критичними у веб-розробці. З його допомогою вдалося створити користувальницький інтерфейс, який відповідає сучасним стандартам, забезпечуючи при цьому відмінну користувацьку взаємодію та збільшуючи загальний рівень зручності використання нашого вебзастосунку.

Використання Nx як інструменту для монорепозиторію виявилося ключовим в аспекті ефективного управління кодом та масштабованості проекту. Nx дозволяє легко організовувати та управляти великим обсягом коду в одному репозиторії, спрощуючи розробку та підтримку проекту. Завдяки цьому інструменту, є

можливість швидко реагувати на зміни, ефективно співпрацювати в команді та забезпечувати стабільну роботу нашого вебзастосунку навіть у великих проектах.

WebStorm, в якості інтегрованого середовища розробки, виявилося невід'ємним помічником, надаючи розробникам потужний інструментарій для комфортної та продуктивної роботи. Його функціонал відзначався не лише розширеними можливостями, але й високою ефективністю, сприяючи підвищенню якості та швидкості розробки.

Однією з ключових переваг WebStorm була його здатність забезпечити ефективну навігацію по коду. Інтелігентний пошук та підказки значно полегшували розробникам знаходження необхідних елементів коду, що дозволяло швидше переходити між частинами проекту та зосереджуватися на конкретних завданнях.Однією з тем роботи був релізний процес мікросервісного вебзастосунку. Описаний процес надавав інсайти в аспекти розгортання, відмічаючи важливість безперервної інтеграції та безперервної доставки для забезпечення стабільності та актуальності продукту.

У підсумку, слід відзначити, що використання мікросервісної архітектури для розробки веб-застосунків для оголошень міста є не лише сучасним, але й надзвичайно ефективним рішенням у світі програмування. Мікросервісна архітектура, завдяки своїм ключовим перевагам, таким як гнучкість, масштабованість та швидкість розгортання, визначає нові стандарти в розробці програмного забезпечення. Розбиття великої системи на невеликі, автономні сервіси не лише сприяє ефективному керуванню їхнім розвитком та підтримкою, але й значно спрощує процес інтеграції нових функцій.

Загалом, дана робота не лише розкрила ключові аспекти розробки веб-застосунків для оголошень міста на базі мікросервісної архітектури, але й виявила, що такий підхід дозволяє будувати не лише сучасні, але й стійкі до змін та ефективні додатки, які в повній мірі задовольняють потреби користувачів у сфері оголошень та дозвільних сервісів.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Чиста архітектура. Роберт Мартин. Видавництво «Фабула» - Львів, 2019 рік -332 с. : іл. Бібліогр.: с 23 – 159.
2. Building Microservices: Designing Fine-Grained Systems Springfield. Sam Newman. Видавництво «o'reilly», 2021 рік - 500 с.
3. Microservices Patterns: With examples in Java. Chris Richardson. Видавництво «o'reilly», 2019 рік - 544 с.
4. Monolith to Microservices: Evolutionary Patterns to Transform Your Monolit. Sam Newman. Видавництво «o'reilly», 2019 рік - 284с.
5. Designing Data-Intensive Applications: The Big Ideas Behind Reliable, Scalable, and Maintainable Systems. Martin Kleppmann. Видавництво «o'reilly», 2019 рік - 590 с.
6. Build a micro-frontend application. Boston. [Електроний ресурс]:[Веб-сайт].- [Електронні дані]. -Режим доступу: https://blog.logrocket.com/build-micro-frontend-application-react/
7. Stephen Watts, Laura Shiff An Overview of Monolithic vs Microservices Architecture. [Електроний ресурс]:[Веб-сайт].- [Електронні дані]. -Режим доступу: https://www.bmc.com/blogs/microservices-architecture/
8. Oleg Dulin, Monolithic repository vs monolith. [Електроний ресурс]:[Веб-сайт].- [Електронні дані]. -Режим доступу: https://www.linkedin.com/pulse/monolithic-repository-vs-monorepo-oleg-dulin
9. Leonardo Losoviz, Monorepo vs Multi-Repo: Pros and Cons of Code Repository Strategies. [Електроний ресурс]:[Веб-сайт].- [Електронні дані]. -Режим доступу: https://kinsta.com/blog/monorepo-vs-multi-repo/
10. 11 Micro Frontends Frameworks You Should Know, USA, 2022. [Електроний ресурс]:[Веб-сайт].- [Електронні дані]. -Режим доступу: https://itnext.io/11-micro-frontends-frameworks-you-should-know-b66913b9cd20
11. Microservices information. [Електроний ресурс]:[Веб-сайт].- [Електронні дані]. -Режим доступу: https://martinfowler.com/tags/microservices.html

12. Microservices part 1. [Електроний ресурс]:[Веб-сайт].- [Електронні дані]. - Режим доступу: https://medium.com/microservices-part-1.

13. Bit JS Documentation. [Електроний ресурс]:[Веб-сайт].- [Електронні дані]. - Режим доступу: https://bit.dev/docs/quick-start/hello-world

14. Module Federation, Great Bratain. [Електроний ресурс]:[Веб-сайт].- [Електронні дані]. -Режим доступу: https://webpack.js.org/concepts/module-federation/ (дата звернення: 11.09.2023).

15. Single SPA Documentation. [Електроний ресурс]:[Веб-сайт].- [Електронні дані]. -Режим доступу: https://single-spa.js.org/docs/getting-started-overview

16. Systemjs Documentation. [Електроний ресурс]:[Веб-сайт].- [Електронні дані]. - Режим доступу: https://github.com/systemjs/systemjs

17. Piral Documentation. [Електроний ресурс]:[Веб-сайт].- [Електронні дані]. - Режим доступу: https://github.com/smapiot/piral

18. Nx Documentation. [Електроний ресурс]:[Веб-сайт].- [Електронні дані]. -Режим доступу: https://nx.dev/getting-started/intro

19. WebStorm Documentation. [Електроний ресурс]:[Веб-сайт].- [Електронні дані]. - Режим доступу: https://www.jetbrains.com/webstorm/

20. Awesome Microservices. [Електроний ресурс]:[Веб-сайт].- [Електронні дані]. - Режим доступу: https://github.com/mfornos/awesome-microservices